The HOL-OCL book
Version 0.9.0

ETH Technical Report 525

The HOL-OCL Book

Version 0.9.0

http://www.brucker.ch/research/hol-ocl/

Achim D. Brucker Burkhart Wolff

August 15, 2006

Information Security
Swiss Federal Institute of Technolgy (ETH)
8092 Zurich
Switzerland

Important note:
This manual describes HOL-OCL 0.9.0 with referential universes and smashed collection types.
The manual of version 0.9.0 is also available as technical report number 525 from the department of computer science, ETH Zurich.

Contents

Part I.

Motivation and Introduction

Chapter 1.

Introduction

1.1. Motivation

Building safe and secure software systems requires engineering techniques. Just as using "blueprints" in common engineering practice, the development of complex software systems requires a detailed specification describing its data structures and its desired behavior. Specification documents can vary in its precision from textual descriptions over structured text up to formal specification using a language based on mathematical logic such as Z [26] or VDM [27]. Depending on the precision (or *formality*), different computer-supported techniques can be applied that assure the consistency of specification documents. For example, *type-checking* or *wellformedness-checking* can be used to find contradictions in requirement and design documents of a system and thus provide an a-priori *analysis*. Moreover, more involved techniques can assure the correct transition from a specification document to its implementation in "real code"; such an a-posteriori analysis techniques is called *validation*. These techniques and their computer-support are summarized under the term *formal methods*. Obviously, the power of these techniques crucially depends on the degree of formality.

Producing formal specifications and maintaining their consistency during system development is a task that requires substantial efforts and training. This holds to an even larger extent for the validation phase, where techniques such as *model-based testing*,*model-checking* or even interactive *theorem proving* were applied. For this reason, the acceptance of formal methods has been very reluctant in industrial practice so far, although it is meanwhile widely accepted that specification and testing activities outweigh by far the costs of the implementation phase of a large system and that formal methods have a positive effect here. Rather, the overwhelming need for specification led to the development of semi-formal specification documents, that have their roots in light-weighed graphical notations. These semi-formal kinds of documents became eventually a more and more defined semantics and were annotated by programs or abstract test code. Thus, instead of using mathematical notation such as Z or the Hoare Calculus, there is a trend to introduce formal specification techniques such as pre and post conditions through the backdoor, leading to a more gradual transition of the industrial practice by bridging the gap to software developers by languages and notations they are familiar with.

The need for computer-support of these "light-weighed" specification methods led to a standardization effort and the *Unified Modelling Language* (UML), which achieved remarkable acceptance in the industry. The UML offers an integrated object-oriented development methodology ranging from "object-oriented modeling" to code-generation in "model-based architecture" development. The UML is continuously defined by the Object Management Group (OMG) [41], an open standardization committee. The UML provides standards for graphical notations, their representation in abstract syntax (called *meta-models*) and partly also their semantics. These notations comprise among others *activity charts*, *sequence charts*, *class diagrams* and *state charts*. The latter two notations are of particular interest from the perspective of formal methods, since they represent forms of data-oriented and behavioral modeling, thus well-known concepts in a new shape. Moreover, since UML 1.3, a logical annotation formalism, called the *Object Constraint Language* (OCL) is a part of the UML and is heavily used in the specification documents of the meta-models of the UML itself.

To sum up, from the perspective of formal methods, the success of CASE tools supporting the UML [40] opens a door for bringing formal methods a step closer to industry. However, to turn this vision into reality, several challenges need to be faced: first, the semantics of UML/OCL is conceptually much closer to an object-oriented programming language than to a traditional logic, although OCL comprises a version of predicate logic and arithmetic. Second, considerable effort has to be invested to cope with the object-oriented features of OCL in a logically clean way, allowing for adequate symbolic computations. Third, a considerable research effort has to be invested to develop an adequate proof methodology for object-oriented modeling as it is meanwhile established in the user community of UML/OCL.

As a technical basis for a long-term effort to meet these challenges, we developed HOL-OCL, an interactive proof environment for acsocl. The main design goals of HOL-OCL are summarized as follows:

- HOL-OCL follows semantically the OCL 2.0 standard [41],
- HOL-OCL provides technical support for importing UML/OCL models (i. e., class-diagrams at the moment, start charts are under preparation) in form of XMI-formats generated by conventional UML/OCL modeling tools, and
- HOL-OCL is intended to analyze UML/OCL models with all the means provided by an up-to-date interactive, tactic-controlled theorem proving environment such as Isabelle [3].

In itself, HOL-OCL also provides achievements for the definition and development of the language UML/OCL itself:

- It defines and supports several *syntactic formats for OCL* (one more text-oriented for programmers and software developers, another more mathematical one for proof engineers).
- It defines a machine-checked *formalization of the semantics* as described in the standard 2.0. This is implemented as a conservative, shallow embedding consisting of OCL into the higher-order logic (HOL) instance of the interactive theorem

prover Isabelle. This includes a typed, extensible UML data models supporting inheritance and subtyping inside the typed lambda-calculus with parametric polymorphism. As consequence of conservativity with respect to HOL, we can guarantee the consistency of the semantic model (see 3 and in particular 2.1 for an in depth discussion). In fact, we consider it as an important contribution to provide a candidate for a "beau ideal" of the OCL semantics.

- The standard 2.0 postulates requirements to the semantics of OCL operators. The OCL semantics—which is contained in an appendix of the standards documents—is not formally related to these requirements. We provide formal proofs that our formalization of the OCL semantics indeed meets the requirements.
- In some (minor) parts, our work detected formal contradictions or inconveniences with respect to derived calculi. We strive for compliance with the standard 2.0. With respect to the faithfulness of our representation, see Sec. 4.5),.
- It provides several derived calculi for UML/OCL that allows for formal derivations establishing the validity of UML/OCL formulae. Automated support for such proofs is also provided, however since HOL-OCL comprises predicate logic with equality and a typed set-theory, the validity of a formula is undecidable and the logic is inherently incomplete with respect to the class of standard models of HOL [8].
- It represents a technical framework (including a graphical front-end in form of ProofGeneral [5] and a programming interface for SML) enabling to implement particular *formal methods* based on UML/OCL.

These future *method supports* can be, for example,

- *consistency checking* of UML models (e. g., is a class-diagram implementable, i. e., do states exist that satisfy the invariants? Contradict pre-conditions and post-conditions to system invariants?),
- proving *refinements* between UML models,
- *code verification* of concrete programs with respect to a UML model by means of a Hoare-Calculus,
- proving *temporal properties* of UML models (e. g., do all reachable states satisfy a security-property?),
- *automated test-case generation* out of UML models for concrete code.

1.2. How to Read This Document

This book has three different aspects and describes them at various levels of detail. Therefore, we divided the material into three parts which can be read largely independently from each other:

- Part I gives an overall introduction and motivation. In particular, it introduces into the foundational notations and the target language.
- Part II presents the *concepts* of the underlying ideas and formalization techniques behind HOL-OCL.
- Part III explains the HOL-OCL system from a users perspective and can be seen as a kind of "users manual."

Further, in the appendix a complete companion of the Isabelle theory files is provided, including the technical details of HOL-OCL and reference material such as, e. g., tables that compare different OCL syntaxis.

Depending on your background knowledge and interests, we recommend kind of cherry-picking strategy for reading this book. For example,

- If you are a HOL-OCL user and want to get a quick "hands-on" experience you may start with the installation instructions (chapter 7) and start reading with chapter 1 and chapter 2. You can then continue directly with chapters 9 and 10.
- If you are interested in the ideas and techniques of HOL-OCL we recommend (after reading the introductory chapters I and 2) an in depth study of chapters 3 to 5.
- If you are an Isabelle hacker and are interested in the details necessary for theorem proving work you can directly start with the theory-files in the appendix, i. e., chapter B. However, we advise to start with the more conceptual descriptions in 5.

1.3. Typographic Conventions

The following typographic conventions appear in this book:

- Pure OCL specifications are either written inline like `self.s->includes(5)` or together with their context specification:

  ```
  context A:
    inv: self.s->includes(5)
  ```

  Keywords are printed in a blue typeface.
- OCL formulae that are interpreted within HOL-OCL, i. e., are written inline as *self*.*s*->`includes`(5), or alternatively in mathematical syntax as: $5 \in (\mathit{self}.s)$. As you can see, we use a (shorter) mathematical notation for OCL expressions. This notation is introduced as an alternative concrete syntax (see Table A.2 for a syntax comparison). Overall, HOL-OCL supports both notations, but we prefer the mathematical one for semantic definitions and proof work.

- We use a color coding to distinguish OCL and HOL sub-expressions in formulae containing both, e. g.:

$$\cup \equiv \mathrm{lift}_2\Big(\mathrm{strictify}(\lambda X.\ \mathrm{strictify}(\lambda Y.\ \mathrm{AbsSet}_{\llcorner}\ulcorner\mathrm{RepSet}\,X\urcorner \cup \ulcorner\mathrm{RepSet}\,Y\urcorner{}_{\lrcorner}))\Big)\,.$$

  or

$$\_ + \_ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda x.\ \mathrm{strictify}(\lambda y.\ {}_{\llcorner}\ulcorner x\urcorner + \ulcorner y\urcorner{}_{\lrcorner})))\,.$$

  Overall, HOL expressions are printed using the default color. We resolve ambiguousities between the underlying mathematical syntax (i.e., HOL) and the OCL level by using colors: Expressions that are internally used within HOL-OCL, like the lifting operator $\llcorner\_\lrcorner$ are printed in a green typeface. Using our mathematical OCL syntax, expressions on the OCL level , like $\_ \wedge \_$, are written in a magenta typeface. For the concrete syntax presented in the standard, e. g., _ `and` _, we use a magenta typeface. In rare cases, notable for the arithmetic operators like $\_ + \_$, we deviate from this scheme out of technical reasons.

- HOL formluae are written using the usual mathematical notion, i. e., $s \in S$. Theory files for HOL-OCL and Isabelle/HOL are printed as follows:

```
theory royals_and_loyals
imports
  OCL
begin
  load_xmi "royals_and_loyals_ocl.xmi"
end
```

  Keywords are printed in a green typeface.

- SML code fragments are written inline like `fn x => 2 * x` or in display style:

```
datatype OclType = Integer | Real | String | Boolean
                 | OclAny | Set of OclType (* ... *)
```

  Keywords are printed in a blue typeface.

Further, we mark problems and extensions to the OCL standard as follows:

- Errors in the standard are marked by with a danger sign on the margin throughout this document, and our definitions can be seen as our proposal for repair. *Glitch*

- In some cases we see our proposals as an extension of the standard, these cases are marked with an exclamation sign on the margin. *Extension*

1.4. Acknowledgements

- HOL-OCL uses the component su4sml which is developed by Achim D. Brucker, Jürgen Doser, and Burkhart Wolff as the underlying UML repository.

- Internally, su4sml uses the XML-parser Functional Extensible Markup Language (XML) Parser (FXP) [2].
- During his semester thesis, Simon Meier checked the library for completeness and substantially extended the theory.

Chapter 2.

Background

2.1. Higher-order Logic

Higher-order Logic (HOL) [14, 8] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, e. g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers.

HOL is based on the typed λ-calculus—i. e., the *terms* of HOL are λ-expressions. The *application* is written by juxtaposition $E\ E'$, and the *abstraction* is written $\lambda\, x.\ E$. Types may be built from *type variables* (like α, β, ..., optionally annotated by *type classes* as in $\alpha :: \mathit{order}$ or $\alpha :: \mathit{bot}$) or *type constructors*(like bool or nat). Type constructors may have arguments (as in α list or α set). The type constructor for the function space $\Rightarrow$ is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ have the alternative syntax $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow \mathrm{bool}$, where bool is the fundamental logical type. We use infix notation: instead of $(\_ = \_)\ E_1\ E_2$ we write $E_1 = E_2$. The logical connectives $\_ \wedge \_$, $\_ \vee \_$, $\_ \rightarrow \_$ of HOL have type $[\mathrm{bool}, \mathrm{bool}] \Rightarrow \mathrm{bool}$, $\neg\_$ has type $\mathrm{bool} \Rightarrow \mathrm{bool}$. The quantifiers $\forall\_.\_$ and $\exists\_.\_$ have type $[\alpha\ \mathrm{set}, \alpha \Rightarrow \mathrm{bool}] \Rightarrow \mathrm{bool}$. The quantifiers may range over types of higher order, i. e., functions.

The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type α set can be defined isomorphic to functions of type $\alpha \Rightarrow \mathrm{bool}$; the elementhood $\_ \in \_$ has then type $[\alpha, \alpha\ \mathrm{set}] \Rightarrow \mathrm{bool}$ and corresponds basically to the application; in contrast, the set comprehension $\{\_.\_\}$ has type $[\alpha\ \mathrm{set}, \alpha \Rightarrow \mathrm{bool}] \Rightarrow \alpha\ \mathrm{set}$ and corresponds to the λ-abstraction. The definition of $\_ \cup \_$, $\_ \cap \_$ and set complement is standard, as well as bounded versions of the quantifiers $\forall\_ \in \_.\_$ and $\exists\_ \in \_.\_$ of type $[\alpha \Rightarrow \mathrm{bool}] \Rightarrow \mathrm{bool}$.

Thus, HOL allows for specifications in a very natural (mathematical) way, similar to naive set theory. In contrast to the latter, however, HOL is a consistent logical system (relative to ZFC) provided that all formulae can be type-checked, which we always assume throughout this text and which is technically done behind the scenes.

The *modules* of larger logical systems built on top of HOL are Isabelle *theory files* (or just: *theories*, if no confusion arises). Among many other constructs, they con-

[1] to be more specific: *parametric polymorphism*

tain type and constant declarations as well as axioms. Since stating arbitrary axioms in a theory is extremely error-prone and should be avoided, only very limited forms of axioms should be admitted and the constraints (both syntactical and semantical) checked by machine. These fixed blocks of declarations and axioms described by a syntactic scheme were called *conservative theory extensions* since and extended theory is consistent ("has models") provided the original theory was. Four different conservative extensions were discussed in the literature: *constant definition*, *type definition*, *constant specification*, *type specification*(see [17] for an in depth discussion, which also includes the correctness proofs). For example, the mostly used constant definition consists of a constant declaration

$$c \ :: \ \tau$$

and an axiom of the form:

$$c \ = \ E$$

where c has not been previously declared, the axiom is well-typed, E is a closed expression (i. e., does not contain free variables) and E does not contain c (no recursion). A further restriction forbids type variables in the types of constants in E that do not occur in the type τ. As a whole, a constant definition can be seen as an "abbreviation" which makes the conservativity of the construction plausible (but see [17] for the hairy details), and the syntactic side-conditions are checked by Isabelle automatically.

The idea of an "abbreviation" is also applied for the conservative *type definition* of a type $(\alpha_1, \ldots, \alpha_n)T$ from a set $\{x \mid P(x)\}$.

In this case, the set of type constructors is extended by the constructor T of arity n. The predicate P of type $\tau \Rightarrow$ bool for a base type τ constructs a set of elements τ set; the new type is defined to be isomorphic to this set. Technically, this isomorphism is stated by the declaration of two constants representing the abstraction and the representation function and by two axioms over them. More precisely, the constant Abs_T of type $\tau \Rightarrow (\alpha_1, \ldots, \alpha_n)T$ and the constant Rep_T of type $(\alpha_1, \ldots, \alpha_n)T \Rightarrow \tau$ were declared. The two isomorphism axioms have the form:

$$Abs_T(Rep_T(x)) = x$$

and

$$P(x) \Longrightarrow Rep_T(Abs_T(x)) = x$$

where the precise meaning of the $\_ \Longrightarrow \_$-arrow will become apparent in the next section; for the moment, we can consider it equivalent to the logical implication $\_ \to \_$. The type definition is conservative if the proof obligation $\exists\, x. P(x)$ holds; this assures that the type is non-empty as required by the semantics of HOL.

The Isabelle/HOL library (and the libraries of similar HOL systems) prove, that the typed set theory including least fixedpoint theory, order theory including well-founded recursion, number theory including real number theory and theories for data-structures like pairs, type sums and lists can be build on top of the HOL core-language entirely by these conservative definitions. A large part of these theories consists in deriving rules over the defined operators, in particular those that allow for simplification and (recursive) computation. This methodology is also applied for HOL-OCL.

2.2. Concepts and Use of Isabelle/HOL

Isabelle [37] is a generic theorem prover of the Logic for Computable Functions (LCF) prover family implemented in Standard Meta Language (SML). We heavily use the possibility to build SML programs performing symbolic computations over formulae in a logically safe way on top of the logical core engine: this is how the proof procedures of HOL-OCL are built technically. Isabelle/HOL offers support for checks for conservativity of definitions, data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

Isabelle's proof engine can directly process *natural deduction rules*: $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, also written $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$, is viewed as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}". Here, $\_ \Longrightarrow \_$ denotes the built-in *meta-implication* of Isabelle. With meta-implications, also more complex rules like: if assumption B can be inferred from assumption A, infer $A \to B$ (also called: *implication introduction*) can exactly be expressed in Isabelle by: $(A \Longrightarrow B) \Longrightarrow A \to B$. In the mathematical literature, such natural deduction rules were also written as:

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}} \qquad\qquad \frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}$$

A *proof state* in Isabelle contains an implicitly conjoint sequence of Horn-clause-like rules called *subgoals* $\phi_1, \ldots, \phi_n$ and a *goal* ϕ. Logically, subgoals and the goal are connected to a theorem of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$. In order to cope with quantifiers naturally occurring in logic, subgoals have a slightly more general format than just Horn-clauses: variables may be bound by a built-in *meta-quantifier*:

$$\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

The meta-quantifier $\bigwedge$ helps to capture the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be logically considered as free variables. Further, Isabelle supports *meta-variables* (written $?x, ?y, \ldots$), which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification and occur only inside proofs.

The initial proof state is built from the trivially true theorem $\phi \Longrightarrow \phi$ for any (typed) formula ϕ. A theorem is proven if a final proof state of the form ϕ could be reached by *tactics* i. e., SML functions allowing for the transformation of proof states. It is a key feature of Isabelles design that all tactics are based on a few operations provided by the logical core engine of Isabelle. Moreover, these core operations log all logical operations in a *derivation tree* called *proof-object*; thus, if someone has serious doubts on the correct implementation of Isabelles proof procedures (i. e., tactics), he/she may generate the proof objects and check the derivations by an independent program following the rules of the logic.

Besides the SML-based programming interface, there is also an own input language to Isabelle theories, called Intelligible semi-automated reasoning (Isar). Isar allows for writing specifications consisting of definitions, proofs, and technical setups for the prover in a fairly readable way. In particular, Isabelle can generate LaTeX-documentation while checking the entire hierarchy of theories (this is, what *this* document is ...).

Isabelle theories written in Isar are supported by a fairly powerful Emacs-based front-end called ProofGeneral. HOL-OCL is in fact integrated and customized for this front-end, such that HOL-OCL specific Isar commands and HOL-OCL specific fonts can be used.

2.3. A Short Introduction into UML/OCL

Now we introduce the concepts of the target of our work, namely Unified Modelling Language (UML) and Object Constraint Language (OCL). As mentioned, Unified Modelling Language (UML) provides a variety of diagram types for describing dynamic (e. g., state charts, activity diagrams) and static (e. g., class diagrams,object diagrams object ␣␣diagram) system properties.

One of the more prominent diagram types of the UML is the *class diagram* for modeling the underlying *object-oriented* data model of a system. The class diagram in Figure 2.1 illustrates a simple accounting scenario where customers can own different kinds of accounts and transfer money between them. In more detail: customers are modelled as as a *class*. There are further classes modelling the different account types, a checkbook with checks and also transactions. A class does not only describe a set of *object instances*, i. e., record-like data consisting of *attributes* such as `balance`, but also *operations* defined over them. The class `Customer` in our example has only attributes, namely `name`, `address`, `gender`, and `title`. The different account types are organized in a hierarchy of subtypes denoted by an arrow, i. e., between `Account` and `BankAccount`. Such an subtype relation is called *inheritance* in the object-oriented paradigm. Sometimes this is also called *specialization*, i.e, a `CurrencyTrading` specializes a `BankAccount`. Further, it is characteristic for the object-oriented paradigm that the functional behavior of a class and all its methods are also accessible for all subtypes. A class is allowed to redefine an inherited method, as long as the method interface does not change; this is called *overriding*, as it is done in the example for the operation `makeWithdrawal()`.

Of course, there is a relation between customers and accounts. This is modelled in UML by an *association* `belongsTo`. An association can be constrained by *multiplicities*. In Figure 2.1, the multiplicities of the association `belongsTo` requires that every object instance of `Account` is associated with exactly one object instance of `Bank`. This captures the requirement that every account belongs to a unique bank. In the other direction, the association models that an instance of class `Bank` is related to a (non-empty) set of instances of class `Account` or its subtypes.

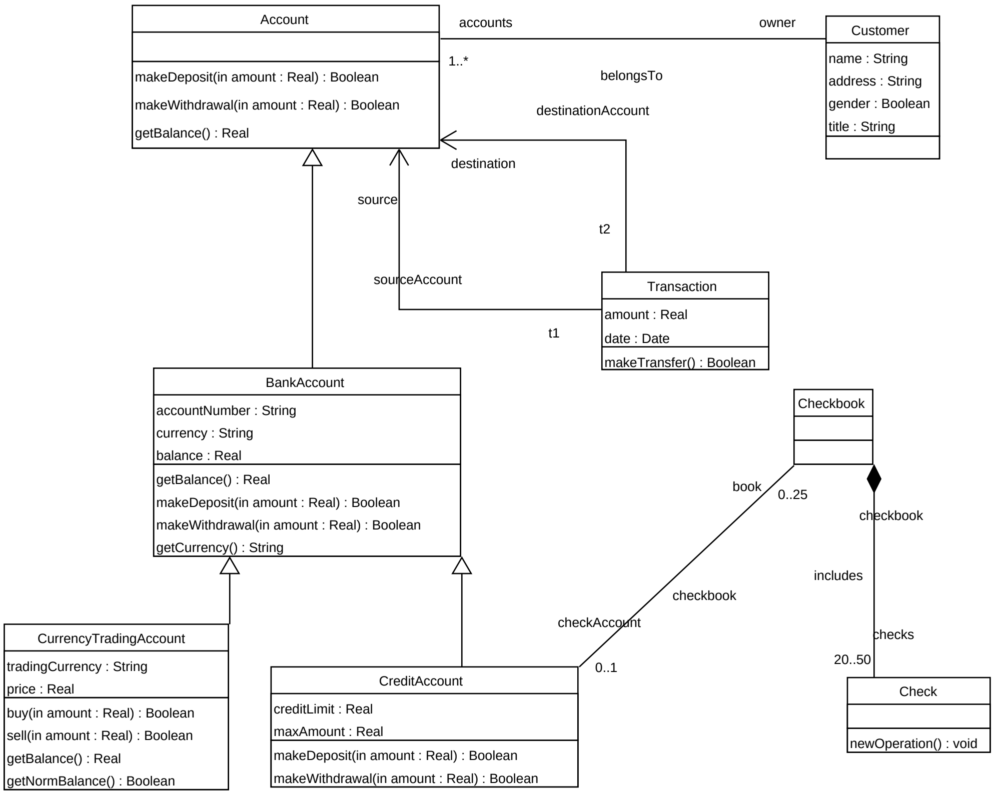


Figure 2.1.: Modeling a simple banking scenario with UML

Understanding OCL as a data-oriented specification formalism, it seems natural to refine class diagrams using OCL for specifying invariants, pre-conditions and post-conditions of operations. By specifying the following OCL constraints one can make the specification of the class `Account` much more precise, e. g., we can describe that `accountNumbers` are unique:

```
context
  BankAccount
  inv:  BankAccount.allInstances
          ->forAll(a1,a2 | a1<>a2 implies
                   a1.accountNumber <> a2.accountNumber)
```

Or constrain the effects of operations:

```
context BankAccount::getBalance():Integer
  post: result=balance

context BankAccount::getCurrency():String
  post: result=currency

context BankAccount::makeDeposit(amount:Integer):Boolean
  post: balance = balance@pre + amount

context BankAccount::makeWithdrawal(amount:Integer):Boolean
  post: balance = balance@pre-amount
        and currency = currency@pre
```

where in post-conditions `@pre` allows one to access the previous state.

In UML, class members can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive data types. Moreover, OCL introduces also recursively specified methods [41]; however, at present, a dynamic semantics of a method call is missing ([12] gives a short discussion of the resulting problems).

Note, that many diagrammatic UML-features can be translated to OCL-expression without losing any information, e. g., associations can be represented by introducing implicit set-valued attributes into the objects with a suitable data invariant describing the multiplicity. These transformations are already described in the UML-standard [40]. Some of these transformation were done during the process of loading an UML/OCL specification into HOL-OCL, e. g., we have not to provide a special treating for associations in our proof environment.[2]

2.4. A Note About Standards

OMG standards are developed in an open process leading to a variety of "standardization" documents. Especially for UML and OCL which have a long history and our particular choice needs some explanation.

[2]Direct support for associations is planned for the future.

OCL was introduced as an OMG specification language as additional document [38] completing the UML 1.1 standard [39]. In later releases of the UML standards of the version 1.x series the OCL standard was a chapter of the UML specification, e. g., [40, Chapt. 6]. As several shortcomings of the OCL 1.x specifications were discussed [50, 32, 21] the development of OCL 2.0 led to many documents describing the different intermediate states.

For our work we have chosen the following two documents as our main references:

1. *OMG Unified Modeling Language Specification* (excluding Chapt. 6 which describes OCL 1.Version 1.5[3] [40] which is publicly available at `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf`.

2. *UML 2.0 OCL Specification* (OMG Final Adopted Specification)[4] [41] which is publicly available at `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.pdf`.

This combination is also used by a many recent tools, especially the OCL type-checker [1] from the Dresden University which is integrated into our tool chain.

[3] also known as OMG document "`formal/03-03-01`"

[4] also known as OMG document "`ptc/03-10-14`"

Part II.

Theory

Chapter 3.

Foundations

3.1. An Overview of Embedding Techniques

A theory representing syntax and semantics of a programming or specification language in another specification language is called an *embedding*. While the underlying techniques are known since the invention of typed λ-calculus, it was not before the late seventies that the overall importance of *higher-order abstract syntax* (HOAS) [43] for the representation of binding in logical rules and program transformations [24] and for implementations [43] was recognized.

As an example, we revise the universal quantifier of HOL already introduced in Section 2.1: it is represented in higher-order abstract syntax (HOAS) by a constant $\mathrm{All} :: (\alpha \Rightarrow \mathrm{bool}) \Rightarrow \mathrm{bool}$, where the term $\mathrm{All}(\lambda\, x.\ P(x))$ is paraphrased by the usual notation $\forall\, x.\ P(x)$. This is in contrast to the usual textbook definition for predicate logic, where a free datatype for terms and predicates, explicit substitution and well-typedness functions over them is provided. This conventional representation requires explicit side-conditions in logical rules over quantifiers preventing variable-clashes and variable capture. The representation using *higher-order abstract syntax* (HOAS) has two advantages:

1. the substitution required by logical rules like $\forall\, x.\ P(x) \Longrightarrow P(t)$ can be directly implemented by the β-reduction underlying the λ-calculus, and

2. the typing discipline of the typed λ-calculus can be used to represent the typing of the represented language. For example, a multi-sorted first-order logic (having syntactic categories for arithmetic terms, list terms, etc.) is immediately possible by admitting expressions of type nat and α list.

In short, HOAS has the advantage of "internalization" of substitution and typing into the meta-language, which can therefore be handled significantly more general and substantially more efficient, which is a prerequisite for using Isabelle as an implementation platform.

When using HOAS-style semantic definitions, the technique is extended into a "shallow embedding" [9] of an object-language, a technique which is opposed to a "deep embedding" which treats syntax by free datatypes and semantics via semantic interpretation functions as common in logical textbooks.

A shallow embedding definition of the universal quantifier is, for example, $\mathrm{All}\, P \equiv (P = \lambda\, x.\ \mathrm{true})$ (the propositional function of the "body" of the quantifier must be equal to the function that yields true for any argument), a deep representation follows usual textbooks:

$$\mathrm{Sem}[\![\forall\, x.\ P(x)]\!]\gamma \equiv \begin{cases} \text{true} & \text{if } \mathrm{Sem}[\![P(x)]\!]\gamma[x := d] \text{ for all } d \\ \text{false} & \text{otherwise} \end{cases}$$

where we assume a meta-language with well-defined concepts such as if, otherwise, and for all, e. g., Zermelo-Fränkel-set theory.

As can be seen, shallow embeddings can have a remarkably different flavor in their semantic presentation, in particular when striving for conservativity as in the example above. However, since the usual inference rules were derived from these definitions (like the rule $\forall\, x.\ P(x) \Longrightarrow P(t)$ from above), they are finally proven as semantically equivalent.

If a shallow embedding is built entirely by conservative theory extensions (which implies that the language definition is consistent if the meta-language is; cf. Section 2.1), we also speak of a *conservative embedding*.

3.2. Embeddings of Specification Languages

Since shallow embeddings in itself are fairly well-known in principle, the question arises, what the fundamental problems and technical challenges exist for representing "real world" specification languages. The main difficulty stems from the fact, that the most successful shallow-embeddings (as, for example, HOL encoded in typed λ-calculus) are *designed* to fit to the underlying meta-language. In contrast, "Real world" languages are typically conceived solely on a kind of mathematical notation based on naive set theory and no specific experience in theorem proving in mind; in some cases, the definition process of a "real" specification language takes place in a "development by committee" process prone to all sorts of arcane compromises. While a deep embedding is always possible whenever a formal semantics exists, a shallow embedding may conflict with certain constraints the shallow technique imposes, be it on the representability of the binding structure, the type discipline and the semantics of the language.

As an example for a different outcome of language design decisions, one might consider the following pathological case: In standard logic textbooks (like [31]), the notion of a model also admits empty carrier sets for sorts or types. Therefore, in many textbook-driven research papers, a lot of consideration of these pathological cases is made. In a tool-oriented research approach, it suffices to see that β-reduction is simply not sound[1]. Given the fact that β-reduction is vital for higher-order logic provers and for the issue of internalization of substitution, this led to a modification of the model notion, i. e., to ruling out non-empty carrier sets and living with the methodological consequences such as side conditions in type definitions.

[1]Consider the two types a and b with some empty carrier set A and a non-empty carrier B. Since B^A is the empty set, $(\lambda\, x :: a \Rightarrow b.\ C)$ is an uninterpretable term and $(\lambda\, x :: a \Rightarrow b.\ C) = C$ does not hold for some constant C interpretable in B and some variable X

Thus, even fairly worked out semi-formal semantic descriptions of a specification language may be quite distant to a formal (i. e., machine-checkable) representation, and even a formal semantic representation may be quite distant to a set of derived rules that allow for the formal support of a particular *method* of this language. Providing paradigmatical embeddings and new techniques helpful for bridging this gap is the main goal of this work. To describe the problem domain in more detail, we use the following classification: Specification languages may be:

1. *process oriented*, i. e., their focus of description is the *behavior*, usually described as possible sequences (*traces*) of states or communication events,
2. *data oriented*, i. e., their focus of description is the structure of data a system processes, its states, and individual steps the system performs when making a transition from one state to another, and
3. *object-oriented*, which is essentially a data-oriented specification approach where object-oriented data structuring techniques such as inheritance and subtyping are emphasized.

Examples for process-oriented specification languages are temporal or modal logics (such as LTL, CTL, μ-calculus [15]) or process-algebras (such as CCS [34] or CSP [45]). Examples for data-oriented specification languages are VDM [27] or Z [47], examples for the object-oriented language class are Object-Z [46] or UML/OCL [40, 41].

3.3. Challenges of a Shallow Embedding of UML/OCL

Producing a shallow embedding of the key features of object-oriented modeling in HOL is the technically most challenging enterprise in this work, but also the one with the greatest potential of use in software engineering. A shallow embedding in HOL must capture the essence of OCL with respect to subtyping and inheritance in a type system that does not provide a notion of subtypes, and provide means for extensibility and reuse to capture the pragmatics of object orientation. Moreover, a formalization must capture that object-oriented specification is deeply intertwined with the notion of *state*, i. e., a map of object-references to objects (abstractions of pieces of memory) and its possible *state transitions*. Technically, these constraints have the form of class invariants or pre and postconditions of methods; OCL in class diagram is thus a typical data-oriented formal modelling technique.

We consider the following example: Since path expressions resulting from UML class-diagrams are basic elements of the OCL language, one can state that the attribute `a` of an object `self` must be `5` in the pre state: `self@pre.a = 5`. With respect to a state transition, we may state that `a` increases during a transition: `self@pre.a = self.a + 1`.

Thus, since expressions in OCL do not only compute values, but also contain operations that refer to the pre and post state (σ, σ_{pre}), the semantics of an expression ϕ depends on such state pairs: $I[\![\phi]\!](\sigma, \sigma_{pre})$ [41, Definition A.30]. The standard also introduces $I[\![\phi]\!]\sigma$, i. e., a semantic interpretation on only one state. This version is used for precondition and invariant expressions. For simplification reasons, we *only* use the former version; invariant and precondition expressions are sufficiently characterized syntactically by the requirement that they must not contain any `@pre` in paths. For our purpose of a machine-representable, shallow embedding for OCL, we make a further simplification and omit an explicit semantic function I. This can be done by viewing ϕ as a function from (σ, σ_{pre}) to the semantic value of $I[\![\phi]\!](\sigma, \sigma_{pre}) = \phi(\sigma, \sigma_{pre})$. Technically, this means that we have to *lift over the context* any semantic function occurring in ϕ, e. g., $\_ = \_$ or $\_ + \_$ in the example above, over the states $\tau = (\sigma, \sigma_{pre})$. More precisely, the addition $\_ + \_$ of type $[\texttt{Integer}, \texttt{Integer}] \Rightarrow \texttt{Integer}$ in the sense of the standard must be lifted over the *context* τ, i. e., the corresponding operation $\_ \dot{+} \_$ in HOL-OCL must have the type $(\tau \Rightarrow \texttt{Integer}) \Rightarrow (\tau \Rightarrow \texttt{Integer}) \Rightarrow \tau \Rightarrow \texttt{Integer}$. When introducing the type synonym $V_\tau(\alpha)$ for $\tau \Rightarrow \alpha$, this type transformation can be made explicit: the HOL-OCL type for the OCL type above is $[V_\tau(\texttt{Integer}), V_\tau(\texttt{Integer})] \Rightarrow V_\tau(\texttt{Integer})$. We call τ also the *context* of this type transformation, which is classified as "semi-canonical" translation (see Section 3.4). It turns out that the complete set of OCL operators, be it from the core logic or be it from the quite rich library, can be syntactically constructed as semi-canonical translations. This includes operators with non-trivial binding-structure such as the `let`-construct, or the *iterators* (corresponding to *fold*-like operators known from functional languages).

A further complication of a shallow embedding of OCL is due to the fact, that paths into the state may be undefined, and therefore all functions in expressions may have undefined results. Technically, this means that we have to extend all types by an additional $\perp$ element—which leads to another "semi-canonical" type transformation—, but for the expression semantics, the logic, and the calculi, this has dramatic consequences: Many rules can only be applied for "defined" values, which results in particular case splits and side-conditions in thousands of lemmas and rules. These side-conditions must be established by more or less complex *side-calculi*; for the example of undefinedness, for example, we need rules that infer facts like "if $a + b$ is defined, a and b must be defined." Thus, although considered semantically fairly trivial in many research communities, a proper treatment of definedness in the sense of the standard is usually avoided in deduction systems over OCL-like or VDM-like languages because of the size and complexity of the side-calculi.

Beyond the typical problems concerning undefinedness and states, the most challenging part of HOL-OCL as a shallow embedding consists in a particular *semantic* understanding of the key-notions UML state diagrams. This implies a notion of typed state, covering concepts such such as static and dynamic types in systems, and combining this with the *practical* need of extensibility of the object-type_system and modular theories over it. In the end, the theory also offers a new perspective on methods and late-binding method invocation. However, a modular treatment of state diagrams requires a kind of "semantic compiler" converting it into a semantic theory over the types, their relation, their properties with respect to object construction and destruction, etc.

3.4. An Overview over Embedding Techniques

To put HOL-OCL as an embedding into perspective, we define several notions for their classification. As already mentioned, we call a syntax translation:

- *canonical*, if and only if the non-terminals (or types) of the object language can be mapped one-to-one to type constructors in the HOL representation,
- *semi-canonical*, if and only if the non-terminals (or types) of the object language can be mapped to type expressions (possibly containing free type variables),
- *pre-compiled*, if and only if the non-terminals (or types) of the object language were mapped to type expressions according to their context in the term-language.

As an example for a canonical logical embedding, consider HOL-CSP [49]. Here, the syntactical category of a "process over an alphabet Σ" can be directly represented by a type_constructor $(\alpha)\,\texttt{process}$, which is represented by the semantic domains of CSP (traces, failures, divergences). Binding of summation operators can be represented directly by *higher-order abstract syntax* (HOAS), as well as the fixedpoint operator. Derived calculi allow for equivalence proofs of processes (even over infinite alphabets) via fixpoint induction.

As an example for a pre-compiled translation, consider the HOL-Z embedding [11]. While most operators from Z's "mathematical toolkit" can be compiled canonically, the expressions for the schema calculus must be pre-compiled. In particular, a schema expression A may be mapped to a function $\alpha_1, \ldots, \alpha_n \Rightarrow \text{bool}$, where both the α_i and the n depend on previous declarations.

We turn now to a classification of type systems resulting from an embedding function $\texttt{embedding} : L \Rightarrow \text{HOL}$ mapping expressions of an object-language L to expressions in HOL. We assume the existence of predicates $\texttt{welltyped}_L$ (assuring that an expression of language L is welltyped with respect to type discipline) and $\texttt{welltyped}_{\text{HOL}}$ (assuring that a HOL-expression is welltyped with respect to the simple-type type-system including parametric polymorphism called λ^α.

Then we can characterize a type translation underlying an embedding as

- *approximate*, if and only if for any e, $\texttt{welltyped}_{\text{HOL}}(\texttt{embedding}\ e)$ is implied by $\texttt{welltyped}_L(e)$,
- *tight*, if and only if the type translation is approximative and we have additionally for any e, $\texttt{welltyped}_L(e)$ is implied by $\texttt{welltyped}_{\text{HOL}}(\texttt{embedding}\ e)$.

Thus, if an embedding is tight, a possible implementation for a type-checker consists in applying the `embedding`-function and trying to type-check its result in HOL; an error on the converted term indicates the existence of a type error on the original term. Note, however, that the practical usability of such a conceptual implementation is usually very limited, in particular for semi-canonical or pre-compiled languages, since the error messages gained by this process are hard to decipher for users not familiar with the semantic details of the embedding.

	Process-Oriented	Data-Oriented	Object-Oriented
Example	HOL-CSP	HOL-Z	HOL-OCL
Syntax	canonical Translation	pre-compiled HOAS	OCL: semi-canonical, UML: pre-compiled
Typing	tight	approximate	tight
Semantics	denotational semantics, complex side-calculi	fairly trivial	complex side-calculi, three-valued logics, theory morphisms

Table 3.1.: Comparing Logical Embeddings

On this basis, we can compare logical embeddings like HOL-CSP [49], HOL-Z [11] and the present HOL-OCL in Tab. 3.1. The reason for the non-tightness of the Z embedding lies in a mere technicality: the Z type-system supports n-ary tuples or records. They are distinct to their (isomorphic) representation as nested pairs: (a, b, c) has not the same type as $(a, (b, c))$. Since the latter is the representation of the former in HOL-Z, an expression like $(a, b, c) = (a, (b, c))$ can be typed after applying the embedding function although it cannot be typed with the Z type discipline.

Some explanation for the remarks in the line summarizing semantic features: in a shallow embedding, it is necessary to express syntactic constraints (e. g., for admissibility as in the case of fixpoint-induction, or continuity in the case of process-refinement, or state-passing in the case of rewriting in OCL) *semantically*, i. e., by second or third-order predicates expressing conditions that were otherwise characterized by inductively defined subsets of the syntax. These semantic predicates result in side-conditions which can be established by derived rules having a similar form and purpose than the inductive rules of a sub-language definition (for example, the admissibility rules characterize formulae built over logical constants, $\_ \wedge \_$, $\_ \vee \_$ and universal quantification; thus, existential quantification or negation are ruled out except that they occur in the construction of constant terms). However, their semantic construction leads sometimes to unexpected generalizations and often improves the insight into their logical nature than their (traditional) syntactic counterparts.

Chapter 4.

Faithfully Representing UML/OCL

In the first two sections of this core chapter on our formalization of UML and OCL semantics, we refine our overall goal for a faithful formal semantics with respect to the standard into several sub-goals and then present an overall architecture of HOL-OCL meeting these goals. The subsequent sections are concerned with various aspects of this architecture.

4.1. A Note About Standard Compliance

We claim that we provide a semantic representation compliant with the OCL standard semantics definition. In this section, we make our claim more precise, in particular we have to discuss to which parts we claim to be compliant. First, the OCL standard is divided into *normative* parts and *informative*, i. e., not normative, parts. The semantics[1] of the standard appears in the following chapters of [41]:

Chapter 7 "OCL Language Description": This *informative* chapter motivates the use of OCL and introduces it in an informal way, mostly by showing examples. We used this chapter mainly for catching the intentions of the standard in cases where the other parts of the standard are unclear or contradictory.

Chapter 10 "Semantics Described using UML": This *normative* chapter describes the "semantics" of OCL using the UML itself. Merely an underspecified "evaluation" environment is presented. Nevertheless, some of the information presented in this chapter is helpful for formalizing the standard.

Chapter 11 "The OCL Standard Library": This *normative* chapter is, in our opinion, the best source of the normative part of the standard describing the intended semantics of OCL. It describes the semantics of the OCL expressions as requirements (in form of pairs of pre- and postconditions) they must fulfill. Overall, we prove these requirements for our embedding and thus show that our embedding satisfies these requirements.

Appendix A "Semantics": This *informative* appendix defines the syntax and semantics of OCL formally in a textbook style paper-and-pencil notion. It is mostly based on [44].

[1] A nice overview of the different usages of the word "semantics" is given in [22].

Overall, we see the semantic foundations of the standard critical for several reasons:

1. The normative part of the standard does not contain a formal semantics of the language.
2. The consistency and completeness of the formal semantics given in "Appendix A" is not checked formally.
3. There is no proof, neither formal nor informal, that the formal semantics given in the informative "Appendix A" satisfies the requirements given in the normative chapter 10.

Nevertheless, we think the OCL standard [41] ("`ptc/03-10-14`") is mature enough to serve as a basis for a machine-checked semantics and formal tools support.

When we claim to be compliant to the standard, we do not mean that we converted "literally" the "Semantics" chapter of the OCL standard [41, appendix A] into an Isabelle theory. The deviations from the standard can be grouped into the following six classes:

Making the standard more precise: The most important point here is the need for a typed set theory as meta-language for the description of the data-universe and the concepts "model" and "signature" left implicit in the standard: Formalizing this in HOL is neither possible nor desirable, since consistency can no longer be guaranteed. Another issue in this line is that collection types have "smashed semantics" in HOL-OCL (see subsection 4.7.1).

Presentational issues: This covers our decision to turn OCL into a shallow embedding, as well as our decision to use a combinator style presentation for the bulk of semantic definitions both for conceptual and technical reasons. In section 4.5, we show in detail why these formulations are equivalent to the ones used in the standard.

Generalizations: This covers for example our decision to use an infinite collection type $\texttt{Set}_\tau$, since logical connections between, e. g., `->IsType` and *class invariants* can therefore be satisfactorily treated inside an own typed set theory in OCL (see section 4.9)

Glitch

Repairing glitches: The standard contains—as can be expected for a large semi-formal document—several errors in local definitions which were revealed during our formalization. Such cases are marked by with a danger sign on the margin throughout this document, and our definitions can be seen as our proposal for repair.

extension

Suggesting Extensions: In some cases we see our proposals as an extension of the standard, these cases are marked with an exclamation sign on the margin.

Proofs for Compliance Requirements: The OCL standard contains a collection of formal requirements in its mandatory part with no established link to the informative "Appendix A" of the standard [41]. We provide formal proofs for the compliance of our OCL semantics with these requirements (see section 4.6).

Providing alternative mathematical syntax: Being the first who did substantial proof work in OCL, we early noticed the need for a compact, mathematical notation for OCL specifications as alternative to the programming-language like notation used in the OCL 2.0 standard. However, we accept the latter one in our front-ends, and as an alternative input format in the proof engine, too. This proceeding can be seen as generalization and is also covered by the standard, see Appendix A.

4.2. Building-Blocks of the HOL-OCL Architecture

In section 3.3, we have already seen that HOL-OCL can be decomposed in an UML part concerned with a typed object store and underlying object universes on the one hand and the OCL part just assuming abstract contexts on the other. This decomposition gives a natural basis for a separation of concern.

Before describing architectural issues in more detail, one general remark on terminology is appropriate here: The UML standard makes a clear separation between *operations* and *operation specifications* on the one hand and *methods* on the other; the former are entirely a specification construct, the latter an implementation construct. The UML allows an operation to have several methods, even in different programming languages. Consequently, methods are out of the scope of our work so far. Further, we distinguish operations from operation specifications by viewing the former as mathematical functions, the latter as relations.

With respect to the OCL part, we have to provide:

- A *technique* for giving semantics to basic (value) types of OCL, i. e., `Boolean`, `Integer`, `Set` and collection types such as `Set`, or `Sequence`.
- A *technique* for giving semantics for built-in operations over contexts, capturing arithmetics, logic and collection type theories.
- A *technique* for giving semantics for user-defined operations.

With respect to the UML part (resulting in a semantics for path expressions), we have to provide:

- A mechanism to generate formal theories of typed *object structures* associated to classes and their relationships (e. g., inheritance).
- A technique for giving semantics for user-defined operations in the context of classes, leading also to a formal semantics of path expressions. In principle, this constitutes an embedding of a subset of the UML foundation package [40].
- And last but not least, we have to bring both embeddings together, i. e., generating semantics for invariants and operation specifications consisting of pre and post conditions.

Further, we aim for mechanisms providing modularization and extensibility:

- The object store should allow for modular proofs, i. e., one should be able to add new classes without the need of re-proving the properties of existing classes.
- The embedding of OCL should be easily useable with another kind of object store, e. g., one idea is to use OCL for a Java-like object store allowing, e. g., a distinction between $\perp$ and the "null" reference (pointer).
- The object-store should be useable without OCL, e. g., for a programming language description that allows for method definitions associated to an operation; thus, one could verify a method with respect to its operation specification in a Hoare-logic style of reasoning.

In this chapter we present techniques and concepts; the concrete collection of theories implementing them is contained in Appendix B.

Each of the mentioned techniques and encoding mechanisms can be organized into *level*, which were built in their core by formally defined theory morphisms called *layer*, in particular:

Level 0: This level defines the ground work for the embeddings. It consist out of two layers:

Defining new Datatypes: In this layer, we define HOL types, in particular auxiliary types for classes.

Datatype Adaption: In this layer, the HOL datatypes are adapted as needed, e. g., we glue basic datatypes together to objects or extend all datatypes by an special "undefined" element.

In summary, this level defines all datatypes and provides a extensible object store.

Level 1: This adds, respectively adapts, the functional behavior and finalizes the embeddings. it consists out of two layers:

Functional Adaption: This layer adapts and extend the functional behavior of our embeddings, e. g., it defines the strictness of operations and defines the semantics of operations invocations in the context of our object store.

Embedding Adaption: This layer adds infrastructure for handling contexts.

In summary, this level provides an embedding of core OCL (i. e., there are only OCL formulae without context declarations) and an embedding of an extensible object store with operation invocation.

Level 2: This level combines the two embeddings, i. e., it introduces the context of OCL formulae and defines the semantics of objects and method invocations with respect to the validity of the corresponding preconditions, postconditions and invariants.

An overview of this architecture is shown in Figure 4.1 and will describe both, levels and layers, in more detail in the following, in particular in section 4.7.

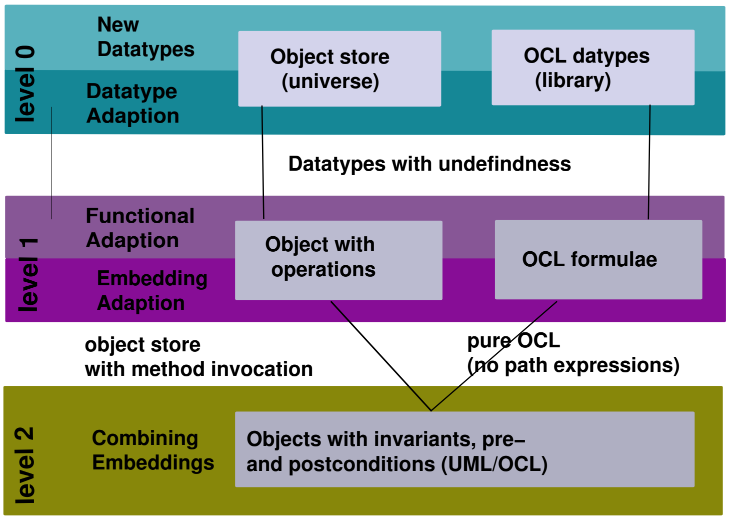


Figure 4.1.: An High-Level Overview of the HOL-OCL Embedding Architecture

4.3. Formal Preliminaries for Representing Semantics in HOL

First of all, we use the usual notation for Cartesian product $\alpha \times \beta$ and sum types $\alpha + \beta$. Recall that polymorphic type variables were represented by Greek letters. The theory of these type constructors is developed in the HOL library and provides the constructor (x, y) for pairs and the projections fst and snd. The constructors for sum are Inl and Inr; the projection is hidden in the "pattern match function"

$$\mathrm{sumCase}(f, g, x) = \begin{cases} f(k) & \text{if } x = \mathrm{Inl}\, k\,; \\ g(k) & \text{if } x = \mathrm{Inr}\, k\ . \end{cases}$$

In OCL, the notion of explicit undefinedness plays a fundamental role, both for the logical and non-logical expressions: c

> Some expressions will, when evaluated, have an undefined value. For instance, typecasting with `oclAsType()` to a type that the object does not support or getting the `->first()` element of an empty collection will result in undefined.
> *(OCL Specification [41], page 15)*

Thus, concepts like *definedness* and *strictness* play a major role in the OCL. We used Isabelle's concept of a *type class* to specify the class of all types bot that contain the undefinedness element $\perp$. Additionally, we required from this class the postulate "all OCL types must have one element different from the undefined value" to rule out certain pathological cases revealed during the proofs. For all types in this class, concepts such as *definedness*

$$\mathrm{def}(x :: \alpha :: \mathrm{bot}) \equiv (x \neq \bot)$$

or *strictness of a function*

$$\mathrm{isStrict}(f) \equiv (f \bot) = \bot$$

are introduced. We define a combinator strictify by

$$\mathrm{strictify}\, f(x) \equiv \mathrm{if}\; x = \bot\; \mathrm{then}\; \bot\; \mathrm{else}\; f(x) \quad \text{with type } (\alpha :: \mathrm{bot} \Rightarrow \beta :: \mathrm{bot}) \Rightarrow \alpha \Rightarrow \beta.$$

The operator strictify yields a strict version of an arbitrary function f defined over the type class bot.

Further, we use the type constructor τ up that assigns to each type τ a type *lifted* by $\bot$, for which we also write $\tau_\bot$. Since any type in HOL contains at least one element, each type $\tau_\bot$ is in fact in the type class bot. The function $\llcorner\_\lrcorner : \alpha \to \alpha_\bot$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\bot \to \alpha$ its inverse for defined values. The case distinction function is defined by:

$$\mathrm{upCase}(f, c, x) = \begin{cases} c & \text{if } x = \bot\,; \\ f(k) & \text{if } x = \llcorner k \lrcorner\,. \end{cases}$$

We will complete our formalization section of the machinery specific to our shallow embedding by the type synonym $V_\tau(\alpha)$ already mentioned in Section 3.3 and some combinators that capture the semantical essence of context lifting. The type synonym $V_\tau(\alpha)$ is defined by:

$$V_\tau(\alpha) = \tau \Rightarrow \alpha\,.$$

On the expression level, context lifting combinators fining the distribution of contexts are defined as follows:

$$\begin{array}{ll} \mathrm{lift}_0\, f \equiv \lambda\, \tau.\; f & \text{with type } \alpha \Rightarrow V_\tau(\alpha)\,, \\ \mathrm{lift}_1\, f \equiv \lambda\, X\; \tau.\; f(X\; \tau) & \text{with type } (\alpha \Rightarrow \beta) \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)\,, \text{ and} \\ \mathrm{lift}_2\, f \equiv \lambda\, X\; Y\; \tau.\; f(X\; \tau)(Y\; \tau) & \text{with type } ([\alpha, \beta] \Rightarrow \gamma) \Rightarrow [V_\tau(\alpha), V_\tau(\beta)] \Rightarrow V_\tau(\gamma)\,. \end{array}$$

The types of these combinators reflect their purpose: they "lift" operations from HOL to semantic functions that are operations on contexts.

Operations constructed by context lifting enjoy the property, that the context is just passed, not changed. We call an operation *context passing* if it satisfies exactly this property. This is expressed formally as follows:

$$\mathrm{cp}(P) \equiv \big(\exists\, f.\, \forall\, X\tau.\; P\; X\; St = f\; (X\; \tau)\; \tau\big) \quad \text{with type } \big(V_\tau(\alpha) \Rightarrow V_\tau(\beta)\big) \Rightarrow \mathrm{bool}\,.$$

Context invariance of expressions will turn out to be a key concept allowing for converting an equivalence on OCL expressions into a congruence; thus cp will play a major role in side-calculi for OCL.

The fundamental theory for undefinedness, elementary strictness combinators, lift-combinators and cp is developed in subsection B.3.1.

4.4. Encoding Standard OCL Operations

The standard contains "principles" for the semantics of the operations. Consider the example:

> In general, an expression where one of the parts is undefined will itself be undefined. *(OCL Specification [41], page 15)*

In other words, one could rephrase this semantic principle as "all operations are strict," be it standard or user-defined operations. The OCL standard requires as default for all operations to be *strict*, both for the case of built-in like the $\_ + \_$ on $\texttt{Integer}_\tau$ or user defined operations declared in class diagrams. Other "principles" are hidden in the semantic definitions; for example the passing of the context. Since we have to define around hundred operators, it is tempting to cover these underlying principles in combinators once and forall, both for conceptual and technical reasons (as we will see in section 4.7 in more detail). For the combinators introduced in section 4.3, it is now straight forward to meet the semantic principles. For the new types $\texttt{Boolean}_\tau$ (subsection B.4.2 "OCL Boolean") and $\texttt{Integer}_\tau$ (subsection B.4.6 "OCL Integer"), for example, a representation as types from the HOL library looks as follows:

$$\begin{aligned} \texttt{Boolean} &= \text{bool}_\perp\,, \\ \texttt{Integer} &= \text{int}_\perp\,, \\ \texttt{Boolean}_\tau &= V_\tau(\texttt{Boolean})\,, \\ \texttt{Integer}_\tau &= V_\tau(\texttt{Integer})\,. \end{aligned}$$

These definitions are contained in theories subsection B.4.14. These type definitions by type synonyms are typical for our semi-canonical type translation.

Basic constant definitions for $\mathsf{T}, \mathsf{F}, \perp$ in $\texttt{Boolean}_\tau$ or $0, 1$, etc., in $\texttt{Integer}_\tau$ using the lifting combinators of section 4.3 is straight-forward:

$$\begin{aligned} \perp &\equiv \text{lift}_0(\llcorner\perp\lrcorner) && \text{with type } \texttt{Boolean}_\tau\,, \\ \mathsf{T} &\equiv \text{lift}_0(\llcorner\text{true}\lrcorner) && \text{with type } \texttt{Boolean}_\tau\,, \\ \mathsf{F} &\equiv \text{lift}_0(\llcorner\text{false}\lrcorner) && \text{with type } \texttt{Boolean}_\tau\,, \\ \perp &\equiv \text{lift}_0(\llcorner\perp\lrcorner) && \text{with type } \texttt{Integer}_\tau\,, \\ 0 &\equiv \text{lift}_0(\llcorner 0\lrcorner) && \text{with type } \texttt{Integer}_\tau\,, \text{ and} \\ 1 &\equiv \text{lift}_0(\llcorner 1\lrcorner) && \text{with type } \texttt{Integer}_\tau\,. \end{aligned}$$

In fact, the definition for undefinedness is done for the *polymorphic constant* $\perp$, expressing the fact that undefinedness is omnipresent in all types of the OCL language. Furthermore, we use within the theories the mathematical syntax; a translation table can be found in Appendix A.

An example for a strict boolean operation is the logical negation:

$$\neg\,\_ \equiv \mathrm{lift}_1(\mathrm{strictify}(\llcorner\_\lrcorner \circ (\neg\_) \circ \ulcorner\_\urcorner) \qquad \text{with type } \texttt{Boolean}_\tau \Rightarrow \texttt{Boolean}_\tau\,.$$

From this definition, the usual logical laws for a strict negation are derived:

$$\neg\bot = \bot \qquad\qquad \neg\mathsf{T} = \mathsf{F} \qquad\qquad \neg\mathsf{F} = \mathsf{T}$$

A typical example for strict binary operation is the addition on integers; all other unary and binary operators of the theory of the follow this scheme of a constant definition:

$$\_ + \_ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \llcorner \ulcorner x\urcorner + \ulcorner y\urcorner \lrcorner)))\,.$$

From these definitions, computational rules on numbers can be derived, which perform computations like $3 + 4$ on the basis of binary representations; thus, computations of this kind can be handled fairly efficiently by the Isabelle rewriter. This representation technique for numbers is by no means new, it is merely a standard in Isabelle and adapted for these new types.

The encoding of the operations along the scheme described above is the default encoding; cases requiring special treatment are non-strict operators as well of higher-order constructs like quantifiers and the iterators of the OCL language similar to `fold` in functional programs:

> These iterator expressions always have a collection expression as their source, as is defined in the well-formedness rules . . . . *(OCL Specification [41], page 149)*

Non-strict constructs such as $\partial\, \mathit{self}$ or $\emptyset\, \mathit{self}$ were defined via context lifting from the definedness predicate def introduced in section 4.3, without using the strictness combinator.

4.5. Textbook vs. Combinator Style Semantics of Operations

As mentioned previously, we use a combinator-style presentation rather than a textbook-style presentation as used in the OCL standard, both for reasons of conciseness as well as better amenability techniques. It is formally shown in our Isabelle theories that from our definitions, the requirements [41, Chapter 11 (normative)]of the standard follow. In this section, we give an alternative reason for this: namely, we formally show that our semantics is equivalent to (a formalized version of) the semantics given standard [41, appendix A (non normative)].

The OCL 2.0 standard presents a definition scheme for all *strict basic operations* just by one example. For the +-operator on integers, it looks as follows: OCL [41, page

A-11] this definition is presented as:

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot\,, \\ \bot & \text{otherwise.} \end{cases}$$

This semantic function for basic operations is integrated in the more general semantic interpretation function for OCL expressions in

> Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \mathrm{Expr}_t$ is a function $I\llbracket e \rrbracket : \mathrm{Env} \to I(t)$ that is defined as follows.
> iv. $I\llbracket w(e_1, \dots e_n \rrbracket \tau = I(w)(\tau)\big(I\llbracket e_1 \rrbracket(\tau), \dots, I\llbracket e_n \rrbracket(\tau)\big)$
>
> *(OCL Specification [41], page A-26, definition A.30)*

There are two more semantic interpretation functions; one concerned with path expressions (i. e., *attribute and navigation expressions* [41, Definitions A.21], and one concerning the interpretation of pre and postconditions $\tau \vDash P$ which is used in two different variants.

To show the equivalence of the two formalization styles, we re-introduce a kind of "explicit semantic function" into our shallow embedding. Of course, which with respect to HOL-semantics, this is just the identity. $\mathrm{Sem}\llbracket E \rrbracket \tau$ can be thought of as the *fusion* of the two semantic functions $I(o)$ and $I\llbracket E \rrbracket$:

$$\mathrm{Sem}\llbracket x \rrbracket \equiv x \qquad\qquad \text{with type } \alpha \Rightarrow \alpha\,.$$

The definitions and exemplary proofs shown here are contained in theory subsection B.4.2.

Now we show for our first strict operation in OCL, the not operator, that it is in fact an instance of the standards definition scheme:

$$\mathrm{Sem}\llbracket \neg\ X \rrbracket \gamma = \begin{cases} \llcorner \neg \ulcorner \mathrm{Sem}\llbracket X \rrbracket \gamma \urcorner \lrcorner & \text{if } \mathrm{Sem}\llbracket X \rrbracket \gamma \neq \bot\,, \\ \bot & \text{otherwise}\,. \end{cases}$$

This is formally proven as lemma "not_faithfully_represented" in the theory subsection B.4.2. The proof is trivial and canonical: it consists of the unfolding of all combinator definitions (they are just abbreviations of re-occurring patterns in the textbook style definitions!) and the semantic function Sem which is merely a syntactic marker in our context.

For the binary example of the integer addition, one proceeds analogously and receives as result:

$$\mathrm{Sem}\llbracket X + Y \rrbracket \gamma = \begin{cases} \llcorner \ulcorner \mathrm{Sem}\llbracket X \rrbracket \gamma \urcorner + \ulcorner \mathrm{Sem}\llbracket Y \rrbracket \gamma \urcorner \lrcorner & \text{if } \mathrm{Sem}\llbracket X \rrbracket \gamma \neq \bot \text{ and } \mathrm{Sem}\llbracket Y \rrbracket \gamma \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases} \tag{4.1}$$

This is formally proven in subsection B.4.2.

In the following, we summarize the differences between the OCL standards textbook definitions and our combinator-style approach:

1. The standard [41, chapter A] assumes an "untyped set of values and objects" as semantic universe of discourse. Since we reuse the types from the HOL-library to give boolean, integers and reals a semantics, meta-expressions like $\{\text{true}, \text{false}\} \cup \{\bot\}$ used in the standard are simply illegal in our interpretation. This makes the injections $\lfloor\_\rfloor$ and projections $\ulcorner\_\urcorner$ necessary.

2. The semantic functions in the standard are split into $I(x)$, $I[\![e]\!]\tau$, $I_{\text{ATT}}[\![e]\!]\tau$ and $\tau \vDash P$. Since we aim at a shallow embedding (which ultimately suppresses the semantic interpretation function), we prefer to fuse all these semantic functions into one.

3. The *environment* τ in the sense of the standard is a pair of a variable map and a state pair. The variable map is superfluous in a shallow embedding (binding is treated by HOL itself), our contexts τ just comprises the pair of pre and post state, thus an implementation of our notion of context.

4.6. Compliance to the Standards OCL Requirements

As already described, the semantics of OCL is spread over several chapters in the OCL standard. For example, For example, with respect to undefinedness, it is stated that:

> In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule,however. First, there are the logical operators:
>
> - True OR-ed with anything is True
> - False AND-ed with anything is False
> - False IMPLIES anything is True
> - anything IMPLIES True is True
>
> The rules for OR and AND are valid irrespective of the order of arguments and they are valid whether the value of the other sub-expression is known or not.
>
> *(OCL Specification [41], page 15)*

---

which implies explicitly that OCL is based on a strong Kleene_Logic. Thus, most operators of the logical type like _ `and` _ (written $\_ \wedge \_$) are explicitly stated exceptions from the "all operations are strict" principle.

In the normative part [41, chapter 11], requirements were formally stated on the standard operations of OCL; the question, if these requirements are met by the informative semantics description[appendix A] [41] has not been systematically investigated. A

Although, for most of the logical connective this can be settled by rephrasing the requirement given in form of a truth table [41, page A-12, Table A.2], there is a contradiction with respect to the truth table for `implies` and the requirement given in the normative part of the standard [41, page 139].

The formal semantics of `implies` *does not satisfy its requirements.*

It is a contribution of our work that we can in fact formally prove the requirements are met by our semantics. In the case of the logical connectives, compliance to the standard is proven by deriving lemmas representing the complete truth table as required in the standard. Further, we also prove the normative requirements, and thus connect the informative formal semantics with the normative requirements of the standard. Moreover, lemmas were derived for special calculi allowing for automated reasoning in OCL (see chapter 5).

With respect to the requirements of the standard [41, chapter 11], e. g., for `isEmpty`:

---

isEmpty():Boolean
Is *self* the empty collection?
`post: result = (self->size() = 0)` *(OCL Specification [41], page 141)*

---

we prove formally requirement satisfaction lemmas:

$$\frac{\vDash \partial\big(\mathit{self}\texttt{->size()}\big)}{\mathit{self}\texttt{->isEmpty()} = \big(\mathit{self} :: (\alpha\ \texttt{Set}_\tau)\big)\texttt{->size()} \doteq 0}\,. \tag{4.1}$$

Instead of using operation specifications, we prefer their reformulation as algebraic properties that are directly usable in proofs. The constraint $\vDash \partial\big(\mathit{self}\texttt{->size()}\big)$ (the size of the set must be defined) is a tribute to our extension of the standard to infinite sets; it has the effect to constrain this specification to finite sets, i. e., to the domain the requirement is intended to hold.

4.7. Organizing the Embedding into Layer

In the previous sections we developed a standard representation technique for OCL operations. In this section, we will revise and generalize this technique to meet a particular technical problem of our approach.

The conservative embedding approach for yielding semantics for a large language such as OCL must provide derivations for several thousand "folklore" theorems to be practically relevant. In this section, we present an approach for deriving the mass of these theorems mechanically from the existing HOL library. The approach assumes a structured *theory morphism* mapping library types and library functions to new types and new functions of the specification language (i. e., OCL) while uniformly modifying some semantic properties. It turns out that the technique also greatly facilitates the technical organization of the automated generation of theories from class diagrams (see section 4.9)

The key idea is to represent the structure of the theory morphism by the *semantic combinators* $\_{}_\bot$, $V\_(\_)$,strictify,$\llcorner\_\lrcorner$, $\ulcorner\_\urcorner$, lift_0, lift_1, lift_2, lift_3 introduced in the previous sections. We say that a theory morphism is layered, iff in each form of conservative extension the following decomposition into elementary theory morphisms (the *layer*) is possible:

1. for type synonyms $(\alpha_1, \dots, \alpha_m)T$, there must be type constructors C_1 to C_n such that
$$(\alpha_1, \dots, \alpha_m)T = C_n\Big(\cdots\big(C_1(T')\big)\Big)$$

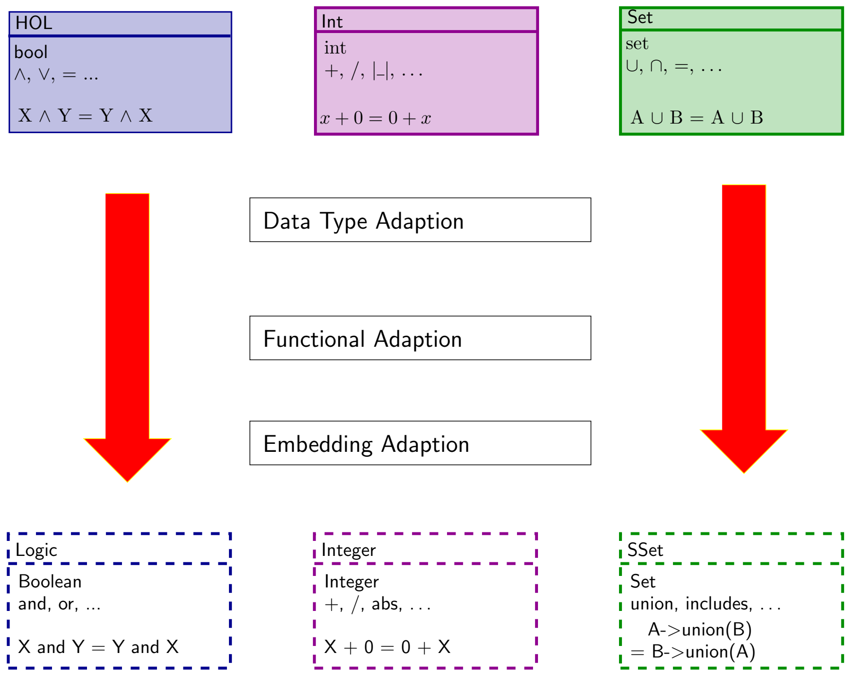


Figure 4.2.: Derivation of the OCL-library

2. for conservative type definitions $(\alpha_1, \dots, \alpha_m)T$, there must be functions C_1 to C_n such that

$$(\alpha_1, \dots, \alpha_m)T = \Big\{x :: C_n\Big(\cdots(C_1(T'))\Big) \,\Big|\, P(x)\Big\}$$

3. for constant definitions c, there must be functions E_1 to E_n such that:

$$c = (E_n \circ \cdots \circ E_1)(c')$$

where each C_i, or E_i are (type constructor) expressions build from *semantic combinators* of *layer* S_i and T'. Where c' is a construct from the meta logic. A *layer* S_i is represented by a specification defining the *semantic combinators*, i. e., constructs that perform the semantic transformation from meta-level definitions to object-level definitions. In Figure 4.2, we show a classification for such layers.

In the following sections, we will present a collection of layers and their combinators in more detail. We will associate the semantic combinators one by one to the specific layers and collect them in a distinguished variable *SemCom*. Finally, we will put them together for our example OCL and describe generic theorem proving techniques that exploit the layering of the theory morphism for OCL .

4.7.1. Datatype Adaption

Datatype adaption establishes the link between meta-level types and object-level types and meta-level constants to object-level constants. While meta-level definitions in libraries of existing theorem prover systems are geared toward good tool support, object-level definitions tend to be geared to a particular computational model, such that the gap between these two has to be bridged. For example, in Isabelle/HOL, the *head*-function applied to an empty list is defined to yield an arbitrary but fixed element; in a typical executable object-language such as SML, Haskell or OCL, however, this function should be defined to yield an exception element that is treated particularly. Thus, datatype adaption copes with such failure elements, the introduction of boundaries (as maximal and minimal numbers in machine arithmetics), congruences on raw data (such as *smashing*; see below) and the introduction of additional semantic structure on a type such as $\alpha :: \mathrm{bot}$, or similarly, complete partial orders [36].

Now, revising the definition (cf. section 4.4):

$$\texttt{Boolean} = \mathrm{bool}_{\bot}\ , \text{and}$$
$$\texttt{Boolean}_{\tau} = V_{\tau}(\texttt{Boolean})\,,$$

we easily recognize the layer structure of these definitions.

We turn now to the semantical function combinators of this layer. We identify them in the injection $\llcorner\_\lrcorner$ and the projection $\ulcorner\_\urcorner$. We mark them as combinators by including them in the specific set: $\_{}_{\bot}$, $V\_(\_)$,strictify,$\llcorner\_\lrcorner$, $\ulcorner\_\urcorner$, lift_0, lift_1, lift_2, $\mathrm{lift}_3 \in \mathit{SemCom}$.

As an other example for a congruence construction, we will show the *smashing on sets*, which occurs in the semantics of SML or OCL, for example. In a language with semantic domains providing $\bot$-elements, the question arises how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can have:

$$(\bot, X) \neq \bot \qquad\qquad \{a, \bot, b\} \neq \bot \qquad\qquad \ldots$$

or:

$$(\bot, X) = \bot \qquad\qquad \{a, \bot, b\} = \bot \qquad\qquad \ldots$$

The latter variant is called *smashed product* and *smashed set*. The semantics chapters make no clear decision here; since OCL tends to define its constructs towards executablility and proximity of object-oriented programming languages such as Java, we opt for smashed collection semantics. The constant definition for the semantic combinator smash reads as follows:

Undefinedness in Collections

$$\mathrm{smash}\, f\ \ X \equiv \mathrm{if}\ f\ \bot\ X\ \mathrm{then}\ \bot\ \mathrm{else}\ X \quad \text{with type } \big[[\beta :: \mathrm{bot}, \alpha :: \mathrm{bot}] \Rightarrow \mathrm{bool}, \alpha\big] \Rightarrow \alpha\ \ .$$

On this basis, the collection type `Set`, for example, is built via the type definition[2]

$$\alpha\ \texttt{Set} = \Big\{X :: (\alpha :: \mathrm{bot}\ \ \mathrm{set}_{\bot}) \Big| \bot \notin \ulcorner X \urcorner\Big\}$$

[2]In fact, in the theories, the equivalent formulation as semantic combinators ($\mathrm{smash}(\lambda\, xX.\ x \in \ulcorner X \urcorner)X) = X$ is used instead of $\bot \notin \ulcorner X \urcorner$

and the type synonym:

$$\alpha\ \texttt{Set}_\tau = V_\tau(\alpha\ \texttt{Set})\,.$$

As a consequence of this definitions, sets in HOL-OCL may be infinite which allows representing the syntactic category of *types* as sets in HOL-OCL.

Infinite Sets

This quotient construction identifies all sets containing $\perp$ in one class which is defined to be the $\perp$ of the type α set. All other sets were injected into an own class. Thus, an embedding of smashed sets into the class bot can be done as via the (overloaded) constant definition:

$$\perp \equiv \mathrm{AbsSet}\perp$$

The injection AbsSet (together with the projection RepSet) is a consequence of the conservative type definition above (cf. section 2.1).

For nested OCL types such as `Set(Set(Integer))`, the HOL type is $\texttt{Integer Set Set}_\tau$ and not $\texttt{Integer}_\tau\ \texttt{Set}_\tau\ \texttt{Set}_\tau$ since context lifting is only necessary on the topmost level for each argument of an operation.

We "mark" the semantic combinators smash, $\perp :: \alpha\ \texttt{Set}$, AbsSet, RepSet by including them into *SemCom*.

4.7.2. Functional Adaption

Functional adaption is concerned with the semantic transformation of a meta-level function into an object-level operation. Functional adaption may involve, for example, the

- *strictification* of an operation, i. e., its result is undefined if one of its arguments is undefined,
- *late-binding invocation* semantics for operations. This semantic conversion process is necessary for converting a function into an operation using dynamic overloading.

Technically, this is achieved by the strictify combinator already introduced in section 4.3. Overloading and late-binding can be introduced by the combinators invoke and invokeS described in this section.

As an example for strictification, we present a definition of a built-in operation over a non-trivial datatype adaption, OCL's union on sets. An intermediate version can now be defined as strictified version of HOL's union over the smashed type α `Set`:

$$\mathrm{union} \equiv \mathrm{strictify}\big(\lambda X.\ \mathrm{strictify}(\lambda Y.\ \mathrm{AbsSet}_\llcorner\ulcorner\mathrm{RepSet}\,X\urcorner \cup \ulcorner\mathrm{RepSet}\,Y\urcorner{}_\lrcorner)\big)$$

with type $[\alpha :: \mathrm{bot}\ \texttt{Set}, \alpha\ \texttt{Set}] \Rightarrow \alpha\ \texttt{Set}$.

The, from object-oriented programming languages, well-known concept of method-overloading is not yet fully supported by OCL. We believe, this is more or less due to some accidental circumstances:

1. The UML standard [40, chapter 4.4.1] requires that operation names are unique within the same namespace. Albeit, the UML standard allows one to (explicitly) overwrite methods, i. e., implementation of operations.

2. The OCL standard [41, chapter 7.3.41] restricts the use of the precondition and postcondition declarations to operations or other behavioral features. Sadly, all OCL tools we know of do not support the specification of preconditions and postconditions for methods.

3. Whereas the OCL standard speaks on several places from operation calls, it does not give an hints how operation overloading should be solved, neither does it explain in details concepts like operation (method) calls or operation (method) invocations.

Underspecification of Overload-Resolution

Bringing these together, one has to conclude, that operation overloading, and thus late-binding, is underspecified, or even not supported in OCL. Nevertheless, we think that overwriting inherited operations or methods is a very important feature of object-orientation and thus should be supported by the OCL. Thus we already provide the theoretical foundations for supporting late-binding (and thus overloading of operations) within HOL-OCL, nevertheless a concrete syntax for specifying this has to be worked out.[3]

As many object-oriented languages provide a particular call-scheme for functions, called *method invocation* which increases the reusabilty of code. Moreover, many object-oriented programming languages such as Java assume a particular overload-resolution strategy for invocations; this well-known construction in programming language theory is called *late-binding*. Since we believe that future versions will overcome the self-restriction to operation calls, and since the UML definition expresses at several places a clear preference for overloading operations, we will discuss in this document and partially implement in HOL-OCL a late-binding semantics of method invocation .

late-binding invocation

The treatment of late-binding requires a particular pre-compilation step concerning the declaration of overloaded-methods discussed in section 4.13 in more detail; in this section, we will concentrate on the callee-aspect of method invocations, i. e., how to represent sub-expressions occuring in post conditions representing an invocation of a user-defined operation_specification:

```
context A::m(a1:t1,...,an:tn):t
  pre:   ...
  post:  ...
```

In an "invocation," e. g., a sub-expression $a.m(a_1, \ldots, a_n)$, the semantic value of the "dynamic type" (see section 4.9 for more details) of *self* is detected, i. e., a set of "objects" whose structure is to be discussed later. This "type" helps to lookup the concrete operation specification in the table. This specification can be turned into a function (just by picking *some* function satisfying the specification) which is applied

[3] As simple workarounds, one ignore for operations the well-formedness constraint of UML that requires operation names to be unique within one namespace, or one could introduce new context declarations allowing one to specify preconditions and postconditions for methods.

to `a` as first argument (together with the other arguments). This semantics is captured in the n-indexed family of analogous `invoke` and `invokeS` semantic combinators; the former define call-by-name semantics, the latter call-by-value semantics satisfying the general strictness principle of OCL. Since invokeS can be realized on top of invoke as usual, we concentrate on the former.

The invoke-combinator is defined for the case $n = 1$, for example, as follows:

$$\mathrm{invoke}\, C\ t\ a\ \mathit{result} \equiv$$

$$\lambda\,\tau.\ \begin{cases} \mathrm{arbitrary} & \text{if } t\ (\mathrm{Least}\ \ X.\ X \in \mathrm{dom}\ \ t \wedge C(a\ \tau) \in X) = \mathrm{None}\,; \\ f\ \ a\ \ \mathit{result}\ \tau & \text{if } t\ (\mathrm{Least}\ \ X.\ X \in \mathrm{dom}\ \ t \wedge C(a\ \tau) \in X) = \mathrm{Some}\, f\,. \end{cases}$$

Here, Least is a HOL operator selecting the least set of a set of sets, that satisfies a certain property. In this case, this property is that *self* (suitably converted by a casting function C, see 4.9 for details) is contained in one of the domains of the lookup table tab_m generated during the processing of the declaration discussed in section 4.13), i. e., some set (of objects) characterizing a type. For such an element of the domain of the lookup table, the specification of the operation is selected and returned.

The conversion C will be instantiated by a suitable "coercion" of a dynamic type to a class type to be discussed later (see section 4.9). Such a conversion is known to the pre-parser from the OCL types.

The process of selecting an arbitrary, but fixed function from a specification (i. e., a relation) is handled by the Choose-combinator omitted here. It is defined essentially as the context-lifting of the HOL Hilbert-Operator $\varepsilon\, x.\ P\ x$ that just gives one result element satisfying P; if this does not exist, the Hilbert-Operator picks an arbitrary element of this type.[4]

Thus, the semantic code for the call-by-value invocation $c.m(a_1, \ldots, a_n)$ is given by:

$$\mathrm{Choose}(\mathrm{invokeS}\ \ \mathrm{C}_{[\mathrm{A}]}\ \ \mathit{tab}_m\ c\ a_1\ \ \ldots\ \ a_n)$$

where c is assumed as an object of class C and to have a subtype of A and $\mathrm{C}_{[\mathrm{A}]}$ is a casting-function that converts C objects to A objects.

4.7.3. Embedding Adaption for Shallow Embedding

Semantic combinators for *embedding adaptions* are related to the embedding technique itself, namely the lifting_over_contexts. Recalling section 3.1, any function o with type $T_1, \ldots, T_n \to T_{n+}$ of the object-language has to be transformed to a function:

$$I[\![o]\!] \qquad\qquad \text{with type } \big[V_\tau(T_1), \ldots, V_\tau(T_n)\big] \Rightarrow V_\tau(T_{n+1})\,.$$

As an example for a binary function like the built-in operation $\cup$ (based on union defined in subsection 4.7.2), we present its constant definition:

$$\cup \equiv \mathrm{lift}_2\,\mathrm{union} \qquad\qquad \text{with type } \big[(\alpha :: \mathrm{bot})\ \mathtt{Set}_\tau, \alpha\ \mathtt{Set}_\tau\big] \Rightarrow \alpha\ \mathtt{Set}_\tau\,.$$

[4]It is a methodological issue to avoid this situation; i.e., a notion of a *consistent OCL specification* is needed which can be reduced to proof obligations. Such a proof methodology is discussed in a future version of this document.

Summing up the intermediate results of the local theory morphisms (i. e., the layers) in the previous subsections, the definition of our running example $\cup$ is given directly by:

$$\cup \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda\, X.\ \mathrm{strictify}(\lambda\, Y.\ \mathrm{AbsSet}_{\llcorner}\ulcorner\mathrm{RepSet}\, X\urcorner \cup \ulcorner\mathrm{RepSet}\, Y\urcorner{}_{\lrcorner}))\Big)$$

One easily recognizes our standard definition scheme, having AbsSet and RepSet as additional semantic combinators. During mechanical lifting of HOL theorems to OCL theorems (such as $A \cup B = B \cup A$), these operators require proofs for the invariance of the underlying quotient constructions; i. e., in this example, it must be proved that the union on representations of OCL sets will again be representations of an OCL set, (i. e., HOL sets not containing $\perp$).

4.8. Extensible Universes in Typed Meta-Language

The standard [41, appendix A] uses naive set theory and an informal notion of "model" (in the sense of mathematical model theory) over a signature given by an informal class model. It implicitly assumes one big universe for values and objects (where undefinedness elements are just there ...) and algebras over it without any concern of existence and consistency.

The standard cannot be formalized in this form in Isabelle, neither in an untyped set theory like Isabelle/ZF or a typed set theory underlying Isabelle/HOL; major efforts to make the foundations precise will have to be invested in both options. But since OCL is a typed language at the end, and since we wanted to have type-issues handled by the Isabelle type-checker and not inside the logic representation, it seemed for us also most natural to use a typed meta-language and typed set theory for this. In particular, in a shallow embedding, there should be a typed representation for each OCL expression, where the OCL type corresponds one-to-one to an HOL type (although not necessarily vice versa). Nevertheless, even our object universe construction should preserve the intentions of the standard, in the sense that general laws reflecting inheritance and subtyping on sets of objects can be derived. If possible, there should be no change of the users perspective of the OCL language, although its foundations have been worked out more precisely.

Recall that the OCL language specifies *states* and relations over them. The states are *object structures*, abstract representations of pieces of memory that were linked via references to each other. Formally, an object structure can be represented by a *state* τ mapping references or *object id's* oid's to *objects*, i. e., tuples of elementary values like integers or strings or object identifiers to other objects in the object structure. The type of these tuples corresponds to the type of the class they are belonging to; the components of the tuple correspond to the attributes of a class. The type of such an object structure is therefore $\mathrm{oid} \Rightarrow \mathscr{U}$.

Instead of constructing a "universe of all objects" (which is either untyped or "too large" for a (simply) typed set theory, where all type sums must be finite), one could think of generating an object universe for each given class diagram. Ignoring subtyping

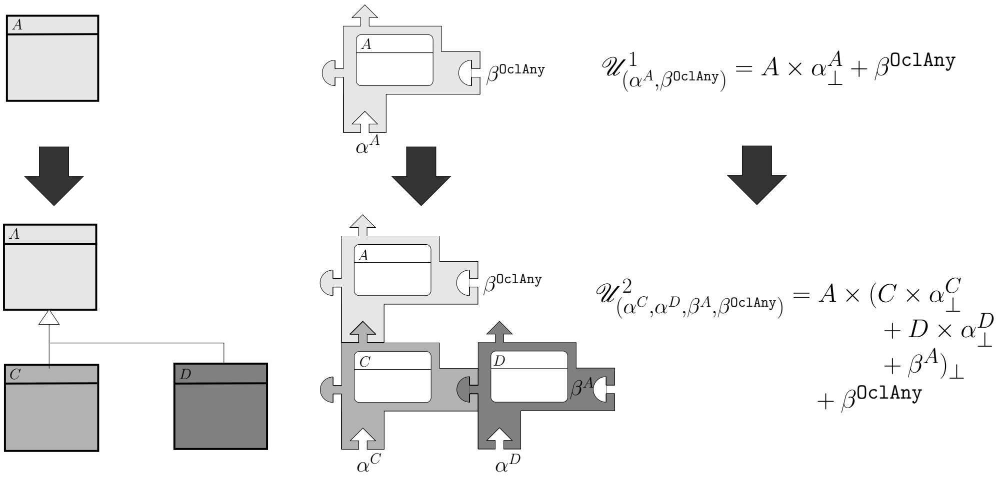


Figure 4.3.: Extending Class Hierarchies and Universes with Holes

and inheritance for a moment, this would result in a universe $\mathscr{U}^0 = A + B + C$ for some class diagram with the classes A, B and C. Unfortunately, such a construction is not extensible: If we add a new class to an existing class diagram, say D, then the "obvious" construction $\mathscr{U}^1 = A+B+C+D$ results in a *different* type to $\mathscr{U}^0$, turning the two object structure types and all values constructed over them into something incomparable. This has quite dramatic consequences: such a representation rules out a modular, incremental construction of larger object systems as used in the UML standard document itself, for example. Properties, that have been proven over $\mathscr{U}^0$ will not hold over $\mathscr{U}^1$; practically, this means that all proof scripts will have to be rerun over an extended universe. Since even library proofs would have to be rerun (in current HOL-OCL, this takes about half an hour ...), such an approach is unfeasible.

Our solution to the problem is to use parametric polymorphisms for *families* for universes $\mathscr{U}^i$ (see Fig. 4.3 for a first overview of this idea), where the families stand for the "possible class diagram extensions". Further, we extend the scheme sketched above by assigning to Classes not directly *objects*, but merely *object extensions*. This "incremental" object view (also used in many implementations) allows for the representation of object inheritance and leads, as we will see, to a smooth integration of object subtyping into the world of parametric polymorphism.

4.9. Encoding Object Structures

4.9.1. The Basics

Recall that object universes are the core of our notion of state, which is the building block of our notion of context τ, which is again the building block of the semantic domain of HOL-OCL expressions: $\tau \Rightarrow \alpha :: \mathrm{bot}$. In this section, we focus on families of object universes $\mathscr{U}^i$, each of which corresponding to a class diagram. Each $\mathscr{U}^i$ comprises all *value types* (`Real`, `Integer`, `String`, `Boolean`, ...) and an extensible *class type representation* induced by a class hierarchy. To each class in a given class diagram, a *class type* is associated which represents the set of *object instances* or *objects*. The *structure* of a $\mathscr{U}^i$ is to provide a family of injections and projections to and from each class type. More precisely, if we assume a class A, this results in:

$$
\begin{array}{ll}
\mathrm{mk}_{\mathrm{A}}^{(0)} & \text{with type } \mathscr{U}^i \Rightarrow A\text{, and} \\
\mathrm{get}_{\mathrm{A}}^{(0)} & \text{with type } A \Rightarrow \mathscr{U}^i
\end{array}
$$

allowing to inject any semantic value of OCL into some $\mathscr{U}^i$. Note, as we need also lifted version of these definitions, we will mark the different versions by different upper indexes. This in turn makes a family of states (containing "object systems") possible:

$$
\begin{array}{ll}
\text{state} & \text{with type oid} \Rightarrow \mathscr{U}
\end{array}
$$

from which concrete values may be accessed via an oid and then be projected via $\mathrm{get}_{\mathrm{A}}^{(0)}$. On this basis, the accessor functions composing OCL path expressions can be built.

The "families" of types and states are represented by parametric polymorphism in our approach. In the big picture, this encodes a subtype type system into the typed λ-calculus with parametric polymorphism underlying HOL in "shallow style" (c.f. [48]).

The extensibility of a universe type is reflected by "holes" (polymorphic variables), that can be filled when "adding" extensions to class objects which means adding subclasses to the class hierarchy. Our construction will ensure that $\mathscr{U}^{i+1}$ (corresponding to a particular class diagram) is just a type instance of $\mathscr{U}^i$ (where $\mathscr{U}^{(i+1)}$ is constructed by adding new classes to $\mathscr{U}^i$). Thus, properties proven over object systems "living" in $\mathscr{U}^i$ remain valid in $\mathscr{U}^{i+1}$.

The universe construction introduced so far follows the interpretation of object models given in the formal semantics of the standard:

> Each object is uniquely determined by its identifier and vice versa. Therefore, the actual representation of an object is not important for our purposes.
>
> *(OCL Specification [41], page paragraph A.1.2.1)*

The standard does not discuss constructors[5] of objects. However, it is interesting to interpret this explicitly stated *bijection* between oids and object instances from the point of view of an operational semantics for them. There are essentially two choices:

1. an object constructor may have *sharing semantics*, i. e., the tuple of attribute values is searched in the global store; if existing, its reference into the store is returned, otherwise the tuple is entered into the state and associated to a fresh object identifier which is returned, or

[5] Although, OCL includes the operation `self.isNew()` which, somehow, evaluates to true if `self` is a "fresh" object.

2. an object constructor may have *creation semantics*, i. e., generate a fresh object identifier and store it together with the tuple of attribute values in the store.

The former seems to be the underlying understanding of the original authors of [41, appendix A]. Even so, the author of the normative part always identifies objects by a reference to it, e. g.:

> If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self.* *(OCL Specification [41], page 15)*

However, the latter view has the advantage to be closer to the usual programming language semantics of object constructors. Furthermore, in this setting, the reference to an object in the store can always be reconstructed which paves the way for reference_types as in Java or C++, and simple semantic definitions for `@pre` or referential equality (as discussed in definition 4.13.

referential vs. non-referential universe construction

Therefore, we provide, among others, one HOL-OCL configuration supporting non-referential universes and one supporting referential universes. Overall, we suggest to resolve this ambiguity in favor of the referential setting, which we also see as the default HOL-OCL configuration.

In our framework, the distinction between sharing semantics and creation semantics is reflected in two alternative universe constructions, namely the *non-referential universe* and the *referential universe*. In more detail, we can choose one of the following two types for `OclAny` and as basis for our universe construction:[6]

$$\alpha\ \texttt{OclAny} = \text{OclAny}_{\text{tag}} \times \alpha_{\bot} \qquad \text{for building a non-referential universe, or}$$

$$\alpha\ \texttt{OclAny} = (\text{OclAny}_{\text{tag}} \times \text{oid}) \times \alpha_{\bot} \qquad \text{for building a referential universe.}$$

Where $\text{OclAny}_{\text{type}}$ is an abstract datatype which makes the type for OclAny unique within our universe construction.

The "initial" universe type $\mathscr{U}^0_\alpha$ is defined as a sum of the built-in value types and the type OclAny (including all its extensions representing subtypes of OclAny):

$$\begin{aligned} \mathit{Values} &= \texttt{Real} + \texttt{Integer} + \texttt{Boolean} + \texttt{String} \\ \mathscr{U}^0_\alpha &= \alpha\ \texttt{OclAny} + \mathit{Values}\,. \end{aligned}$$

Since a class can be extended in several ways, the class hierarchy has a tree-like structure; where the leaves may be "holes." Extending a universe type by instantiating the hole α is done as follows:

$$\alpha \mapsto T \times \alpha_{\bot} + \beta$$

where T is the accumulative type of the attributes of the class extension and $\alpha_{\bot}$ represents the possible (future) extensions of this class. We present this idea in more formal detail in the next section.

[6] Technically, this different HOL-OCL configuration requires another theory for the basic type definition for OclAny (see subsection B.3.8 "Type Definition for OclAny" for details).

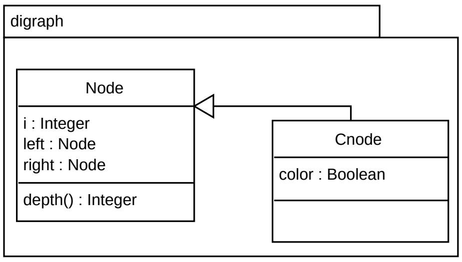


Figure 4.4.: Modelling Directed Graphs: Data Model

```
package digraph

  context Node
  inv range_of_i: self.i > 5

  context Cnode
  inv: (not self.oclAsType(Node).left.oclIsUndefined()
        implies
        (not (self.oclAsType(Node).left.
              oclAsType(Cnode).color = self.color)))
       and
        (not self.oclAsType(Node).right.oclIsUndefined()
         implies
         (not (self.oclAsType(Node).right.
               oclAsType(Cnode).color = self.color)))

endpackage
```

Listing 4.1: Modelling Directed Graphs: OCL Specification

4.9.2. The Formal Details of Encoding Object Structures

We will present the framework of our object encoding together with a small example: assume a class `Node` with an attribute `i` of type integer and two attributes `left` and `right` of type `Node`, and a derived class `Cnode` (thus, `Cnode` is a subtype of `Node`) with an attribute `color` of type `Boolean` (see Figure 4.4 and Listing 4.1 for details).[7]

The `oclAsType()` expressions are not necessary since the OCL standard [41, p. 20] allows their syntactic omission. In OCL [41, p. 8], an invariant of class musst evaluate to `true` for all instances (i.e., object of this class) at any time. Morover, OCL is based on a three-valued logic. These two choices lead to several consequences:

1. Invariants must be valid, i. e., evaluate to `true` and are therefore defined.

2. From the definedness of the invariant and the strictness of `_ < _` follows the

[7] A more detailed discussion of a similar example can be found in section C.1.

definedness of `self.i`.

3. From the definedness of the invariant and the strictness of the equality follows that `self.color` must be defined, if the term `self.oclAsType(Node).left` or `self.oclAsType(Node).right` is defined.

4. From the definedness of the invariant and the strictness of the equality follows that `self.oclAsType(Node).left.oclAsType(Cnode).color` must be defined, if `self.oclAsType(Node).left` is defined. The case for `right` is symmetric.

5. From `self.oclAsType(Node).left.oclAsType(Cnode).color` being defined follows the definedness of `self.oclAsType(Node).left.oclAsType(Cnode)` and thus in this case `left` must be of type `Cnode`. Moreover, this means that `left` must fulfill the invariant for `Cnode`.

4.9.3. Type Constructions

In the following we define several type sets which all are subsets of the types of the HOL type-system. This set, although denoted in usual set-notation, is a meta-theoretic construct, i. e., it cannot be formalized in HOL. We start by defining all possible types for class attributes.

Definition 4.1 (Attribute Types) The set of *attribute types* $\mathfrak{A}$ is defined inductively as follows:
1. $\{\texttt{Boolean}, \texttt{Integer}, \texttt{Real}, \texttt{String}, \mathrm{oid}\} \subset \mathfrak{A}$, and
2. $\{a\ \texttt{Set}, a\ \texttt{Sequence}, a\ \texttt{Bag}, a\ \texttt{OrderedSet}\} \subset \mathfrak{A}$ for all $a \in \mathfrak{A}$. □

Attributes with class types, i. e., the attribute `left` of class `Node`, are encoded using the type oid. These object identifiers (i. e., references) will be resolved by accessor functions for a given state; an access failure will be reported by $\bot$.

Similar to the description in the OCL standard we represent the classes as a pairs of its attribute types which we extend by an an abstract datatype for each class which makes guarantees that each class type is unique. This gives a foundation for a strongly typed universe (with regard to the object-oriented type system).

Definition 4.2 (Tag Types) For each class C we assign a *tag type* $t \in \mathfrak{T}$ which is just an abstract type used to make class types unique. The set $\mathfrak{T}$ is called the set of tag types. □

For class `Node` we assign an abstract datatype $\mathrm{Node}_{\mathrm{t}}$ with the only element $\mathrm{Node}_{\mathrm{key}}$, and we introduce the base type for classes:

Definition 4.3 (Base Class Types) The set of *base class types* $\mathfrak{B}$ is defined as follows:
1. classes without attributes are represented by $(t \times \mathrm{unit}) \in \mathfrak{B}$, where $t \in \mathfrak{T}$ and unit is a special HOL type denoting the empty product.
2. if $t \in \mathfrak{T}$ is a tag type and $a_i \in \mathfrak{A}$ for $i \in \{0, \dots, n\}$ then $(t \times a_0 \times \cdots \times a_n) \in \mathfrak{B}$. □

Thus, the base object type of class `Node` is $\mathrm{Node}_\mathrm{t} \times \mathtt{Integer} \times \mathrm{oid} \times \mathrm{oid}$ and of class `Cnode` is $\mathrm{Cnode}_\mathrm{t} \times \mathtt{Boolean}$.

Without loss of generality, we assume in our object model a common supertype of all objects. In the case of OCL, this is `OclAny`, in the case of Java this is `Object`. This assumption is no restriction because such a common supertype can always be added to a given class structure without changing the overall semantics of the original object model.

Definition 4.4 (OclAny) Let $\mathrm{OclAny}_{\mathrm{tag}} \in \mathfrak{T}$ be the tag of the common supertype `OclAny` and oid the type of the object identifiers,
1. in the *non-referential* setting, we define $\alpha\ \mathtt{OclAny} := (\mathrm{OclAny}_{\mathrm{tag}} \times \alpha_\perp)$.
2. in the referential setting, we define $\alpha\ \mathtt{OclAny} := \big((\mathrm{OclAny}_{\mathrm{tag}} \times \mathrm{oid}) \times \alpha_\perp\big)$. □

Now we have all the foundations for defining the type of our family of universes formally:

Definition 4.5 (Universe Types) The set of all universe types $\mathfrak{U}_{\mathrm{ref}}$ resp. $\mathfrak{U}_{\mathrm{nonref}}$ (abbreviated $\mathfrak{U}_\mathrm{x}$) is inductively defined by:
1. $\mathscr{U}^0_\alpha \in \mathfrak{U}_\mathrm{x}$ is the initial universe type with one type variable (hole) α.
2. $\mathscr{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)} \in \mathfrak{U}_\mathrm{x}$, $n, m \in \mathbb{N}$, $i \in \{0,\dots,n\}$ and $c \in \mathfrak{B}$ then
$\mathscr{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)}\Big[\alpha_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}_\mathrm{x}$
This definition covers the introduction of "direct object extensions" by instantiating α-variables.
3. $\mathscr{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)} \in \mathfrak{U}_\mathrm{x}$, $n, m \in \mathbb{N}$, $i \in \{0,\dots,m\}$, and $c \in \mathfrak{B}$ then
$\mathscr{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)}\Big[\beta_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}_\mathrm{x}$
This definition covers the introduction of "alternative object extensions" by instantiating β-variables. □

The initial universe $\mathscr{U}^0_\alpha$ represents the common supertype (i. e., `OclAny`) of all classes, i. e., a simple definition would be

$$\mathscr{U}^0_\alpha = \alpha\ \mathtt{OclAny}\,.$$

Alternatively one can also encode values $\mathit{Values} = \mathtt{Real} + \mathtt{Integer} + \mathtt{Boolean} + \mathtt{String}$ within the initial universe type, e. g.,

$$\mathscr{U}^0_\alpha = \alpha\ \mathtt{OclAny} + \mathit{Values}\,.$$

For HOL-OCL we choose to represent also values within the universe which makes extensions possible that need to store values within the store. Thus we define the universes as follows:

Definition 4.6 (Referential Universe Types) The *referential universe* $\mathfrak{U}_{\mathrm{ref}}$ is constructed using the referential definition $\alpha\ \mathtt{OclAny} := \big((\mathrm{OclAny}_{\mathrm{tag}} \times \mathrm{oid}) \times \alpha_\perp\big)$ in the defintion of the inital universe:

$$\mathscr{U}^0_\alpha = \alpha\ \mathtt{OclAny} + \mathtt{Real} + \mathtt{Integer} + \mathtt{Boolean} + \mathtt{String}\,. \qquad \square$$

Definition 4.7 (Non-Referential Universe Types) The *non-referential universe* $\mathfrak{U}_{\text{nonref}}$ is constructed using the non-referential definition $\alpha\ \texttt{OclAny} := (\text{OclAny}_{\text{tag}} \times \alpha_{\perp})$ in the defintion of the inital universe:

$$\mathscr{U}^0_\alpha = \alpha\ \texttt{OclAny} + \texttt{Real} + \texttt{Integer} + \texttt{Boolean} + \texttt{String}\,. \qquad \square$$

Extending the initial universe $\mathscr{U}^0_{(\alpha)}$, in parallel, with the classes `Node` and `Cnode` leads to the following universe type:

$$\begin{aligned}\mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})} = \Big(&(\text{Node}_\text{t} \times \texttt{Integer} \times \text{oid} \times \text{oid})\\ &\times \big((\text{Cnode}_\text{t} \times \texttt{Boolean}) \times (\alpha_\text{C})_\perp + \beta_\text{C}\big)_\perp + \beta_\text{N}\Big)\ \texttt{OclAny} + \mathit{Values}\end{aligned}$$

We pick up the idea of a universe representation without values for a class with all its extensions (subtypes). We construct for each class a type that describes a class and all its subtypes. They can be seen as "paths" in the tree-like structure of universe types, collecting all attributes in Cartesian products and pruning the type sums and β-alternatives.

Definition 4.8 (Class Type) The set of *class types* $\mathfrak{C}$ is defined as follows: Let $\mathscr{U}$ be the universe covering, among others, class C_n, and let $C_0, \dots, C_{n-1}$ be the supertypes of C_n, i.e, C_i is inherited from C_{i-1}. The class type of C_n is defined as:
1. $C_i \in \mathfrak{B}, i \in \{0, \dots, n\}$ then
$\mathscr{C}^0_\alpha = \Big(C_0 \times \big(C_1 \times (C_2 \times \dots \times (C_n \times \alpha_\perp)_\perp)_\perp\big)_\perp\Big)_\perp \in \mathfrak{C}$,
2. $\mathfrak{U}_\mathfrak{C} \supset \mathfrak{C}$, where $\mathfrak{U}_\mathfrak{C}$ is the set of universe types with $\mathscr{U}^0_\alpha = \mathscr{C}^0_\alpha$. $\square$

Thus in our example we construct for the class type of class `Node` the type

$$\begin{aligned}&(\alpha_C, \beta_C)\ \texttt{Node} =\\ &\Big((\text{Node}_\text{t} \times \texttt{Integer} \times \text{oid} \times \text{oid}) \times \big((\text{Cnode}_\text{t} \times \texttt{Boolean}) \times (\alpha_C)_\perp + \beta_C\big)_\perp\Big)\ \texttt{OclAny}\,,\end{aligned}$$

and for `CNode` the class type

$$\begin{aligned}&\alpha_C\ \texttt{Cnode} =\\ &\quad\Big((\text{Node}_\text{t} \times \texttt{Integer} \times \text{oid} \times \text{oid}) \times \big((\text{Cnode}_\text{t} \times \texttt{Boolean}) \times (\alpha_C)_\perp\big)_\perp\Big)\ \texttt{OclAny}\,.\end{aligned}$$

Here, α_C allows for extension with new classes by inheriting from `Cnode` while β_C allows for direct inheritance from `Node`. Alternatively, one could omit the lifting of the base types of the supertypes in the definition of class types. This would lead to:

$$\mathscr{C}^0_\alpha = \Big(C_0 \times \big(C_1 \times (C_2 \times \dots \times (C_n \times \alpha_\perp))\big)\Big)_\perp$$

We see our definition as the more general one, since it allows for "partial objects" potentially relevant for other object-oriented semantics for programming languages.

For example Java, for which partial class objects may occur during construction. This paves the way for establishing the definedness of an object "lazy." Furthermore, since the injections and projections are only built to define attribute accessors, partial objects are hidden from the OCL level.

In both cases the outermost $\_{\bot}$ reflects that class objects may also be undefined, in particular after projection from some elements in the universe or from failing type casts. This choice has the consequence that constructor arguments may be undefined.

4.9.4. Treatment of Instances

For each class we provide injections and projects for each class. In the case of `OclAny` these definitions are quite easy, e. g., using the Inl constructors for type sums we can easily insert an `OclAny` object into the inital universe via

$$\mathrm{mk}^{(0)}_{\mathrm{OclAny}}\, o = \mathrm{Inl}\, o \qquad\qquad \text{with type } \alpha\ \texttt{OclAny} \to \mathscr{U}^0_\alpha$$

and the inverse function for constructing an `OclAny` object out of an universe can be defined as follows:

$$\mathrm{get}^{(0)}_{\mathrm{OclAny}}\, u = \mathrm{sumCase}(\lambda\, x.\ x, \lambda\, x.\ \varepsilon\, x.\ \mathrm{true}, u) \qquad \text{with type } \mathscr{U}^0_\alpha \to \alpha\ \texttt{OclAny}.$$

In the general case, the definitions of the injections and projections is a little bit more complex, but follows the same schema: e. g., for the injections we have to find the "right" position in the type sum and insert the given object into that position. Using the following auxiliary functions

$$\begin{aligned}
\mathrm{base}\, x &= \begin{cases} b & \text{if } x = \llcorner(a,b)\lrcorner\,, \\ \mathrm{down} & \text{otherwise,} \end{cases} \\
\sup o &= \mathrm{fst}\ulcorner o\urcorner\,, \text{ and} \\
\mathrm{FromL}\, x &= \begin{cases} \mathrm{sumCase}(\lambda\, x.\ \llcorner x\lrcorner, \lambda\, x.\ \mathrm{down}, x) & \text{if } x = \llcorner v\lrcorner\,, \\ \mathrm{down} & \text{otherwise,} \end{cases}
\end{aligned}$$

the injections for the classes `Node` and `Cnode` can be defined as follows:

$$\begin{aligned}
\mathrm{get}^{(0)}_{\mathrm{Node}}\, o &= \mathrm{get}^{(0)}_{\mathrm{OclAny}}(\sup o, \llcorner_\lrcorner \circ \mathrm{Inl} \circ \ulcorner_\urcorner \circ \mathrm{base})o) \text{ and} \\
\mathrm{mk}^{(0)}_{\mathrm{Cnode}}\, o &= \mathrm{get}^{(0)}_{\mathrm{Node}}\llcorner(\sup o, \llcorner((\sup \circ \mathrm{base})o, (\llcorner_\lrcorner \circ \mathrm{Inl} \circ \ulcorner_\urcorner \circ \mathrm{base} \circ \mathrm{base})o)\lrcorner)\lrcorner
\end{aligned}$$

And analogous for the projectors:

$$\begin{aligned}
\mathrm{get}^{(0)}_{\mathrm{Node}}\, u &= \llcorner((\sup \circ \mathrm{get}^{(0)}_{\mathrm{OclAny}})u, (\mathrm{FromL} \circ \mathrm{base} \circ \mathrm{get}^{(0)}_{\mathrm{OclAny}})u)\lrcorner, \text{ and} \\
\mathrm{get}^{(0)}_{\mathrm{Cnode}}\, u &= \llcorner((\sup \circ \mathrm{get}^{(0)}_{\mathrm{Node}})u, \\
&\qquad \llcorner((\sup \circ \mathrm{base} \circ \mathrm{get}^{(0)}_{\mathrm{Node}})u, (\mathrm{FromL} \circ \mathrm{base} \circ \mathrm{base} \circ \mathrm{get}^{(0)}_{\mathrm{Node}})u)\lrcorner)\lrcorner\,.
\end{aligned}$$

Further, we define in a similar way projectors for all class attributes, e. g., for the attributes `i` and `self` of class `Node` we generate:

$$\begin{aligned} \mathit{self}\,.\,\mathrm{i}^{(0)} &\equiv \mathrm{fst} \circ \mathrm{snd} \circ \mathrm{fst} \circ \ulcorner\_\urcorner \circ \mathrm{base}\,\mathit{self} && \text{with type } (\alpha_C, \beta_C)\ \texttt{Node} \to \mathrm{int}, \text{ and} \\ \mathit{self}\,.\,\mathrm{left}^{(0)} &\equiv \mathrm{fst} \circ \mathrm{snd} \circ \mathrm{snd} \circ \mathrm{fst} \circ \ulcorner\_\urcorner \circ \mathrm{base}\,\mathit{self} && \text{with type } (\alpha_C, \beta_C)\ \texttt{Node} \to \mathrm{oid}. \end{aligned}$$

In a next step, we define type test functions; for universe types we need functions for testing if a universe represents a specific type, i. e., we need to test that the corresponding extensions of the universe type are defined. For `OclAny` we define:

$$\mathrm{isUniv}^{(0)}_{\mathrm{OclAny}}\, u = \mathrm{sumCase}(\lambda\, x.\ \mathrm{true}, \lambda\, x.\ \mathrm{false}, u) \qquad \text{with type } \mathscr{U}^0_\alpha \to \mathrm{bool}.$$

For class types we define two type tests, an exact one that tests if an object is exactly of the given *dynamic type* and a more liberal one that tests if an object is of the given type or a subtype thereof. Testing the latter one, which is called *kind* in the OCL standard, is quite easy. We only have to test, that the base type of the object is defined, e. g., not equal to $\perp$:

$$\mathrm{isKind}^{(0)}_{\mathrm{OclAny}}\, o = \mathrm{def}\, o \qquad \text{with type } \alpha\ \texttt{OclAny} \to \mathrm{bool}.$$

An object is exactly of a specific type, if it is of the given kind and the extension is undefined, e. g.:

$$\mathrm{isType}^{(0)}_{\mathrm{OclAny}}\, o = \mathrm{isKind}^{(0)}_{\mathrm{OclAny}} \wedge \neg(\mathrm{def} \circ \mathrm{base}) o \qquad \text{with type } \alpha\ \texttt{OclAny} \to \mathrm{bool}.$$

The type tests for user defined classes are defined in a similar way by testing the corresponding extensions for definedness.

Finally, we define coercions, i. e., ways to type-cast classes along their subtype hierarchy. Thus we define for each class a cast to its direct subtype and to its direct supertype. We need no conversion on the universe types where the subtype relations are handled by polymorphism. Therefore we can define the type casts as simple compositions of projections and injections, e. g.:

$$\begin{aligned} \mathrm{Node}^{(0)}_{[\mathrm{Object}]} &= \mathrm{get}^{(0)}_{\mathrm{Object}} \circ \mathrm{mk}^{(0)}_{\mathrm{Node}} && \text{with type } (\alpha_1, \beta)\ \mathrm{Node} \to \alpha_{\mathrm{Any}}\ \texttt{OclAny}, \\ \mathrm{Object}^{(0)}_{[\mathrm{Node}]} &= \mathrm{get}^{(0)}_{\mathrm{Node}} \circ \mathrm{mk}^{(0)}_{\mathrm{Object}} && \text{with type } \alpha_{\mathrm{Any}}\ \texttt{OclAny} \to (\alpha_1, \beta_1)\ \mathrm{Node}, \end{aligned}$$

where $\alpha_{\mathrm{Any}} = (\alpha_1, \beta)\ \mathrm{Node} + \beta_{\mathrm{Any}}$, i. e., the free type variable α_{Any} of α_{Any} `OclAny` in the Universe $\mathscr{U}^0_{\alpha_{\mathrm{Any}}}$ is instantiated to match type of the $\mathscr{U}^1_{(\alpha_{\mathrm{C}}, \beta_{\mathrm{C}}, \beta_{\mathrm{N}})}$.

These type-casts are changing the *static type* of an object, while the *dynamic type* remains unchanged, i. e., one can always re-cast an object to the type it was initially created.

Note, for a universe construction without values, e. g.,.

$$\mathscr{U}^0_\alpha = \alpha\ \texttt{OclAny}\,,$$

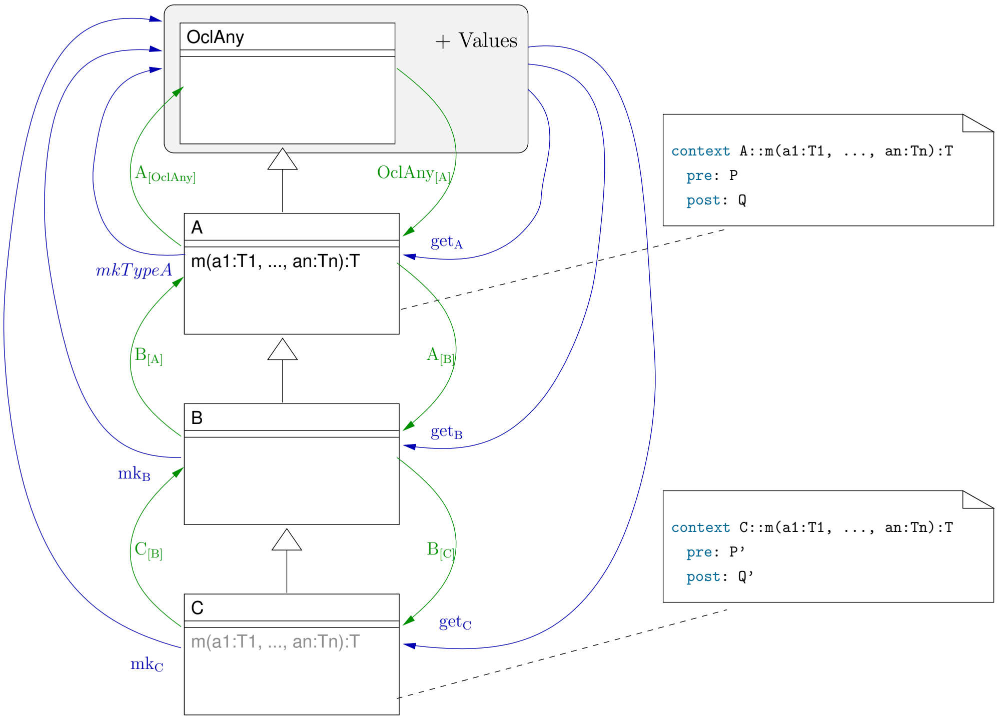


Figure 4.5.: The Relation Between Type Casts and Injections and Projections

the universe type and the class type for the common supertype are the same. In that case there is a particular strong relation between class types and universe types on the one hand and on the other there is a strong relation between the conversion functions and the injections and projections function. In more detail, see also Figure 4.5, one can understand the projections as a cast from the universe type to the given class type and the injections are inverse.

As a consequence, a theorem over class invariants (based finally on these projections, injections, casts, characteristic sets, etc.), it will remain valid even if we extend the universe via α and β instantiations. In particular, this holds if we "finalize" classes by instantiating the α's and β's related to this class by instantiating them by the *unit* type. We consider this fact as a solution to the long-standing problem of structured extensionally for object-oriented languages, enabling to represent "open world" and "closed world" assumptions as polymorphism on data universes.

4.9.5. Adaption to Higher Layers

The previous presented definitions are on the lowest layer, i. e., the introduction of new datatypes. Just as the HOL definition are adopted, over several layers, to match the OCL definitions, we have to adopt the new definitions for the UML core. For example, after the functional and embedding adoptions, the accessor for the attributes `i` is:

$$\mathit{self}\,.\,\mathrm{i}^{(1)} \equiv \mathrm{strictify} \circ \mathrm{lift}_0(\mathit{self}\,.\,\mathrm{i}^{(0)}) \qquad \text{with type } (\alpha_C, \beta_C)\ \mathtt{Node}_\tau \to \mathtt{Integer}_\tau,$$

and for `left`:

$$(\mathit{self}\ \tau)\,.\,\mathrm{left}^{(1)} \equiv \mathrm{upCase}(\mathrm{get}^{(0)}_{\mathrm{Node}}, \bot, \mathrm{fst}\ \tau\ (\mathit{self}\ \tau)\,.\,\mathrm{left}^{(0)})$$

with type (α_C, β_C) $\mathtt{Node}_\tau \to (\alpha_C, \beta_C)$ $\mathtt{Node}_\tau$. The accessor for the `left` is somewhat more complicated as we have to resolve reference, i. e., note the change of types during adaption.

As already shown in these examples, we will use an upper index to distinguish the different definitions on different levels.

4.10. Faithful Representing UML Object Structures

For backing our claim that the presented encoding of object structures models faithfully the concepts that are normally described as object-oriented (e. g., in the sense of programming language like Java or Smalltalk or the UML standard [40]), we prove several of the required properties. These theorems are proven for each class, e. g., during loading a specific UML model. This is similar to other datatype packages in interactive theorem provers like Isabelle/HOL. Further, these theorems are also a prerequisite for a successful reasoning over object structures.

First we prove some basic properties of our injections and projects, i. e., mainly that our conversion between universe representations and object representation is lossless. Therefore we show for very class C that

$$\mathrm{isKind}^{(0)}_{\mathrm{C}}\ o \Longrightarrow \mathrm{get}^{(0)}_{\mathrm{C}}(\mathrm{mk}^{(0)}_{\mathrm{C}}\ o) = o \quad \text{and} \quad \mathrm{isUniv}^{(0)}_{\mathrm{C}}\ u \Longrightarrow \mathrm{mk}^{(0)}_{\mathrm{C}}(\mathrm{get}^{(0)}_{\mathrm{C}}\ u) = u\,.$$

holds. Further, we show that the injection

$$\mathrm{isType}^{(0)}_{\mathrm{C}}\ o \Longrightarrow \mathrm{isUniv}^{(0)}_{\mathrm{C}}(\mathrm{mk}^{(0)}_{\mathrm{C}}\ o)$$

results in a correct universe type and that the projection

$$\mathrm{isUniv}^{(0)}_{\mathrm{C}}\ u \Longrightarrow \mathrm{isKind}^{(0)}_{\mathrm{C}}(\mathrm{get}^{(0)}_{\mathrm{C}}\ u)$$

results in the right class type. We also show the important subtyping relation, i. e., that injections a class C creates a universe type which is of the kind of the parent class P:

$$\mathrm{isKind}^{(0)}_{\mathrm{P}}\ o \Longrightarrow \mathrm{isUniv}^{(0)}_{\mathrm{P}}(\mathrm{mk}^{(0)}_{\mathrm{C}}\ o)\,,$$

and that every universe of type C is also of the kind of the parent class P:

$$\mathrm{isUniv}^{(0)}_{\mathrm{C}}\ u \Longrightarrow \mathrm{isUniv}^{(0)}_{\mathrm{P}}\ u\,.$$

Moreover, we show that our definitions support lossless re-casting, i. e.,

$$\mathrm{isKind}_{\mathrm{C}}^{(0)}\ \mathit{self} \Longrightarrow \mathrm{P}_{[\mathrm{C}]}^{(0)}(\mathrm{C}_{[\mathrm{P}]}^{(0)}\ \mathit{self}) = \mathit{self}$$

4.11. A Constrained Object Store: Encoding Invariants

In this section, we bring OCL and UML together, i. e., we enrich the pure UML data model with invariants which supports the encoding of *recursive* object structures with class invariants. This constructions follows the spirit of the already presented scheme: we begin with a co-recursive construction of type sets (which can be seen as a kind of first level invariant) which later on define a second level construct, providing a representation that looks similar to people involved in object-oriented modeling.

4.11.1. A Co-Recursive Type and Kind Set Construction

In a setting with subtyping, we need two characteristic type sets, a sloppy one, the *characteristic kind set*, and a fussy one, the *characteristic type set*. We define these sets co-recursively. As basis for our co-recursive construction, we built for each invariant a "HOL representation," i. e., in each formula where we replace recursively the logical connectives of OCL with them from HOL by requiring the validness of the subformula. This is done using the *logical judgement* $\tau \vDash P$ which means that the OCL formula P *is valid* (i. e., evaluates to $\top$) in context τ. This is explained in more detail in chapter 5. As we want to use these invariants for a co-recursive construction we parametrize them over the current state τ, the object *self* and the type set C we are constructing.

Recall our previous example (see Figure 4.4 and Listing 4.1), where the class `Node` describes a potentially infinite recursive object structure. The invariant of class `Node` constrains the attribute `i` to values greater than 5. For this constraint, we generate

$$\mathrm{hol\_inv\_range\_of\_i}\,\tau\ C\ \mathit{self} \equiv \tau \vDash \mathit{self}\,.\mathrm{i}^{(1)} > 5$$

Further, we generate the two invariants expressing the fact that `left` and `right` must be either undefined or of type `Node`:

$$\begin{aligned}\mathrm{hol\_inv\_type\_left}\ \tau\ C\ \mathit{self} \equiv{}& \tau \vDash \not\partial\, \mathit{self}\,.\mathrm{left}^{(1)}\\ &\lor \tau \vDash \mathit{self}\,.\mathrm{left}^{(1)} \in (\lambda\, f.\ f\tau)\ \text{`}\ C\\ \mathrm{hol\_inv\_type\_right}\ \tau\ C\ \mathit{self} \equiv{}& \tau \vDash \not\partial\, \mathit{self}\,.\mathrm{right}^{(1)}\\ &\lor \tau \vDash \mathit{self}\,.\mathrm{right}^{(1)} \in (\lambda\, f.\ f\tau)\ \text{`}\ C\end{aligned}$$

Now we define a function for construction the kind set for `Node` which approximates the set of possible instances of the class `Node` and its subclasses:

$$\begin{aligned}\text{NodeKindF} &:: \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow (\alpha_C, \beta_C)\ \texttt{Node}\ \text{set}\\ &\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow (\alpha_C, \beta_C)\ \texttt{Node}\ \text{set}\\ \text{NodeKindF} &\equiv \lambda\,\tau.\ \lambda\,X.\ \big\{\mathit{self}\ \big|\ \text{hol\_inv\_type\_left}\,\tau X\,\mathit{self}\\ &\qquad\qquad\qquad\qquad \wedge \text{hol\_inv\_type\_left}\ \tau\ X\ \mathit{self}\\ &\qquad\qquad\qquad\qquad \wedge \text{hol\_inv\_type\_right}\ \tau\ X\ \mathit{self}\big\}\end{aligned}$$

By adding the conjunct $\tau \models \text{isType}^{(1)}_{\text{Node}}\ \mathit{self}$, we can construct another approximation function (which has obviously the same type as NodeKindF):

$$\begin{aligned}\text{NodeTypeF} &:: \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow (\alpha_C, \beta_C)\ \texttt{Node}\ \text{set}\\ &\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow (\alpha_C, \beta_C)\ \texttt{Node}\ \text{set}\\ \text{NodeTypeF} &\equiv \lambda\,\tau.\ \lambda\,X.\ \big\{\mathit{self}\ \big|\ (\mathit{self} \in (\text{NodeKindF}\ \tau\ X))\\ &\qquad\qquad\qquad\qquad \wedge \tau\,\text{isType}^{(1)}_{\text{Node}}\ \mathit{self}\big\}\end{aligned}$$

Thus, the characteristic kind set for the class `Node` can be defined as the greatest fixedpoint over the function NodeKindF:

$$\begin{aligned}&\text{NodeKindSet} :: \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow \mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})}\ \text{St} \Rightarrow (\alpha_C, \beta_C)\ \texttt{Node}\ \text{set}\\ &\text{NodeKindSet} \equiv \lambda\,\tau.\ (\text{gfp}(\text{NodeKindF}\ \tau))\,.\end{aligned}$$

For the characteristic type set we proceed analogously. Further, we prove automatically, using the monotonicity of the approximation functions, the point-wise inclusion of the kind and type sets:

$$\text{NodeTypeSet} \subset \text{NodeKindSet}$$

This property represents semantically the subtype relation. This kind of theorem remains valid if we add further classes in a class system.

Now we relate class invariants of subtypes to class invariants of supertypes. The core of the construction for characteristic sets taking the class invariants into account is a greatest fixedpoint construction (reflecting their co-algebraic properties). We proceed by defining a new approximation for an inherited class Cnode on the basis of the approximation function of the superclass:

$$\begin{aligned}\text{CnodeF} \equiv \lambda\,\tau.\ \lambda\,X.\ \big\{\mathit{self}\ \big|\ \mathit{self}^{(1)}_{[\text{Node}]} &\in (\text{NodeKindF}\ \tau\ (\lambda\,o.\ \text{o}^{(1)}_{[\text{Node}]})\ \text{`}\ X)\\ &\wedge (\varphi\ \tau\ X\ \mathit{self})\big\}\end{aligned}$$

where φ stand for the constraints specific to the subclass. Note φ must appropriately include $\tau \models \mathit{self}\,.\text{right}^{(1)} \in (\lambda\,f.\ f\tau)\ \text{`}\ X$ to make the implicit recursion in the `Cnode` invariant explicit.

Similar to [42] we can handle mutual-recursive datatype definitions by encoding them into a type sum. However, we already have a suitable type sum together with the needed injections and projections, namely our universe type with the make and get methods for each class. The only requirement is, that a set of mutual recursive classes must be introduced "in parallel," i. e., as *one* extension of an existing universe.

Our construction for type sets and kind sets provides for an object a tight connection between "being of a type" and "fulfilling its invariant," i. e., the invariant for `Node` can be defined semantically as follows:

$$\text{Node\_sem\_inv}\, \mathit{self} \equiv \mathit{self} \in \text{NodeKindSet} \quad \text{with type } (\alpha_C, \beta_C)\ \texttt{Node} \to \texttt{Boolean}_\tau .$$

4.11.2. Functional Adoption: Invariant Awareness

Using our invariant definition we can do the last step needed to combine our OCL embedding with our our embedding of the UML core, i. e., we have to test if an object fulfills its invariant. This can easily done, e. g., for the accessor for attribute `i` of class `Node` we define:

$$\mathit{self}\,.\,\text{i} \equiv \text{if}\, \text{Node\_sem\_inv}\, \mathit{self}\ \text{then}\ \mathit{self}\,.\,\text{i}^{(1)}\ \text{else}\ \bot\ \text{endif}\ .$$

All other, previously introduced definitions, are also adopted by wrapping them in an `if-then-else-endif` statement.

4.11.3. Defining Invariants

The semantic invariant definition previously introduced is not a representation an object-oriented modeler would expect. Therefore we define for each class a "user representation." This representation is based on the accessor introduced in the last section, e. g., for `Node` we define:

$$\begin{aligned} \text{Node\_defined}\, \mathit{self} &\equiv \partial\, \mathit{self}\ , \\ \text{Node\_inv\_range\_of\_i}\, \mathit{self} &\equiv \mathit{self}\,.\,\text{i} > 5\ , \text{ and} \\ \text{Node\_inv}\, \mathit{self} &\equiv \text{Node\_defined}\, \mathit{self} \wedge \text{Node\_inv\_range\_of\_i}\, \mathit{self}\ . \end{aligned}$$

The constraint Node_defined ensures the definedness of *self*, which allows us to prove the equivalence of both invariant representations, e. g., for `Node` we prove:

$$\text{Node\_inv}\, \mathit{self} = \text{Node\_sem\_inv}\, \mathit{self}$$

Thus, we can provide the users of HOL-OCL an invariant representation that looks familiar to the OCL specification:

```
context Node
inv range_of_i: self.i > 5
```

Moreover, this representations allows proving of many system properties without the need of using co-recursive induction schemes.

4.12. An Object-Oriented Type-System

Characteristic Type Sets

Within HOL-OCL we represent types via their characteristic set. Therefore, we lift the both, the characteristic type set and characteristic kind set of each type (i. e., all basic types like Integer and also for all user defined types) introduced in subsection 4.11.1 to the OCL level. For doing so, we need to extend the OCL standard in two ways:

infinite set

- The characteristic set, i. e., the set of all instances, can be infinite (e.g., for the type Integer). Therefore we use an infinite set theory for HOL-OCL. In subsection 5.6.3 we discuss the advantages of this setup in more detail.
- For supporting characteristic sets in front-end tools, we introduce two new operations:

typeSetOf()

```
-- Returns alls possible instances of self, this may be an
-- infinite set. The Type T is equal to self.
OclType::typeSetOf()::Set(T)
```

kindSetOf()

```
-- Returns alls possible instances of self and its subtypes,
-- this may be an infinite set. The Type T is equal to self.
OclType::typeSetOf()::Set(T)
```

For example, they allow one to specify, that the addition on Integers is associative:

```
Integer.typeSetOf()->forall(x,y | (not x.oclIsUndefined()
                                   and
                                   not y.oclIsUndefined())
                                  implies
                                  x + y = y + x)
```

Our encoder will map the the expression `type.typeSetOf()` and `type.kindSetOf()` directly to the corresponding type (kind) set.

4.13. Operation Invocations

We distinguish built-in operations (i. e., all library operations such as the logical operation $\neg X$, the arithmetical operation $X + Y$ or the collection operation $X \cup Y$) and user-defined operations declared in class diagrams.

While built-in operations cannot be overloaded in HOL-OCL in order not to hamper algebraic reasoning on OCL expressions, user-defined operations can be overloaded which is considered as a main feature of object-oriented programming. However, as already discussed in subsection 4.7.2, the OCL 2.0 standard [41] assumes that operation calls are statically known and makes therefore no statement on which overload resolution strategy is used; in earlier documents of the OCL semantics definition, it is suggested to users to be compliant to an arbitrary resolution strategy.

It is clearly feasible to make any *static* resolution strategy explicit in OCL by suitable `->oclIsType()` or `->oclIsKind()` tests on the arguments. However, as an experimental feature, we suggest to extend the OCL 2.0 standard by a *dynamic* late-binding strategy. In this section, we show how the semantics of strict operation invocations is encoded using the semantic combinator for strict invoke defined in 4.7.2.

late-binding invocation

4.13.1. The Invocation Encoding Scheme

Initial Operation Definition

In the following, we show the semantic representation scheme of invocation for user-defined operations by an example. We assume the three classes `A`, `B` and `C`, where `C` inherits from `B` and `B` inherits from `A`, see Figure 4.5 on page 59 for an illustration of this scenario. Further, we assume that an operation `m`, specified in the topmost class `A`, with arguments $a_1 :: T_1, \dots, a_n :: T_n$, return type T, the precondition P, and postcondition Q.

In the following, we will present declaration related infrastructure for invocations as introduced in subsection 4.7.2 more detail. We reconsider the scheme of an operation specification first: an operation m, with arguments $a_1 :: T_1, \dots, a_n :: T_n$, return type T, is declared by a pair of precondition and postcondtion, thus, for each method we introduce a precondition and a postcondition:

$$\begin{aligned} \mathrm{pre}\, \mathit{self}\ a_1\ \dots\ a_n \quad &\text{of type } \big[V_\tau(A), V_\tau(T_1), \dots, V_\tau(T_n)\big] \Rightarrow \mathtt{Boolean}_\tau \text{ and} \\ \mathrm{post}\, \mathit{self}\ a_1\ \dots\ a_n\ \mathit{result} \quad &\text{of type } \big[V_\tau(A), V_\tau(T_1), \dots, V_\tau(T_n), V_\tau(T)\big] \Rightarrow \mathtt{Boolean}_\tau\,. \end{aligned}$$

The precondition pre is a predicate function depending on the input parameter including the implicit input parameter *self*. The postcondition post is a predicate function depending on the input parameter and the implicit *result* parameter. The implicit parameters *self* and *result* are defined in OCL; they represent a common concept in object-oriented languages.

The standard defines the semantics of an operation specification by

$$\tau_{\mathrm{pre}} \vDash \mathrm{pre}\, \mathit{self}\ a_1\ \dots\ a_n\ \wedge (\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) \vDash \mathrm{post}\, \mathit{self}\ a_1\ \dots\ a_n\ \mathit{result},$$

assuming technically two different validity predicates $\vDash$. We represent the former by the latter (replacing all accessors occuring in pre by their `@pre`-variant) and slightly generalize to:

$$S_{\mathrm{m}}\ \mathit{self}\ a_1\ \dots\ a_n\ \mathit{result}\ \tau \equiv \tau \vDash \big((\mathrm{pre}\, \mathit{self}\ a_1\ \dots\ a_n) \wedge (\mathrm{post}\, \mathit{self}\ a_1\ \dots\ a_n\ \mathit{result})\big)$$

allowing to distinguish undefined specifications from contradictory ones. We will abreviate this by S_{m} in the sequel.

Furthermore, we define the totalized_operation_specification:

$$
\begin{aligned}
S_{\text{m-tot}}\ \mathit{self}\ a_1\ \ldots\ a_n\ \mathit{result}\ \tau \equiv \tau \vDash\ & \text{if}\ \partial(\text{pre}\ \mathit{self}\ a_1\ \ldots\ a_n) \land \text{pre}\ \mathit{self}\ a_1\ \ldots, a_n \\
& \text{then}\ \text{post}\ \mathit{self}\ a_1\ \ldots\ a_n\ \mathit{result} \\
& \text{else}\ \not\partial\ \mathit{result} \\
& \text{endif}
\end{aligned}
$$

which can be used alternatively in HOL-OCL and which is preferable for methodological issues, namely for proofs of consistency of OCL specifications (not discussed in this document). We will abreviate this by $S_{\text{m-tot}}$ in the sequel.

Now, when processing the above part of the class declaration, the following HOL declaration of a lookup table tab_{m} is generated:

$$\mathit{tab}_{\text{m}} :: \big(A\ \text{set} \rightharpoonup [V_\tau(A), V_\tau(T_1), \ldots, V_\tau(T_n), V_\tau(T)] \Rightarrow \texttt{Boolean}_\tau\big)$$

Here, the arrow $\_ \rightharpoonup \_$ stands for the type of *partial maps* from the HOL library. The main difference of partial maps compared to total functions $\_ \Rightarrow \_$ is that partial functions have a domain operator dom with type $\alpha \rightharpoonup \beta \Rightarrow \alpha$ set. Additionally, the axiom:

$$\mathit{tab}_{\text{m}}\ \ A\ \equiv \text{Some}(S_{\text{m}}) \tag{4.2}$$

or

$$\mathit{tab}_{\text{m-tot}}\ \ A\ \equiv \text{Some}(S_{\text{m-tot}}) \tag{4.3}$$

where A is the characteristic set of the class `A`. In the concluding subsection, we will discuss the conservativity issue for this type of axioms which is similar, but technically unequal to a constant definition since the table is not defined once and for all, but pointwise for a finite set of arguments.

Inheritance of Operation

Now we consider the case, that the class `B` is declared, but operation m is not redeclared, i. e., inherited from class `A`. This leads to the axioms:

$$\mathit{tab}_{\text{m}}\ \ B\ \equiv \text{Some}(S_{\text{m}}) \tag{4.4}$$

or

$$\mathit{tab}_{\text{m-tot}}\ \ B\ \equiv \text{Some}(S_{\text{m-tot}}) \tag{4.5}$$

Recall that due to our object universe construction, the type of B is an instance of the type of A even if the class `B` has been inserted into the system in a later stage than the compilation of `A`, i. e., A and B live in different universes. Moreover, in the later universe, the property $B \subset A$ holds and has been proven automatically.

Operation Overriding

Now we consider the case of an operation overriding, i. e., a new declaration, we introduce a new specification for the operation m for class C (and its subclasses) with precondition P' and postcondition Q'. Again, see Figure 4.5 for details. Analogously to the overloading case, the two axioms:

$$tab_{\mathrm{m}}\ \ B\ \equiv \mathrm{Some}(S'_{\mathrm{m}})$$

or

$$tab_{\mathrm{m\text{-}tot}}\ \ B\ \equiv \mathrm{Some}(S'_{\mathrm{m\text{-}tot}})$$

were constructed where S'_{m} and $S'_{\mathrm{m\text{-}tot}}$ were the operation specifications or totalized operation specifications as defined in 4.13.1.

4.13.2. Considering Conservativity

The axioms generated in the previous sections are conservative; however, they do *not* fit into one of the standard schemes such as constant definition (the argument of tab_{m} and $tab_{\mathrm{m\text{-}tot}}$ are a changing constant not allowed in this scheme). Rather, it is a *finite family* of constant definitions, where the overall type is refined from universe to universe. In the following, we characterize the syntax of this axiom scheme and sketch a proof of conservativity for it.

Definition 4.9 (Finite Constant Definition Family) A *finite constant definition family* is a theory extension (Σ, A) where Σ is a constant declaration $c :: \tau_1 \to \tau_2$ and A is a finite sequence of axioms of the form $c\ D_1 \equiv E_1\ \ldots c D_n \equiv E_n$ where D_i and E_i are closed expressions. One further optional rule, the *catch-all rule*, has the form: $X \notin \{D_1, \ldots, D_n\} \Longrightarrow c\ X \equiv E_{n+1}$. Furthermore, the following conditions must be satisfied:

1. c does not occur in E_1.
2. c does only occur in E_j (in fact: in no defining expression E in a definition except that the catch-all rule is known) in the form $c\ D_i$ with $i < j$.
3. The type of c in axiom j must be an instance of the type of c in axiom i with $i < j$.
4. All type variables occurring in any type of a subterm of E_j must occur in the type of $c\ D_j$
5. $D_j = D_k \Longrightarrow j = k$, i. e., the D_j must be pairwise disjoint.

□

The catch-all rule is used when a class is *finalized*, i. e., the class cannot be used as starting point for further inheritance.

Theorem 4.1 (Conservativity) *A finite constant definition family is conservative, i. e., provided the original theory (Σ, A) is consistent ("has models"), the theory $(\Sigma \cup \Sigma', A \cup A')$ extended by the extension (Σ', A') is also consistent.* □

PROOF In case that the catch-all rule is unknown, translate the constant definition family into a family of constant definitions with constants c_{D_j}. In case that the catch-all rule is known, the constant definition family can be replaced equivalently by the constant definition $c\ X \equiv \text{if}\ X = D_1\ \text{then}\ E_1\ \text{else if} \ldots \text{else}\ E_{n+1}$. ∎

The axioms above represent a constant definition family since partial maps $\alpha \rightharpoonup \beta$ are just a synonym for $\alpha \Rightarrow \beta$ option. The pairwise disjointness follows from the full inclusion of the characteristic sets assured by construction.

4.13.3. Limits to Recursive Invocations

Let us briefly reconsider why conservativity is fundamental for HOL-OCL: an UML/OCL model can be inconsistent in the sense that there is no state that satisfies all invariants or that a method produces a state contradicting the invariants. However, no axiom generated during the compilation of the UML/OCL model into Isabelle/HOL should introduce a *logical* inconsistency into the meta-logic HOL, i. e., reasoning in itself should never be corrupted. A proof over the consistency of an UML/OCL model in the above sense should be valid in any case, independent of any generated axiom. Therefore, we require that method definitions in class diagrams satisfy the constraints of a finite constant definition family.

This has consequences on the form of admissible recursive operation invocations in HOL-OCL: the conditions 1 and 2 in definition 4.9 rule out a general recursive invocation of the operation to be specified. Consider again the operation m specified in class `C`, which overrides the operation m already defined in class `A`. Assume that the postcondition of m defined in class `C` is given by

```
context C::m(a1:T1,...,an:Tn):Integer
  post: result = 1 + self.m(a1,...,an)
        and self.m(a1,...,an).isDefined()
```

Following subsection 4.7.2, the operation invocation is represented by:

$$\text{Choose}(\text{invokeS}\ \ \mathrm{C}_{[\mathrm{A}]}\ \ tab_{\mathrm{m}}\ \ self\, a_1\ \ \ldots\ \ a_n)$$

which conflicts with tab_{m} directly to condition 2 of definition 4.9.

The example also shows why this kind of syntactic restriction is necessary: from `self.m(a1,...,an) = 1 + self.m(a1,...,an)` and the definedness of the result one can infer `1 = 0` in OCL, and then false in HOL, and then simply everything from there.

The OCL standard vaguely requires that recursions should always be terminating to rule out such kind of problems:

Underspecification of Recursion

> The right-hand-side of this definition may refer to operations being defined (i. e., the definition may be recursive) as long as the recursion is not infinite.
>
> *(OCL Specification [41], page paragraph 7.5.2, pp.16)*

and also:

> For a well-defined semantics, we need to make sure that there is no infinite recursion resulting from an expansion of the operation call. A strict solution that can be statically checked is to forbid any occurrences [...]. However, allowing recursive operation calls considerably adds to the expressiveness of OCL. We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable. *(OCL Specification [41], page A-31)*

Unfortunately, in a proof-environment we have to be sufficiently more specific than this. Furthermore, HOL-OCL is designed to live with the open-world assumption, i. e., with the potential extensibility of object universes, as a default; further restrictions such as finalizations of class diagrams or a self-restriction to Liskov's Principle may be added on top, but the system in itself does not require them. This has the consequence that even in the following version:

```
context C::m(a1:T1,...,an:Tn):Integer
  post: result = if a1.p()
                 then 1 + self.m(a1.q(),...,an)
                 else 0
                 endif
```

the termination for the invocation `self.m(a1.q(),...,an)` is fundamentally unknown (even if `p` and `q` are known and terminating): a potential overriding may destroy the termination of this recursive scheme.

In form of a pre-translation process, operation specifications with a limited form of recursive invocations can be converted into the format that satisfies the constraints of a finite constant definition family. These limited forms are assumed to occur in the postcondition `Q'` and can be listed as follows:

- calls to superclass operations, i. e., `(self.asType(A)).m(x1,...,xn)`, or
- direct recursive well-founded invocations, i. e., `(self.asType(C)).m(x1,...,xn)`.

The former invocation can be translated simply into $\mathrm{Choose}(S_{\mathrm{m}}\ \mathit{self}\ x_1\ \dots\ x_n)$ or $\mathrm{Choose}(S_{\text{m-tot}}\ \mathit{self}\ x_1\ \dots\ x_n)$. The latter is based on the theory of well-founded orders and the well-founded recursor wfrec in Isabelle/HOL, and so to speak an application of the standard HOL methodology to OCL. We assume the following version:

```
context C::m(a1:T1,...,an:Tn):Integer
  post: result = if a1.p()
        then 1 + (self.asType(C)).m(a1.q(),...,an)
        else 0
        endif
```

The idea is to abstract away the occurrence of the recursive call:

$$
\begin{aligned}
\mathrm{post}_{\mathrm{rec}}\, f\ \mathit{self}\ a_1\ \ldots\ a_n \mathit{result} \equiv \mathit{result} \triangleq\ & \mathrm{if}\, a_1.p() \\
& \mathrm{then}\, 1 + (f\ a_1.q())\ \ldots\ a_n \\
& \mathrm{else}\, 0 \\
& \mathrm{endif}
\end{aligned}
$$

and to build the operation specification notions on top of it: Let

$$
\begin{aligned}
S_{\text{m-rec}}\ f\ \mathit{self}\ a_1\ \ldots\ a_n\ \mathit{result}\ \tau \equiv \tau \vDash \big(&\mathrm{pre}\, \mathit{self}\ a_1\ \ldots\ a_n \\
&\land \mathrm{post}_{\mathrm{rec}}\, f\ \mathit{self}\ a_1\ \ldots\ a_n\ \mathit{result}\big)
\end{aligned}
$$

and

$$
\begin{aligned}
S_{\text{m-tot-rec}}\ f\ \mathit{self}\ a_1\ \ldots\ a_n\ \mathit{result}\ \tau \equiv \tau \vDash\ & \mathrm{if}\, \partial(\mathrm{pre}\, \mathit{self}\ a_1\ \ldots\ a_n) \land \mathrm{pre}\, \mathit{self}\ a_1\ \ldots\ a_n \\
& \mathrm{then}\, \mathrm{post}_{\mathrm{rec}}\, f\ \mathit{self}\ a_1\ \ldots\ a_n\ \mathit{result} \\
& \mathrm{else}\, \not\partial\ \mathit{result} \\
& \mathrm{endif}
\end{aligned}
$$

then, for direct recursive calls, the operation specifications are defined as $S_{\mathrm{m}} \equiv \mathrm{Wfrec}\ M\ S_{\text{m-rec}}$ or $S_{\text{m-tot}} \equiv \mathrm{Wfrec}\ M\ S_{\text{m-tot-rec}}$ where Wfrec is the context-lifted version of wfrec and M is an ordering (such as $x < y$ restricted on positive integers). If this ordering is well-founded, from this definition, the original user-specified post-condition follows from this definition. Since the critical call is now incorporated into the well-founded recursion construction, the definition is conservative; and provided the user gives a suitable ordering, it can be shown that the desired specification follows from the constructed definitions.

Alternatively, in case of a finalized class, an invocation

$$\mathrm{invokeS}\ C\ \ \mathit{tab}_{\mathrm{m}}\, \mathit{self}\, a_1\ \ldots\ a_n$$

can by replaced by the case-switch:

$$\mathrm{if}\ \mathit{self}\,\texttt{->IsType}(A)\ \mathrm{then}\ S\ \mathrm{else\ if}\ldots \mathrm{else}\ S''$$

Summing up, conservativity implies that only limited forms of recursive invocations are admissible in HOL-OCL. In an open world (no class finalization so far), only operation calls can be treated, in a closed world (the class hierarchy has been finalized), an invocation can be expanded to a case-switch considering the dynamic type of *self* over calls.

4.14. Comparing the Referential and Non-Referential Universe

Historically, object-oriented systems are equipped with a variety of different "equalities" [30]. Answering the question if two objects are equal is not so obvious: e. g., are two objects equal only if their object identifier is equal (are they the same object?) or are two objects equal if their values are equal, or are they equal if they are observably equivalent with respect to the accessor functions? In the following we will use this problem as a linchpin to discuss the differences between a referential universe and a non-referential universe.

Whereas in traditional specification formalism the equality is defined over values, the most basic equality over objects is the reference quality or identity equality which is also the kind of equality that is usually provided as a default, i. e., "built-in," equality in object-oriented languages. Thus, there is usually a fundamental difference between values and objects.

Definition 4.10 (Value Types) The set of *value types* $\mathfrak{V}$ is defined inductively as follows:
1. $\{\texttt{Boolean}, \texttt{Integer}, \texttt{Real}, \texttt{String}, \} \subset \mathfrak{V}$, and
2. $\{v\ \texttt{Set}, v\ \texttt{Sequence}, v\ \texttt{Bag}, v\ \texttt{OrderedSet}\} \subset \mathfrak{V}$ for all $v \in \mathfrak{V}$. □

Definition 4.11 (Values) An instance of a value type, e.g., $x :: v$ with $v \in \mathfrak{V}$ is called *value*. □

Normally one expects that an equality is an equivalence relation.

Definition 4.12 (Equivalence Relation) An *equivalence relation* $\_ \sim \_$ is a binary relation over a set S for which the following properties hold:
- Reflexivity: $a \sim a$, for all $a \in S$.
- Symmetry: $a \sim b$ if and only if $b \sim a$, for all $a, b \in S$.
- Transitivity: if $a \sim b$ and $b \sim c$ then $a \sim c$, for all $a, b \in S$. □

Now we introduce in an abstract way the basic qualities of object-oriented systems, we ignore undefinedness in these definitions. In a second step, we will show, that the handling of undefinedness is orthogonal and can be combined with any of the following equalities.

Most object-oriented languages have the concepts of references or object identifiers where a reference uniquely identifies an object. Thus it seems a natural choice to use these references for defining an equality, namely the reference quality.

Definition 4.13 (Reference Equality) The *reference equality* or *identity equality* is defined as follows:
1. Two values are reference equal, if they are of the same type and represent the same value.
2. Two objects are reference equal, if their object identifiers (references) are equal. □

Thus, the reference equality tests if two objects represent in fact the same object in a store. If we want to test, if two objects are identical (i. e., they represent the same value) we have two options: a shallow and a deep one:

Definition 4.14 (Shallow Value Equality) The *shallow value equality* or just *value equality* is defined as follows:

1. Two values are shallow value equal, if they are of the same type and represent the same value.
2. Two objects are shallow value equal, if they are of the same type and all attributes with value types are pairwise shallow value equal. □

This definition is not recursive, hence the name shallow equality. The main idea behind the shallow equality is to compare two singular objects as values. In contrast to this, we can define the deep value equality for comparing the values of two object structures.

Definition 4.15 (Deep Value Equality) The *deep value equality* is defined as follows:
1. Two values are deep value equal, if they are of the same type and represent the same value.
2. Two objects are deep equal, if they are of the same type and
a) all attributes with value types are pairwise deep value equal.
b) all attributes with type oid (object type) are pairwise deep value equal if they objects represented bye the object identifiers are deep value equal. □

Summarizing, we have already three different equalities:

1. the reference equality which checks if two objects are in fact the same object,
2. the shallow value equality which compares the values of the attributes on the first level, and
3. the deep value equality which compares recursively the object structure comparing the equality of the corresponding parts.

Assuming a setting, where all values and oid are defined, i. e., the classical two-valued view, each of them is an equivalence relation on objects. It seems to be obvious that that in a universe without undefinedness definition 4.15 refines definition 4.14 and definition 4.14 refines definition 4.13, Thus, two objects, that are reference equal are also shallow equal and deep equal. But if we have undefined values and object it is not clear how these equalities relate to each other. First, in a world with undefinedness we can apply the concept of strictness to equivalence relations:

Taking undefinedness into account, e. g., values and references can be undefined, the setting gets more complicated. First we generalize the concept of equivalence relations by introducing equivalence operators.

Definition 4.16 (Equivalence Operators) An equality operator $\backsim$ is a binary operator that satisfies the following properties for a state τ and a context-passing P:

- Quasi-reflexivity:

$$\frac{\tau \vDash \partial\, x}{\tau \vDash x \backsim x}$$

- Quasi-symmetry:

$$\frac{\tau \vDash \partial\, x \quad \tau \vDash \partial\, y \quad \tau \vDash x \backsim y}{\tau \vDash y \backsim x}$$

- Quasi-transitivity:

$$\frac{\tau \models \partial x \quad \tau \models \partial y \quad \tau \models \partial z \quad \tau \models x \backsim y \quad \tau \models y \backsim z}{\tau \models x \backsim z}$$

- Quasi-substitutivity:

$$\frac{\tau \models \partial x \quad \tau \models \partial y \quad \tau \models x \backsim y \quad \tau \models P(x)}{\tau \models P(y)}$$
□

This definition uses the *logical judgement* $\tau \models P$ which means that the OCL formula P *is valid* (i. e., evaluates to T) in context τ; this judgement is defined and discussed at length in chapter 5.

In the following, we characterize certain classes of three-valued *equality operators*.

Definition 4.17 (Strong Equality) An equality operator is called a *strong equality* if it satisfies the property: $(\bot \triangleq \bot) = \mathsf{T}$. □

This equality operation is quasi-reflexive, quasi-symmetric, quasi-transitive and quasi-substitutive even for undefined values which explains its outstanding importance in deduction.

Applying the concept of strictness to an equality operator results in the following definition:

Definition 4.18 (Strict Equality) An equality operator is called *strict equality* if it evaluates to undefined whenever one of its arguments is undefined, i. e., if the following properties hold:

$$(o \doteq \bot) = \bot\,, \qquad (\bot \doteq o) = \bot\,, \text{ and} \qquad (\bot \doteq \bot) = \bot\,.$$
□

Strictly speaking, these last definitions are merely algebraic *characterizations* and not definitions. These operation symbols were characterized by some properties, but they are obviously not *defined* up to isomorphism. In our context, two interpretations of the equalities into the semantic domain of universes are of particular importance: when comparing objects, we can define the equality operation via HOL-equality in the object representation in the referential or the non-referential universe (when comparing values, we compare them via HOL-equality anyway; see subsection B.4.3 for technical details).

Thus, the concept of strictness is orthogonal to the semantics of equality if the arguments are defined. Thus we can combine this with all of our previous equality variants. In principle this results in six different equalities for the object logic (see also Table 4.1). Albeit, with respect to the interpretation of the equality operators (assuming only objects in the range of the state whose reference field just contains the reference to the object in the store), strong and strict equality operators both coincide with referential equality since we have a bijective mapping between the values of an object and the object identifier. As a consequence, the advantage of a non-referential universe is its potential for efficient implementations of strict and strong equality for an executable fragment of OCL.

	strict	strong (non-strict)
referential equality	$o_1 \doteq o_2$	$o_1 \triangleq o_2$
shallow value equality	$o_1 \dot{\simeq} o_2$	$o_1 \overset{\triangle}{\simeq} o_2$
deep value equality	$o_1 \dot{\approxeq} o_2$	$o_1 \overset{\triangle}{\approxeq} o_2$

Table 4.1.: Equalities in an Object-oriented Setting

$\Longrightarrow$	$\tau \models o_1 \doteq o_2$	$\tau \models o_1 \dot{\simeq} o_2$	$\tau \models o_1 \dot{\approxeq} o_2$	$\tau \models o_1 \triangleq o_2$	$\tau \models o_1 \overset{\triangle}{\simeq} o_2$	$\tau \models o_1 \overset{\triangle}{\approxeq} o_2$
$\tau \models o_1 \doteq o_2$	✓			✓	✓	✓
$\tau \models o_1 \dot{\simeq} o_2$		✓			✓	
$\tau \models o_1 \dot{\approxeq} o_2$		✓	✓		✓	✓
$\tau \models o_1 \triangleq o_2$				✓	✓	✓
$\tau \models o_1 \overset{\triangle}{\simeq} o_2$					✓	
$\tau \models o_1 \overset{\triangle}{\approxeq} o_2$					✓	✓

Table 4.2.: Comparing Equalities (non-referential universe)

The OCL standard defines equality as the strict equality over values [41, Sec. A.2.2], and since objects are values, but object identifiers are not distinguished from object values [41, Definition A.10] we choose the strict reference quality $\_ \doteq \_$ as the default OCL equality. In addition, we provide both referential equalities and for each class both shallow value equalities. . The deep value equalities can be defined by the user within HOL-OCL at the HOL-level, if needed.

strong ($\triangleq$) and strict ($\doteq$) referential equality

strong ($\overset{\triangle}{\simeq}$) and strict ($\dot{\simeq}$) shallow value equality

Note, besides smashed sets and unsmashed sets, we could also define smashed objects (i. e., an undefined attribute would result in an undefined object) and unsmashed objects. This would also influence the relations between the different equalities, i. e., compare Table 4.2 with Table 4.3.

Finally, we report on some strange effect of the `self.OclIsNew()`-operator in the non-referential universe which can be seen as a non-standard behavior: if we assume a class `A` with an attribute `i` and if we assume the natural specification of the creation-method:

```
context A::create(val:Integer):Boolean
post: self.OclIsNew() and self.i = val
```

This operation specification is not satisfiable if the objects has already been created and exists already in the store—this makes the whole concept of `self.OclIsNew()`-operator quite useless in the non-referential universe interpretation of the OCL 2.0 standard.

`self.OclIsNew()` ill-defined

$\Longrightarrow$	$\tau \models o_1 \doteq o_2$	$\tau \models o_1 \dot{\simeq} o_2$	$\tau \models o_1 \dot{\approxeq} o_2$	$\tau \models o_1 \triangleq o_2$	$\tau \models o_1 \overset{\triangle}{\simeq} o_2$	$\tau \models o_1 \overset{\triangle}{\approxeq} o_2$
$\tau \models o_1 \doteq o_2$	✓	✓	✓	✓	✓	✓
$\tau \models o_1 \dot{\simeq} o_2$		✓			✓	
$\tau \models o_1 \dot{\approxeq} o_2$		✓	✓		✓	✓
$\tau \models o_1 \triangleq o_2$				✓	✓	✓
$\tau \models o_1 \overset{\triangle}{\simeq} o_2$					✓	
$\tau \models o_1 \overset{\triangle}{\approxeq} o_2$					✓	✓

Table 4.3.: Comparing Equalities (referential universe)

4.15. Specifying Frame Properties

The OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i. e., it allows arbitrary relations from pre states to post states. For most applications this is too general: there must be a way to express that parts of the state *do not change* during a system transition, i. e., to specify the frame properties of system transition. Thus we suggest to extend OCL one to specify the frame properties explicitly:

->modifiedOnly()

```
(S:Set(OclAny))->modifiedOnly():Boolean
```

where `S` is a set of object identifiers (i. e., a set of `OclAny` objects). This also allows recursive operations to collect the set of objects that are potentially changed within a recursive data structure. Obviously, similar to `@pre` the use of `->modifiedOnly()` is restricted to postconditions.

The semantics for `->modifiedOnly()` can be defined in HOL-OCL quite easily. First we define a predicate oidOf for accessing the object identifier of an object:

$$\text{oidOf}\,\tau\;X \equiv (\{x.\;(\tau\;x) = \text{Some}(\text{mk}_{\text{OclAny}}\,X)\})$$

which allows a uniform definition of `->modifiedOnly()` for the referential universe and the non-referential universe:

$$X\texttt{->modifiedOnly()} \equiv \lambda(\tau,\tau').\quad \llcorner\forall\, i \notin \left(\bigcup (\text{oidOf}\,\tau)\,\text{`}\,\ulcorner\text{RepSet}(X(\tau,\tau'))\urcorner\right).\;\tau\;i = \tau'\;i\lrcorner\,.$$

Thus requiring `Set{}->modifiedOnly()` in a postcondition of an operation allows for stating explicitly that an operation is a query in the sense of the OCL standard, i. e.,, the `isQuery` property is true. Further, constructs like

```
Set{x,y}->modifiedOnly() and Set{z}->modifiedOnly()
```

is semantically the *intersection* of the argument lists, and thus equivalent to the term `Set{}->modifiedOnly()`, i. e., “nothing is changed.”

Chapter 5.

Calculi

One contribution of HOL-OCL is the development of proof calculi for OCL. Having a conservative embedding of OCL in HOL, one might ask why this is necessary. Of course, one could always unfold the definitions, thus converting an OCL formula into a HOL expression, and try to prove the latter. However, this is not an effective proof technique, analogously to the fact that the Turing machine is not necessarily an effective means to simulate an up-to-date microprocessor. Abstract, human-readable statements on OCL formulae are both important for the human user performing interactive proof as well as automated decision- or normalization procedures for certain fragments of OCL. To this means, appropriate logical statements have to be conceived that were linked via derived inference rules (i. e., formally proven in Isabelle/HOL) to each other, such that their use in a programmed tactic process finally yields the necessary automated support for effective interactive proof.

As the OCL standard does not provide any calculi we had to develop them based on previous works [28, 20] for three valued logics and own considerations and experiences with previous interactive proof environments [11]. We extended these works to support of UML-like data-models and we also developed a tool-supported methodology. One central point of the OCL semantics presented in the in the OCL standard [41, appendix A] is the notion of *local validity judgements*:

$$(\tau_{\text{pre}}, \tau_{\text{post}}) \vDash Q \quad \text{iff} \quad I[\![Q]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \text{true}$$

(OCL Specification [41], page A.33)

We use this as a basis of our calculi and generalize this form of judgements to:

$$\big(\tau \vDash P\big) \equiv \big(I[\![P]\!](\tau) = \llcorner\text{true}\lrcorner\big)\,, \tag{5.1a}$$

$$\big(\tau \vDash_{\mathsf{F}} P\big) \equiv \big(I[\![P]\!](\tau) = \llcorner\text{false}\lrcorner\big)\,, \text{ and} \tag{5.1b}$$

$$\big(\tau \vDash_{\perp} P\big) \equiv \big(I[\![P]\!](\tau) = \llcorner\perp\lrcorner\big)\,. \tag{5.1c}$$

As a shorthand for all three variants, we will write $\tau \vDash_x P$ for $x \in \{\perp, \mathsf{F}, \mathsf{T}\}$. When omitting the index, we assume T as default (which will turn out to be complete). Moreover, we generalize local validity judgements to a notion of global or universial judgments::

$$\big(\vDash_x P\big) \equiv \big(\forall\,\tau.\ \tau \vDash_x P\big) \tag{5.2}$$

In principle, reasoning over OCL formulae can either be based on a decomposition strategy of judgements or on exploiting equivalences between formulae or judgements over them; in the latter case, the transport of knowledge of contexts is a major technical issue in reasoning over OCL[1] which turns out to be even more important (i. e., more fundamental) than reasoning over definedness of subterms.

In the following sections, we will explore the potential of deduction via equivalences, in section 5.9 we will discuss a tableaux calculus that can process proof states built from judgements on a completely different way.

5.1. A Theory of Basic Judgement

Following the previous definitions, we can easily check the following link between judgements and equalities:

$$\big(\tau \vDash A\big) = \big(A\,\tau = \llcorner\mathrm{true}\lrcorner\big) = \big(A\,\tau = \mathsf{T}\tau\big) \tag{5.3}$$

The following analgous equations reveal that only one kind of judgements is needed, as canonical form we take the validity judgement:

$$(X = \bot) = (\vDash \partial\!\!\!/\, X) \tag{5.4a}$$
$$(X = \mathsf{F}) = (\vDash \neg\, X) \tag{5.4b}$$
$$(X = \mathsf{T}) = (\vDash X) \tag{5.4c}$$
$$(X\ \tau = \bot\ \tau) = (\tau \vDash \partial\!\!\!/\, X) \tag{5.4d}$$
$$(X\ \tau = \mathsf{F}\ \tau) = (\tau \vDash \neg\, X) \tag{5.4e}$$
$$(X\ \tau = \mathsf{T}\ \tau) = (\tau \vDash X) \tag{5.4f}$$

Applied from right to left, these theorems reveal also the character of judgements as rewrite-rules that can be used by automatic rewriting procedures. From these equalities, the base cases for judgements follow directly:

$$\neg(\vDash \bot) \tag{5.5a}$$
$$\neg(\vDash \mathsf{F}) \tag{5.5b}$$
$$\vDash \mathsf{T} \tag{5.5c}$$
$$\neg(\tau \vDash \bot) \tag{5.5d}$$
$$\neg(\tau \vDash \mathsf{F}) \tag{5.5e}$$
$$\tau \vDash \mathsf{T} \tag{5.5f}$$

A last fundamental fact of judgements is related to the three-valuedness of OCL, i. e., *non quatrium datur*:

$$(\tau \vDash A) \lor (\tau \vDash \neg\, A) \lor (\tau \vDash \partial\!\!\!/\, A) \tag{5.6}$$

With this fact, a defined formula can be converted into formulae which are true or which are false; this gives rise for six corresponding case-split lemmas.

[1]there is a notable similarity to labelled deduction systems [16, 51].

5.2. Basic Equivalences and Congruences

In principle, we distinguish four equivalences over OCL formulae; one of them is already a congruence.

UC: *Universal (Formula) Congruence*. This congruence requires that two formulae A and B, both of type $\texttt{Boolean}_\tau$, agree in all contexts τ and in all three truth values of type $\texttt{Boolean}_\tau$. They have the form

$$A = B \quad \text{or} \quad \frac{A_1 = B_1 \quad \cdots \quad A_n = B_n}{A_{n+1} = B_{n+1}}\,.$$

LE: *Local (Formula) Equivalence*. This equivalence requires that two formulae agree on all three truth values of $\texttt{Boolean}_\tau$ in a specific context τ: They have the form

$$A\ \tau = B\ \tau \quad \text{or} \quad \frac{H_1 \quad \cdots \quad H_n}{A_{n+1}\ \tau = B_{n+1}\ \tau}\,.$$

Premises can have the form $A\ \tau = B\ \tau$ or instances of this scheme such as $\tau \vDash_x A$.

UJE: *Universal Judgement Equivalence*. This equivalence requires that for all contexts τ agree on one value X from *Boolean*. They have the form $\vDash_x A = \ \vDash_x A$ or horn-clauses over them.

LJE: *Local Judgement Equivalence*. This equivalence requires that two formulae agree on a specific truth value X of type $\texttt{Boolean}_\tau$ in a specific context τ: $\tau \vDash_x A = \tau \vDash_x B$ or horn-clauses over them.

Since all three possible kinds of judgements $\vDash_x A$ (universal) respectively $\tau \vDash_x A$ (local) called *validity*, *invalidity* and *undefinedness*, can be converted into each other, we can choose just one of them, validity, as representative. We will abbreviate $\vDash_\mathsf{T} A$ or $\tau \vDash_\mathsf{T} A$ by $\vDash A$ or $\tau \vDash A$ resp.

The UJE-format is only of notational interest: it is not possible to build a complete calculus using only UJE-rules. While the rule:

$$\frac{\vDash \partial\, A \qquad \vDash \partial\, B}{(\vDash A \wedge B) = \big((\vDash A) \wedge (\vDash B)\big)} \tag{5.7}$$

is in fact valid (due to distribution of universal quantification over _ ∧ _), but even an analogue version for the OCL disjunction, however, does not hold.

The LE-format, however, is flexible enough to build both practical relevant conversion calculi as well as (relative) complete calculi. Consider:

$$\frac{\tau \vDash \partial\, A \quad \tau \vDash \partial\, B}{(\tau \vDash A \wedge B) = \big((\tau \vDash A) \wedge (\tau \vDash B)\big)} \tag{5.7}$$

as a propositional equivalence or

$$
\begin{aligned}
(\tau \vDash \forall x \in S.\, A \wedge B) = \big(\forall x.\, (\tau \vDash x \in S) \longrightarrow (\tau \vDash A)\big) \\
\wedge \big(\forall x.\, (\tau \vDash x \in S) \longrightarrow (\tau \vDash B)\big)
\end{aligned} \tag{5.8}
$$

as a predicative equivalence.

Since judgements are propositional and LJE's can be decomposed into implications from left-to-right and from right-to-left, there is another line to automated reasoning over OCL-formulae: they can be turned into a tableau calculus (the *local judgement tableaux calculus*, LTC, see section 5.9).

Unfortunately, there is a trade-off between completeness of the various calculi based on these equivalences and deductive efficiency. UC is the only congruence that can be directly processed by Isabelle's simplifier; normalizations in UC can be computed relatively efficiently. While UC comprises several thousands of rules (among them, the strictness and computational rules of operators) it does not form a complete calculus for principle reasons: Some properties in OCL are inherently context dependent, in particular when referring to paths. Others are difficult to formalize in the form of the straight-jacket of a universal congruence. On the other end of the spectrum, since local judgements are simply propositions, they are extremely flexible. When extending LJE's to equivalences over propositional (predicative) formulae, it is not difficult at all to convert them into a fairly abstract but still provably complete calculus with analytic rules (rules whose application yield to an equivalent transformation of the proofstate. Their use is "safe" in the sense that no logical content is lost.)

With respect to LTC, it is well-known that the complexity for tableaux-based reasoning in Kleene-Logic is unfavorably high [19]. However, the logic is only one little fragment of the overall problem of building decision procedures in OCL: Most operations are strict, and from the data-invariants, definedness of many literals can be inferred, such that large fragments of the language are in fact two-valued. Furthermore, we do not only have the logic but also a rich datatype theory with collection types which give the overall language a flavour in its own. As consequence, a good combination of all these types of calculi is a prerequisite for automated reasoning for practically relevant reasoning work in OCL.

5.3. Reasoning on Context-Passing Predicate cp

In the following, we discuss the first side-calculus, the reasoning over *context-passingness* or cp. Revising the definition

$$
\mathrm{cp} :: (V_\tau(\alpha) \Rightarrow V_\tau(\beta)) \Rightarrow \mathrm{bool} \tag{5.9}
$$

$$
\mathrm{cp}(P) \equiv \big(\exists\, f.\, \forall\, X\tau.\; P\; X\; \tau = f(X\tau)\tau\big) \tag{5.10}
$$

presented in section 4.3, one might wonder why this quite arcane definition is so pivotal for reasoning in OCL. An answer can be drawn from the following lemma:

$$
\frac{A\ \tau = B\ \tau \qquad \mathrm{cp}\, P}{P\ A\ \tau = P\ B\ \tau} \tag{5.10}
$$

$$\mathrm{cp}(\lambda\, X.\ c) \tag{5.11}$$

$$\mathrm{cp}(\lambda\, X.\ X) \tag{5.12}$$

$$\mathrm{cp}(\mathrm{lift}_0\, c) \tag{5.13}$$

$$\frac{\mathrm{cp}\, P}{\mathrm{cp}(\lambda\, X.\ \mathrm{lift}_1\, f\, (P X))} \tag{5.14}$$

$$\frac{\mathrm{cp}\, P \quad \mathrm{cp}\, P'}{\mathrm{cp}\big(\lambda\, X.\ \mathrm{lift}_2\, f\, (P\ X)(P'\ X)\big)} \tag{5.15}$$

Table 5.1.: The Context Passing Side-Calculus (Excerpt)

In other words, any local equivalence (LE) $A\ \tau = B\ \tau$ is in fact a congruence for all terms $P\ X$ that are cp. As a consequence, cp is a pre-requisite for replacing a term through another in some (context-passing) term $P\ X$; P can also be interpreted as the "surrounding term" marked by the "position" X. Since global equivalence is semantically closely connected to strong equality, this means that all sorts of term-rewriting in OCL will be constrained to "adequate surrounding terms," i. e., those that are cp. Context-passingness is a tribute to the fact that OCL is a context dependent typed logic; it can be seen as an invariant of semantic functions representing the OCL operations.

The inference rules for establishing cp are contained in Table 5.1 and follow an inductive scheme over the structure of OCL expressions: The base-cases are straightforward, i. e., constant expressions or identities are context passing. The latter three lemmas contain the step-cases and work for all operators that had been defined via the context lifting combinators.

Since we presented all OCL operations "operator style" with these combinators, these generic step-cases pave the way for the automatic generation of one cp-rule with a uniquely defined pattern for each OCL operator. Thus, for all expressions built entirely from OCL operators—as those formulae generated from the "encoder" when loading class-diagram and an associated OCL specification, the derivation of $\mathrm{cp}\, P$ formulae are done fully automatic by backward chaining (using both by the Isabelle simplifier as well as the Isabelle classical reasoner).

5.4. Reasoning on Undefinedness and Definedness

Definedness and undefinedness are indeed opposite concepts in OCL, i. e., they satisfy the "classical" rule for all X in all types:

$$\tau \vDash \not\partial\, X \vee \tau \vDash \partial\, X \tag{5.16}$$

This gives, of course, rise for case-split techniques that can be applied automatically in UJE or LJE calculi.

However, since OCL is biased towards strict operations, the use of undefinedness in deduction is easier than its counterpart.

Undefinedness can be propagated throughout a proof state via forward reasoning and exploited via rewriting. The forward reasoning part is covered by rules like:

$$\frac{\tau \vDash \not\partial\, X \qquad \mathrm{cp}\, P}{P\ X\ \tau = P\ \bot\ \tau} \tag{5.33}$$

and several variants used for technical purposes. Having replaced some term X by $\bot$, strictness rules like $f\ \bot\ Y$ or $f\ Y\ \bot$ can reduce the size of subgoals drastically.

We now focus on the far more involved treatment of definedness. The core of reasoning over definedness is in fact representable in an UC calculus. It is summarized in Equation 5.2(a). This rule set containes also a class of rules for "strict standard operations f." With this set of operations, we refer to operations that had been defined by constant definitions of the form:

$$f \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \llcorner g\ \ulcorner x \urcorner \ulcorner y \urcorner \lrcorner))) \tag{5.34}$$

As in the case of the generation of cp-inference rules, we exploit the combinator-style definitions of the standard operators here and generate this type of rules in pre-computation steps once and for all.

The power and the main drawback of this type of UC-based calculus stems from the rules listed in Equation 5.2(b). They result in the generation of numerous case splits, which may be inappropriate in most deductions: if we know, that all variables in a subgoal are defined (and this is an important special case that we can achieve by initial case-splits done once and for all), simple conditional rules leading to direct backward-chaining are sufficient.

By the way, this strong definedness calculus can be extended to quantifiers reads as follows:[2]:

$$\frac{\mathrm{cp}\, P}{\partial(\forall\, x \in S \,\mathbf{.}\, P x) = \partial\, S \wedge ((\exists\, x \in S \,\mathbf{.}\, (\neg\, P x)) \vee (\forall\, x \in S \,\mathbf{.}\, (\partial\, P x)))} \tag{5.35}$$

To overcome this drawback, we derived the following alternative rule-set listed in Equation 5.2(c). It reduces the burden of applicability to the question, if the definedness of a term can be derived. We will discuss in section 5.10 how this can be assured by prior normal form computations. By the way, the $\tau \vDash \partial\, x$-part in the last rule is strictly speaking redundant (as we will see when we discuss the Set-theory in more detail), but facilitates the establishment of $\tau \vDash \partial\, P x$ since this additional assumption will be used if x occurs in (the instance of) $P\ x$.

[2]The analogous rule for existential quantification is ommited here.

(a) Core Definedness Rules

$$\partial\,\mathsf{F} = \mathsf{T} \qquad \partial\,\mathsf{T} = \mathsf{T} \tag{5.17}$$

$$\partial\,\bot = \mathsf{F} \qquad \partial\,\partial\,X = \mathsf{T} \tag{5.18}$$

$$\partial(\neg\,X) = \partial\,X \qquad \partial(\neg\,X \wedge \partial\,X) = \mathsf{T} \tag{5.19}$$

$$\partial(X \wedge \partial\,X) = \mathsf{T} \qquad \partial(\partial\,X \wedge \neg\,X) = \mathsf{T} \tag{5.20}$$

$$\partial(\partial\,X \wedge X) = \mathsf{T} \tag{5.21}$$

$$\partial\,X \triangleq Y = \mathsf{T} \qquad \partial\,X \doteq Y = \partial\,X \wedge \partial\,Y \tag{5.22}$$

$$\partial\,f\;X = \partial\,X \qquad \partial\,f\;X\;Y = \partial\,X \wedge \partial\,Y \tag{5.23}$$

for all strict standard operations f

(b) Strong Definedness Rules

$$\partial(\mathrm{if}\,X\,\mathrm{then}\,Y\,\mathrm{else}\,Z\,\mathrm{endif}) = \partial\,X \wedge (X \wedge \partial\,Y \vee \neg\,X \wedge \partial\,Z) \tag{5.24}$$

$$\partial(X \wedge Y) = \partial\,X \wedge \partial\,Y \vee \partial\,X \wedge \neg\,X \vee \partial\,Y \wedge \neg\,Y \tag{5.25}$$

$$\partial(X \longrightarrow Y) = \partial\,X \wedge \partial\,Y \vee \partial\,X \wedge \neg\,X \vee \partial\,Y \wedge Y \tag{5.26}$$

$$\partial(X \vee Y) = \partial\,X \wedge \partial\,Y \vee \partial\,X \wedge X \vee \partial\,Y \wedge Y \tag{5.27}$$

(c) Weak Definedness Rules

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y \quad \tau \vDash \partial\,Z}{\tau \vDash \partial(\mathrm{if}\,X\,\mathrm{then}\,Y\,\mathrm{else}\,Z\,\mathrm{endif})} \tag{5.28}$$

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y}{\tau \vDash \partial(X \longrightarrow Y)} \tag{5.29}$$

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y}{\tau \vDash \partial(X \vee Y)} \tag{5.30}$$

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y}{\tau \vDash \partial(X \wedge Y)} \tag{5.31}$$

$$\frac{\mathrm{cp}\,P}{\begin{array}{l}(\tau \vDash \partial(\forall\,x \in S\,.\,Px)) = (\tau \vDash \partial\,S \wedge ((\exists\,x.\ \tau \vDash x \in S \wedge \neg\tau \vDash Px)\\ \qquad \vee\,(\forall\,x.\ \tau \vDash \partial\,x \wedge \tau \vDash x \in S \longrightarrow \tau \vDash \partial\,Px)))\end{array}} \tag{5.32}$$

Table 5.2.: The Definedness Calculi

5.5. Logical Bridges Between UC, LE, LJE, and LTC

The following characterizations between statements in these logical formats hold:

$$(A = B) = (\forall\,\tau.\ A\ \tau = B\ \tau) \tag{5.36}$$

$$(A = B) = \Big(\big(\forall\,\tau.\ (\tau \vDash A) = (\tau \vDash B)\big) \wedge \big(\forall\,\tau.\ (\tau \vDash \not\partial\, A) = (\tau \vDash \not\partial\, B)\big)\Big) \tag{5.37}$$

$$(A\ \tau = B\ \tau) = \Big(\big((\tau \vDash A) = (\tau \vDash B)\big) \wedge \big((\tau \vDash \not\partial\, A) = (\tau \vDash \not\partial\, B)\big)\Big) \tag{5.38}$$

$$(A = B) = \Big(\big(\forall\,\tau.\ (\tau \vDash A) = (\tau \vDash B)\big) \wedge \big(\forall\,\tau.\ (\tau \vDash \neg\, A) = (\tau \vDash \neg\, B)\big)\Big) \tag{5.39}$$

$$(A\ \tau = B\ \tau) = \Big(\big((\tau \vDash A) = (\tau \vDash B)\big) \wedge \big((\tau \vDash \neg\, A) = (\tau \vDash \neg\, B)\big)\Big) \tag{5.40}$$

$$\big((\tau \vDash A) = (\tau \vDash B)\big) = \big((\tau \vDash A \longrightarrow \tau \vDash B) \wedge (\tau \vDash B \longrightarrow \tau \vDash A)\big) \tag{5.41}$$

$$\big(\tau \vDash (a \triangleq b)\big) = (a\ \tau = b\ \tau) \tag{5.42}$$

The two variants Equation 5.38 and Equation 5.40 implicitly exploit Equation 5.6: If two formulae agree in two truth-values, they have also to agree on the third. There is a third variant that is omitted here.

The characterization Equation 5.41 justifies an own tableaux calculus on the basis of local judgements. This rule is the “entry point” into section 5.9.

The rule Equation 5.42 also shows the connection of strong equality to local formula equivalence LE.

5.6. The Logics

5.6.1. Reasoning over Strong and Strict Equality

The strong equality satisfies the usual properties *except* the Leipnitz rule (substitutivity); instead, the slightly weaker form Equation 5.42 of substitutivity (for context passing contexts P) holds. This side-constraint is not surprising, since since Equation 5.42 shown in section 5.5 we know that strong equality and global equivalence are semantically equivalent.

$$\tau \vDash a \triangleq a \tag{5.42}$$

$$\frac{\tau \vDash a \triangleq b}{\tau \vDash b \triangleq a} \tag{5.42}$$

$$\frac{\tau \vDash a \triangleq b \qquad \tau \vDash b \triangleq c}{\tau \vDash a \triangleq c} \tag{5.42}$$

$$\frac{\tau \vDash a \triangleq b \qquad \tau \vDash P\ a \qquad \mathrm{cp}\, P}{\tau \vDash P\ b} \tag{5.42}$$

The following lemmas show that strict equality is indeed convertible into strong equality and is a stronger equality:

$$\frac{\tau \vDash \partial\, a \quad \tau \vDash \partial\, b}{(a \doteq b)\tau = (a \triangleq b)\tau} \tag{5.42}$$

$$\frac{\tau \vDash a \doteq b}{\tau \vDash a \triangleq b} \tag{5.42}$$

5.6.2. Core-Logic (Boolean)

With *core-logic* of OCL we refer to the sub-language consisting of the logical connectives $\neg\,\_$, $\_\wedge\_$, $\_\vee\_$, etc., which we also call the *propositional fragment* of OCL (although theunderlying semantics is strictly speaking not propositional but Strong Kleene Logic SKL). The core-logic is contained in subsection B.4.2. Besides the computational rules like $\bot \wedge \mathsf{F} = \mathsf{F}$, the core-logic enjoys a lattice-like structure with the rules shown in Equation 5.3(a).

In Equation 5.3(b), the rules are shown that deal with "logical reasoning" related to implication. Note, however, that the UC-rules do not form a complete calculus. The problem is (obviously) hidden in the only conditional rule, which has to be rephrased as LE-rule to achieve completeness. Unfortunately, this conditional rule corresponds to "assumption" and is therefore particularly vital in deduction. The Boolean case-split rule in Equation 5.3(c) is interesting for automated reasoning. Consequent case-splits over all Boolean variables yields a proof procedures rof sufficient power, as can be seen for many facts in the basic library.

5.6.3. Set Theory and Logics

Set theory is the theory of membership $x \in S$ on the one hand and set constructions like *comprehensions* (corresponding to the `->select()`-construct in OCL) on the other.

In OCL, we have a typed form of a set theory which rules out, for example, Russels Paradox. With respect to typedness, OCL's set theory is more related to HOL's set theory, but more distant to ZF. Undefinedness, on the other hand, is a distinguishing feature of OCL's set theory: Inclusion of elements in a set may result in an undefined set or in a set, that just contains undefined elements. As mentioned in subsection 4.7.1, we opt for a smashed sets semantics; the OCL standard is not clear with this respect.

On the deduction level, smashed semantics boils down to the following rule:

$$\frac{\tau \vDash x \in S}{(\tau \vDash \not\partial\, x) \wedge (\tau \vDash \not\partial\, S)} \tag{5.61}$$

This has the consequence, that whenever we eliminate a universal or existential OCL quantification, we know that the variable over which a quantifier ranges is defined. In itself, this is also useful to deduce that the body of a quantifier is defined—which it will usually not.

(a) Lattice

$$\mathsf{F} \wedge X = \mathsf{F} \tag{5.43}$$
$$\mathsf{T} \wedge X = X \tag{5.44}$$
$$\mathsf{T} \vee X = \mathsf{T} \tag{5.45}$$
$$\mathsf{F} \vee X = X \tag{5.46}$$
$$X \ op\ X = X \tag{5.47}$$
$$X \ op\ Y = Y \ op\ X \tag{5.48}$$
$$X \ op\ (Y \ op\ Z) = (X \ op\ Y) \ op\ Z \tag{5.49}$$
$$\text{where } op \in \{\wedge, \vee\} \tag{5.50}$$
$$\neg(\neg X) = X \tag{5.51}$$
$$(X \vee Y) \wedge Z = (X \wedge Z) \vee (Y \wedge Z) \tag{5.52}$$
$$Z \wedge (X \vee Y) = (Z \wedge X) \vee (Z \wedge Y) \tag{5.53}$$
$$(X \wedge Y) \vee Z = (X \vee Z) \wedge (Y \vee Z) \tag{5.54}$$
$$Z \vee (X \wedge Y) = (Z \vee X) \wedge (Z \vee Y) \tag{5.55}$$
$$(X \wedge Y) = \neg(\neg(X) \vee \neg(Y)) \tag{5.56}$$
$$\neg(X \wedge Y) = \neg(X) \vee \neg(Y) \tag{5.57}$$
$$\neg(X \vee Y) = \neg(X) \wedge \neg(Y) \tag{5.58}$$

(b) Logic

$$X \longrightarrow \mathsf{F} = \neg X \qquad X \longrightarrow \mathsf{T} = \mathsf{T} \tag{5.59a}$$
$$\mathsf{F} \longrightarrow X = \mathsf{T} \qquad \mathsf{T} \longrightarrow X = X \tag{5.59b}$$
$$\frac{\partial(X) = \mathsf{T}}{(X \longrightarrow X) = \mathsf{T}} \tag{5.59c}$$
$$X \longrightarrow (Y \wedge Z) = (X \longrightarrow Y) \wedge (X \longrightarrow Z) \tag{5.59d}$$
$$X \longrightarrow (Y \vee Z) = (X \longrightarrow Y) \vee (X \longrightarrow Z) \tag{5.59e}$$
$$(X \wedge Y) \longrightarrow Z = X \longrightarrow (Y \longrightarrow Z) \tag{5.59f}$$
$$(X \vee Y) \longrightarrow Z = (X \longrightarrow Z) \wedge (Y \longrightarrow Z) \tag{5.59g}$$
$$X \longrightarrow (Y \longrightarrow Z) = Y \longrightarrow (X \longrightarrow Z) \tag{5.59h}$$

(c) Boolean Case-Split

$$\frac{P\ \bot = P'\ \bot \quad P\ \mathsf{T} = P'\ \mathsf{T} \quad P\ \mathsf{F} = P'\ \mathsf{F} \quad \mathrm{cp}(P) \quad \mathrm{cp}(P')}{P\ X = P'\ X} \tag{5.60}$$

Table 5.3.: The UC Core-Calculus ("Propositional Calculus")

Further, we allow sets to be infinite. This generalization has the advantage, that general Sets can be used to represent the syntactic category of "types" in the standard as "characteristic sets" in HOL-OCL and enables the possibility to reason over them. From the pragmatics point of view, this also allows for quantifications of sets of "values" in the sense of OCL 2.0. For example, it is possible in HOL-OCL (see section 4.12 for details) to express the commutativity law on Integers *inside* OCL as follows:

```
Integer.typeSetOf()->forall(x,y | (not x.oclIsUndefined()
                                    and
                                    not y.oclIsUndefined())
                                   implies
                                   x + y = y + x)
```

In an infinite set theory, each OCL type can be assigned to a *characteristic set*. Based on the characteristic sets for the base types `Integer`, `Real`, `Boolean`, `String`, we have defined *set constructors* $\mathtt{Set} :: \alpha :: \bot\ \mathtt{Set}_\tau \Rightarrow \alpha\ \mathtt{Set}\ \mathtt{Set}_\tau$ that mimick the semantic effect of type constructors. The construction assures that characteristic sets are always defined objects $\partial\,\mathtt{Integer} = \mathsf{T}, \ldots, \partial\,\mathtt{X} = \mathsf{T} \Longrightarrow \partial\,\mathtt{Set}(\mathtt{X}) = \mathsf{T}$ etc.

Infiniteness of a set in the OCL can be naturally expressed by testing if the cardinality of the set is defined:

$$\partial \| \mathtt{Boolean} \| = \mathsf{T} \qquad\qquad \partial \| \mathtt{Integer} \| = \mathsf{F} \tag{5.62}$$

In contrary, finiteness of sets paves the way for an induction-scheme in HOL-OCL.

In the following, we discuss the core of the collection theories at the example of the set theory. We ommit the type constraints from this presentation which enforce that the overloaded symbols are in fact interpreted as operations on sets.

Quantifiers and set constructors in OCL have a highly operational character with respect to undefinedness; in the standard, quantifiers were defined via iterators, hence fold-like constructs which also reflect the behaviour in case of undefinedness. This represents a particular challenge for deduction; on the other hand, similar languages like Spec# or VDM have the same characterizations, too, and the problem needs to be resolved for the forthcoming generation of "light-weight"-specification languages that are semantically "operation-like" and therefore closer to programming languages.

The combinator-based definition:

$$\begin{aligned}
\forall\, x \in S \,.\, P x \equiv \lambda\,\tau.\ & \text{if}\ \mathrm{def}(S\,\tau) \\
& \text{then if}\ \forall x \in \mathrm{set}\ulcorner \mathrm{RepSequence}(S\,\tau)\urcorner.\ P(\mathrm{lift}_0\,x)\,\tau = (\llcorner\mathrm{true}\lrcorner) \\
& \qquad \text{then}\ \llcorner\mathrm{true}\lrcorner \\
& \qquad \text{else if}\ \exists\, x \in \mathrm{set}\ulcorner \mathrm{RepSequence}(S\,\tau)\urcorner.\ P(\mathrm{lift}_0\,x)\,\tau = (\llcorner\mathrm{false}\lrcorner) \\
& \qquad\qquad \text{then}\ \llcorner\mathrm{false}\lrcorner \\
& \qquad\qquad \text{else}\ \bot \\
& \text{else}\ \bot
\end{aligned}$$

turns out to be equivalent (for the finite case) to the operational formulation of the standard:

$$\frac{\tau \vDash \partial \| S \|}{(\forall\, x \in S \,\bullet\, P\, x)\tau = \big(S\text{->}\mathtt{iterate}(x; y = \mathsf{T} | (P\, x) \land y)\big)\tau} \tag{5.63}$$

Of course, we are interested for the general case of infinite sets. As general UC-rules, we have:

$$\frac{}{\forall\, x \in \bot.\ P\, x = \bot} \tag{5.64}$$

$$\frac{}{\exists\, x \in \bot.\ P\, x = \bot} \tag{5.65}$$

$$\frac{}{\forall\, x \in \emptyset.\ P\, x = \mathsf{T}} \tag{5.66}$$

$$\frac{}{\exists\, x \in \emptyset.\ P\, x = \mathsf{F}} \tag{5.67}$$

$$\frac{\tau \vDash \partial\, X \quad \tau \vDash \partial\, a \quad \operatorname{cp} P}{(\forall\, x \in X\text{->}\mathtt{including}(a).\ P\, x)\ \tau = \big((P\, a) \land (\forall\, x \in X.\ P\, x)\big)\ \tau} \tag{5.68}$$

$$\frac{\tau \vDash \partial\, X \quad \tau \vDash \partial\, Y}{\big(\forall\, x \in (X \cup Y).\ P\, x\big)\ \tau = \big((\forall\, x \in X.\ P\, x) \land (\forall\, x \in Y.\ P\, x)\big)\ \tau} \tag{5.69}$$

The latter two rules allow for the elimination of quantifications over known finite sets via computation.

Besides, there is a tableaux calculus for quantifier elimination can be directly derived from the rules in Equation 5.4(b).

The core of OCL set theory is the relation between the element-hood and the set comprehension (`_ ->select( _ | _)`) and the relation to equality, i. e., a form of set extensionality:

$$\frac{\tau \vDash a \in S \quad \tau \vDash \not\partial(P\,a) \quad \mathrm{cp}\,P}{\left((\!| x \in S | P\,x |\!)\right)\tau = \bot\tau} \tag{5.70}$$

$$\frac{\tau \vDash \partial\,S \quad \tau \vDash \partial\,a \quad \bigwedge x.\ \begin{array}{c}\tau \vDash x \in S\\ \vdots\\ \tau \vDash \partial(P\,x)\end{array} \quad \mathrm{cp}\,P}{\left(\tau \vDash a \in ((\!| x \in S | P\,x |\!))\right) = \left(\tau \vDash P\,a \land \tau \vDash a \in S\right)} \tag{5.71}$$

$$\frac{\tau \vDash \partial\,S \quad \tau \vDash \partial\,a \quad \tau \vDash a \in S \quad \bigwedge x.\ \begin{array}{c}\tau \vDash x \in S\\ \vdots\\ \tau \vDash \partial(P\,x)\end{array} \quad \mathrm{cp}\,P}{\left(\tau \vDash a \in ((\!| x \in S | P\,x |\!))\right) = \left(\tau \vDash P\,a\right)} \tag{5.72}$$

$$\frac{\bigwedge x.\ \begin{array}{c}\tau \vDash \partial\,x\\ \vdots\\ (x \in S)\,\tau = (x \in T)\,\tau\end{array}}{S\tau = T\tau} \tag{5.73}$$

5.7. Arithmetic Computational Rules

An important source of deduction is computation. Computation is needed when $\tau \vDash 3 + 5 \doteq 4$ is refuted. So far, we have only used declarative concepts to introduce numbers; the question arises how this can be used for computation, and even: how can this be used fairly efficiently in deduction.

This problem is by no means new and deeply intertwined with the existing solution in Isabelle/HOL. In the HOL library, a type bin for binary two's complement representation has been introduced—by classical, conservative means. Now, the Isabelle parser is configured to parse a literal like 3 to bitstring representation representing $(101)_2$. Further, an axiomatic class num is defined providing a function declaration numberOf :: bin $\Rightarrow \alpha$ :: num that can be overloaded for each type declared to be an instance of class num. Thus, for new datatypes, just a new function is defined that converts a bitstring representation to this new type. In the library, such a conversion has been provided, for example, for int. Based on this definitions, suitable rules have been derived that perform the integer operations like addition on the two's complement representation directly; these rules can be directly processed by the simplifier.

With respect to the types $\mathtt{Integer}_\tau$, $\mathtt{Real}_\tau$ and $\mathtt{String}_\tau$ we can proceed analogously. For example, after declaring $\mathtt{Integer}_\tau$ to be an instance of num, we provide the following definition numberOf for the representation conversion: rrrrrrr

$$(\mathrm{numberOf} :: \mathrm{bin} \Rightarrow \mathtt{Integer}_\tau)\ b \equiv \mathrm{lift}_0(\llcorner(\mathrm{numberOf} :: \mathrm{bin} \Rightarrow \mathrm{int})b\lrcorner) \tag{5.74}$$

This means, that the "new" numberOf with type bin $\Rightarrow \mathtt{Integer}_\tau$ is the context-lifted, $\bot$-lifted version of the "old" numberOf on integers. From this definition, among others, the following facts follow:

$$\partial(\text{numberOf}\, a) = \mathsf{T} \tag{5.75}$$
$$(\text{numberOf}\, a) + (\text{numberOf}\, b) = \text{numberOf}(a +_2 b) \tag{5.76}$$
$$(\text{numberOf}\, a) \cdot (\text{numberOf}\, b) = \text{numberOf}(a \cdot_2 b) \tag{5.77}$$

$$\frac{\neg\, \text{iszero}(\text{numberOf}(a -_2 b))}{((\text{numberOf}\, a) \triangleq (\text{numberOf}\, b)) = \mathsf{F}} \tag{5.78}$$

Thus, besides definedness-related computations ("all values are defined"), computations in OCL were mapped directly to computations in the underlying meta-logic HOL . This setup enables the standard simplifier of Isabelle to refute judgements like

$$\tau \vDash 3 + 2 \doteq 7 \tag{5.79}$$

fully automatic, i.e., without user interaction.

5.8. Converting OCL to HOL

For a fragment of OCL, that is built for expressions that are always defined, the following "conversion" of OCL formula into standard HOL formulae over local judgements equivalences LJE are possible. The propositional part of the translation is described in Equation 5.4(a), the predicative part in Equation 5.4(b). The rules for the other collection types are accordingly.

5.9. The Judgement Tableaux Calculus LTC

The conversion technique discussed in section 5.8 requires reasoning on side-conditions such as $\text{cp}\, P$ or definedness $\partial\, X$. The question arises, if this can be avoided when performing a logical decomposition of the OCL formulae directly.

The tableaux methodology is one of the most popular approaches to design and implement proof-procedures. While originally geared towards first-order theorem proving, in particular for non-clausal formulae accommodating equality, renewed research activity is being devoted to investigating tableaux systems for intuitionistic, modal, temporal and many-valued logics, as well as for new families of logics, such as non-monotonic and sub-structural logics. Many of these recent approaches are based on a special labeling technique on the level of judgments, called labeled deduction [16, 51]. Of course, labeling can also be embedded into a higher-order, classical meta-logic. Being a special case of a many-valued logic, tableaux calculi for SKL based on labeled deduction have been extensively studied [29, 20, 19]. In this section, we present an experimental tableaux calculus for the predicative fragment of OCL, i. e., for SKL roughly following [29]. It is designed to be processed by Isabelle's generic proof procedures, which are geared towards natural deduction.

(a) Propositional Conversion

$$\frac{\tau \vDash \partial A}{(\tau \vDash \neg A) = (\neg \tau \vDash A)} \tag{5.80}$$

$$\frac{\tau \vDash \partial A \quad \tau \vDash \partial B}{(\tau \vDash A \wedge B) = (\tau \vDash A \wedge \tau \vDash B)} \tag{5.81}$$

$$\frac{\tau \vDash \partial A \quad \tau \vDash \partial B}{(\tau \vDash A \vee B) = (\tau \vDash A \vee \tau \vDash B)} \tag{5.82}$$

$$\frac{\tau \vDash \partial A \quad \tau \vDash \partial B}{(\tau \vDash A \longrightarrow B) = (\tau \vDash A \longrightarrow \tau \vDash B)} \tag{5.83}$$

(b) Predicative Conversion

$$\frac{\tau \vDash \partial(S :: (\beta :: bot)\ \mathtt{Set}_\tau) \quad \mathrm{cp}\, P}{(\tau \vDash \forall x \in S_{\bullet}\ P(x :: \tau \Rightarrow \beta)) = (\forall x.\ \tau \vDash x \in S \longrightarrow \tau \vDash P\ x)} \tag{5.84}$$

$$\frac{\tau \vDash \partial(S :: (\beta :: bot)\ \mathtt{Set}_\tau) \quad \mathrm{cp}\, P}{(\tau \vDash (\exists x \in S_{\bullet}\ P(x :: \tau \Rightarrow \beta))) = (\exists x.\ \tau \vDash x \in S \wedge \tau \vDash (P\ x))} \tag{5.85}$$

Table 5.4.: The OCL to HOL Conversion Rules

Tableau proofs may be viewed as trees where the nodes are lists of formulae. Tableau rules extend the leaves of a tree by a new subtree, i. e., by adding leaves below, where the latter case is called "branching" and is used for case splits. Classical tableau rules are purely analytic: each rule captures the full logical content of the expanded connective. Backtracking from a rule application is never necessary. The goal of the process is to construct trees in a deterministic manner, where the leaves can eventually all be detected as "closed," i. e., a logical contradiction is detected. This last step, however, may be combined with the non-deterministic search for a substitution making this contradiction possible.

The LTC appears in Table 5.5(a).

5.10. Towards Automated Deduction in HOL-OCL

In the current HOL-OCL distribution, only two major OCL-related proof procedures have been implemented:

- OCL_hypsubst
- OCL_subst

They mimick the Isabelle [3] standard tactics hypsubst and subst and serve as an abstract interface to to the various rules that express local congruences, strict and strong

(a) Definedness Introduction and Elimination

$$\frac{\tau \models \partial(A) \qquad \begin{matrix}[\tau \models A]\\ \vdots \\ R\end{matrix} \qquad \begin{matrix}[\tau \models \neg A]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{\begin{matrix}[\neg(\tau \models \neg A)]\\ \vdots \\ \tau \models A\end{matrix}}{\tau \models \partial(A)} \tag{5.86}$$

$$\frac{\begin{matrix}[\neg(\tau \models \neg A)]\\ \vdots \\ \tau \models A\end{matrix} \qquad \tau \models \not\partial A}{\tau \models \neg \partial(A)} \qquad \frac{\begin{matrix}[\neg(\tau \models \neg A)]\\ \vdots \\ \tau \models A\end{matrix} \qquad \tau \models \neg \partial(A)}{\tau \models \not\partial A} \tag{5.87}$$

(b) Negation

$$\frac{\tau \models \neg(\neg A)}{\tau \models A} \quad \frac{\tau \models A}{\tau \models \neg(\neg A)} \quad \frac{\tau \models \not\partial(\neg A)}{\tau \models \not\partial A} \quad \frac{\tau \models \not\partial A}{\tau \models \not\partial(\neg A)} \quad \frac{\tau \models (\neg A)}{\tau \models \neg A} \quad \frac{\tau \models \neg A}{\tau \models (\neg A)} \tag{5.88}$$

(c) Conjunction Introduction and Elimination

$$\frac{\tau \models A \wedge B \qquad \begin{matrix}[\tau \models A, \tau \models B]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{\tau \models \neg(A \wedge B) \qquad \begin{matrix}[\tau \models \neg A]\\ \vdots \\ R\end{matrix} \qquad \begin{matrix}[\tau \models \neg B]\\ \vdots \\ R\end{matrix}}{R} \tag{5.89}$$

$$\frac{\begin{matrix}[\tau \models A, \tau \models B]\\ \vdots \\ R\end{matrix} \qquad \tau \models A \quad \tau \models B \qquad \begin{matrix}[\neg(\tau \models \neg B)]\\ \vdots \\ \tau \models \neg A\end{matrix}}{\tau \models (A \wedge B) \qquad \tau \models \neg(A \wedge B)} \frac{\begin{matrix}[\tau \models \partial(B))]\\ \vdots \\ \tau \models \neg \partial(A)\end{matrix} \qquad \begin{matrix}[\tau \models \partial(A)]\\ \vdots \\ \tau \models A\end{matrix} \qquad \begin{matrix}[\tau \models \partial(B)]\\ \vdots \\ \tau \models B\end{matrix}}{\tau \models \not\partial(A \wedge B)} \tag{5.90}$$

$$\frac{\tau \models \not\partial(A \wedge B) \qquad \begin{matrix}[\tau \models \not\partial A, \tau \models \not\partial B]\\ \vdots \\ R\end{matrix} \qquad \begin{matrix}[\tau \models \not\partial A, \tau \models B]\\ \vdots \\ R\end{matrix} \qquad \begin{matrix}[\tau \models A, \tau \models \not\partial B]\\ \vdots \\ R\end{matrix}}{R} \tag{5.91}$$

(d) Contradictions

$$\frac{\tau \models A \quad \tau \models \neg A}{R} \quad \frac{\tau \models A \quad \tau \models \not\partial A}{R} \quad \frac{\tau \models \neg A \quad \tau \models \not\partial A}{R} \quad \frac{\tau \models \not\partial \partial(A)}{R} \quad \frac{\tau \models A}{\tau \models \partial(A)} \quad \frac{\tau \models \neg A}{\tau \models \partial(A)} \tag{5.92}$$

Table 5.5.: The Core of LTC

equalities and congruences hidden in local judgements as pointed out in section 5.1.

For the moment, this tactic setup allows for a step-by-step reasoning using the rules of the logic and UC-rules (including computational rules) in the Isabelle simplifier. Provided that sufficient information on the definedness of free variables is available in a proof state, this enables a conversion to HOL formulae with the rules discussed in section 5.8 possible. A converted formula can be treated by the standard Isabelle proof procedures like safe_tac, blast_tac and auto_tac possibly intertwined with OCL_hypsubst. This covers to a certain extent logical reason automatically.

However, the situation is clearly not satisfactory for larger, application oriented proof projects in OCL so far. Here is a list of the most painful shortcomings when comparing it with proofs in "pure" Isabelle/HOL:

- The library on datatypes and on datatype-oriented rules is far from being sufficiently developed.
- An arithmetic decision procedure such as arith_tac is missing.
- All rules of the LE-format are not usable by the simplifier. Since the complete core calculus and many datatype-oriented rules are in this format, the proof engineer is limited to elementary proof techniques excluding the simplifier whenever these rules are involved in a proof.
- Rules of the LE-format are also excluded from the classical reasoner, i. e., the fast_tac, blast_tac and auto_tac procedures.
- A combined automatic tactic integrating all these local procedures like the auto_tac-procedure.

Partly, the situation is comparable to HOL 10 years ago—and a fair comparison to similar logical languages has to take into account that the development of HOL libraries and proof procedures needed this time. For OCL, the development of proof procedures and, more critical, the technical support of formal methods based on OCL is still at the beginning. The latter will have to cope with path-expressions, modifies-clauses, and refinement-like situations.

In the following, we will summarize our ideas about potential future tactics to reason over OCL automatically.

With respect to arithmetic, besides a step-by-step reasoning, only the following paths to use automated procedures seem to be viable: defined arithmetic terms have to be converted (by unfolding semantic combinators and blowing away the cascades of definedness-conditions) into pure HOL arithmetic formulae and reuse the existing procedure (the adaption approach). Alternatively, arith_tac must be rewritten to cope with definedness issues (the re-engineering approach).

With respect to rewriting, we see (besides the not very attractive re-engineering approach) the following techniques to adapt to existing Isabelle technology:

Proof-object transformation. Since one can instantiate the simplifier with new equalities obeying the Leipnitz rule, one can run it in an unsafe-mode without checking the cp-side conditions. Proof-objects generated in an unsafe mode could be extended to "full" proof objects where the missing parts are reconstructed. It remains to be explored how costly this approach would be (in development time as well as runtime; previous experience (`blast_tac`) suggests that at least the runtime costs are insignificant).

Making context-passing explicit. One can transform proof-states and rules in a format where context-passingness is encoded directly at all positions in a term. As a consequence, the simplifier can process the transformed rules directly. Additionally to the conversion tactics that perform this term-transformation in forward and backward proof, the major changes for this technique boil down to the management of transformed and un-transformed rule-sets used by the simplifier and `auto_tac`.

It is worth to present the letter option a bit more in detail:

$$\frac{\mathrm{cp}\, P}{P\ X\ \tau = P\big(\mathrm{lift}_o(X\ \tau)\big)\ \tau} \tag{5.93}$$

One can also annotate any redex X explicitly with the context it is referring to. This leads to a linear blowup on the size of terms. On such redeces, LE-rules match directly and can be processed by the simplifier. To enable the simplification of assumptions of a rewriting rule, this technique requires a pre-computation on all rules which should be hidden from the user.

Depending on the decision for the simplification, it is either possible to adopt tableaux calculi for the classical reasoning (however, first experiments showed that the built-in strategies are not compatible with three-valued logic) or to convert a proof-goal to a HOL formula to be processed by standard classical reasoning.

At present, we consider the latter strategy for an automatic setup as more promising. Thus, an `auto_tac` could proceed as follows:

1. general logical simplification and normalization with the UC calculi,
2. performing a definedness case-split on all free variables in a subgoal, which are not already known to be defined or undefined. Moreover, for all potentially undefined subterms (like $1\ \mathrm{div}\ 0$), a suitable case-split is generated,
3. using `OCL_hypsubst`, all $\perp$ were propagated and simplified,
4. a conversion to HOL-formulae is performed (following the rules in 5.8), extended by rules converting equalities on collection types, and simplifying path expressions along their structure,
5. making cp explicit (see above),
6. and apply standard `auto_tac`.

We call the first three phases the *splintering* of an OCL formula, the resulting subgoals are the *splinters*. In principle, there may be exponential many splinters depending on the number of free variables in the formula. Variables bound by OCL quantifier are implicitly defined due to smashed collection semantics. However, due to the fact that expressions are usually built from strict operators, this growth is unlikely in practice. The subsequent phases will deeply interact with the datatype specific rules of the library.

Chapter 6.

The HOL-OCL System Architecture

6.1. An Architectural Overview

HOL-OCL is integrated into a framework [10], which among others provides a model repository, su4sml, written in SML. For a quick overview over main system components of HOL-OCL see Figure 6.1. HOL-OCL is based on the SML interface of Isabelle/HOL and the UML/OCL model repository. As front-end, HOL-OCL provides a special instance of ProofGeneral [5] and a LaTeX based documentation generation.

In this chapter, we will briefly present the main components of the HOL-OCL core system, namely:

- our *datatype package*, or *encoder*, which encodes UML/OCL models into HOL-OCL, i.e., from a user's perspective it provides the XMI import facilities.
- the *HOL-OCL library* which provides the core theorems needed for verification and also provides a formal semantics for OCL.
- the *theory morpher* which derives many of the core OCL theorems by "lifting" them based on the corresponding theorems already proven for HOL.

As further background, we will also give a short introduction of the model repository su4sml.

6.2. The Model Repository: su4sml

The Model Repository *su4sml* [10] provides an interface to models expresseed in the UML core (mainly class diagrams and statemachines) and OCL.

OCL expressions naturally translate into an abstract datatype in SML, as shown in Listing 6.1 and Listing 6.2. This abstract datatype is modeled closely following the standard OCL 2.0 metamodel. In addition to these datatype definitions, the repository structure defines a couple of normalization functions, for example for converting association ends into attributes with corresponding type, together with an invariant expressing the cardinality constraint.

For class models (see Listing 6.3), su4sml resembles the tree structure given by the "containment hierarchy." For example, a class contains attributes, operations, or

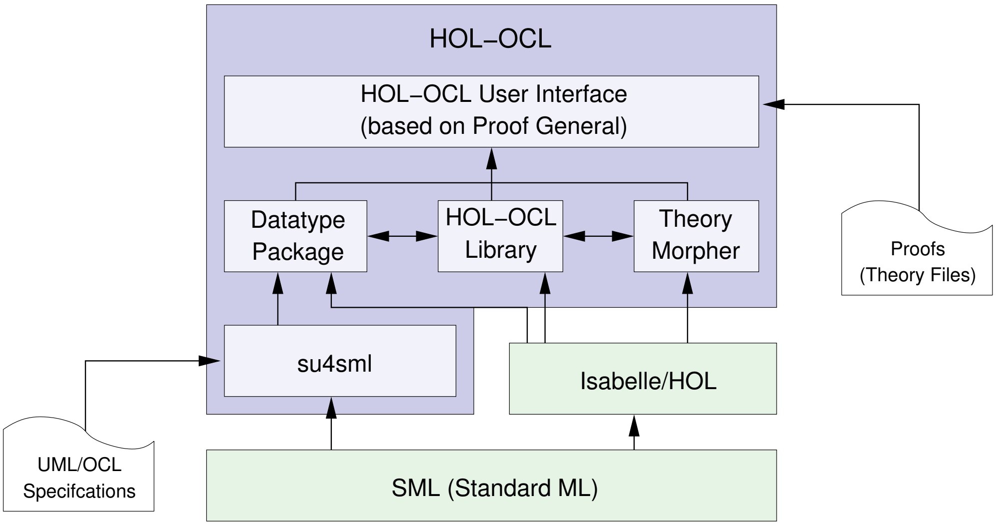


Figure 6.1.: Overview of the HOL-OCL architecture

```
  signature REP_OCL_TYPE = sig

  type Path = string list

  datatype OclType = Integer | Real | String | Boolean      (* Primitive Types *)
                   | OclAny | OclVoid
                   | Set of OclType | Sequence of OclType
                   | OrderedSet of OclType | Bag of OclType
                   | Collection of OclType
                   | Classifier of Path             (* user-defined classifiers *)
                   | DummyT                  (* dummy type for untyped expressions *)
end
```

Listing 6.1: su4sml: Representing OCL types

```
signature REP_OCL_TERM = sig
include REP_OCL_TYPE

datatype OclTerm =
     Literal              of string * OclType                   (* Literal with type  *)
   | CollectionLiteral    of CollectionPart list * OclType      (* content with type  *)
   | If                   of OclTerm * OclType                  (* condition          *)
                           * OclTerm * OclType                  (* then               *)
                           * OclTerm * OclType                  (* else               *)
                           * OclType                            (* result type        *)
   | AssociationEndCall   of OclTerm * OclType                  (* source             *)
                           * Path                               (* assoc.-enc         *)
                           * OclType                            (* result type        *)
   | AttributeCall        of OclTerm * OclType                  (* source             *)
                           * Path                               (* attribute          *)
                           * OclType                            (* result type        *)
   | OperationCall        of OclTerm * OclType                  (* source             *)
                           * Path                               (* operation          *)
                           * (OclTerm * OclType) list           (* parameters         *)
                           * OclType                            (* result tupe        *)
   | OperationWithType    of OclTerm * OclType                  (* source             *)
                           * string * OclType                   (* type parameter     *)
                           * OclType                            (* result type        *)
   | Variable             of string * OclType                   (* name with type     *)
   | Let                  of string * OclType                   (* variable           *)
                           * OclTerm * OclType                  (* rhs                *)
                           * OclTerm * OclType                  (* in                 *)
   | Iterate              of (string * OclType) list            (* iterator variables *)
                           * string * OclType * OclTerm         (* result variable    *)
                           * OclTerm * OclType                  (* source             *)
                           * OclTerm * OclType                  (* iterator body      *)
                           * OclType                            (* result type        *)
   | Iterator             of string                             (* name of iterator   *)
                           * (string * OclType) list            (* iterator variables *)
                           * OclTerm * OclType                  (* source             *)
                           * OclTerm * OclType                  (* iterator-body      *)
                           * OclType                            (* result type        *)
and CollectionPart = CollectionItem of OclTerm * OclType        (* element with type  *)
                   | CollectionRange of OclTerm                 (* first              *)
                                      * OclTerm                 (* last               *)
                                      * OclType                 (* type of range      *)
end
```

Listing 6.2: su4sml: Representing OCL expressions

```
signature REP_CORE = sig
type Scope
type Visibility
type operation =      { name           : string,
                        precondition   : (string option * OclTerm) list,
                        postcondition  : (string option * OclTerm) list,
                        arguments      : (string * OclType) list,
                        result         : OclType,
                        isQuery        : bool,
                        scope          : Scope,
                        visibility     : Visibility }

type associationend = { name           : string,
                        aend_type      : OclType,
                        multiplicity   : (int * int) list,
                        ordered        : bool,
                        visibility     : Visibility,
                        init           : OclTerm option }

type attribute =      { name           : string,
                        attr_type      : OclType,
                        visibility     : Visibility,
                        scope          : Scope,
                        stereotypes    : string list,
                        init           : OclTerm option }

datatype Classifier = Class of { name        : Path,
                                 parent      : Path option,
                                 attributes  : attribute list,
                                 operations  : operation list,
                                 associationends : associationend list,
                                 invariant   : (string option * OclTerm) list,
                                 stereotypes : string list,
                                 interfaces  : Path list,
                                 activity_graphs : ActivityGraph list}
                    | Interface   of { ... }   (* similar to Class *)
                    | Enumeration of { ... }
                    | Primitive   of { ... }
end
```

Listing 6.3: su4sml: Representing the UML core

```
signature REP_ENCODER =
sig
type mdr    = {   theory         : theory,
                  universe       : typ,
                  classifiers    : Classifier list }
  val add_classifiers : Classifier list -> mdr -> mdr
end
```

Listing 6.4: The Top-level Interface of the Repository Encoder

statemachines. We also decided to ignore associations as such. We only represent their association ends, again as part of the participating classifiers.

Overall, the top-level data structures (see Listing 6.1-6.3) of su4sml are inspired by the metamodels of OCL [41, Chapter 8] and UML [40] and readers familiar with these metamodels should recognize the similarities.

6.3. The Encoder: An Object-oriented Datatype Package

Encoding object-oriented data structures in HOL, as needed for HOL-OCL, is a tedious and error-prone activity, which should be automated. In this section, we give an overview of the su4sml-based datatype package we implemented to automate this process. In the theorem prover community, a *datatype package* [33] is a module that allows one to introduce new datatypes and automatically derive certain properties over them. A (conservative) datatype package has two main tasks:

1. generate all required (conservative) constant definitions, and
2. prove as much (interesting) properties over the generated definitions as possible automatically behind the scenes.

For our datatype package, we use the possibility to build SML programs performing symbolic computations over formulae in a logically safe way.

In the following, we give a brief overview what our package does. The datatype package is implemented on top of the su4sml interface on one hand and on top of the Isabelle core on the other (see Listing 6.4 for details). During the encoding, our datatype packages extends the given theory by a HOL-OCL-representation of the given UML/OCL model. This is done in an extensible way, i.e., classes can be added later on to an existing theory preserving all proven properties. The obvious tasks of the datatype package are:

1. declare HOL types for the classifiers of the model,
2. encode the core data model into HOL, and
3. encode the OCL specification and combine it with the core data model.

In fact, the most important task is probably not that obvious: The package has to generate formal proofs that the generated encoding of object-structures is a faithful representation of object-orientation (e.g., in the sense of the UML standard [40], or Java). These theorems have to be proven for each model during its encoding phase. Among many other properties, our package proves for each pair of classes `A` and `B` where `B` is a generalization[1] of `A` the following facts:

$$\frac{\texttt{self.oclIsType(B)}}{\texttt{self.oclIsKind(A)}}\,. \tag{6.1}$$

as well as the more complicated property:

$$\frac{\texttt{self.oclIsDefined()}\quad\texttt{self.oclIsType(B)}}{\begin{array}{l}\texttt{self.oclAsType(A).oclAsType(B).oclIsDefined()}\\ \texttt{and self.oclAsType(A).oclAsTypeB.oclIsType(B)}\end{array}} \tag{6.2}$$

Listing 6.5 presents a simplified version of the SML function `cast_class_id` that proves the property (6.2). The expression starting in line 5 generates a type-checked instance of the current theorem to prove with respect to the current class (and its parent). Readers familiar with LCF-style theorem provers will recognize the "proof script" in lines 10 to 23. Finally, the function registers the proven theorem in Isabelle's theorem database. Logical rules like (6.1) or (6.2) or co-induction schemes given by class invariants constitute the object-oriented datatype theory of a given class diagram and represent the basic weapon for proofs over them, in particular verifications of UML/OCL specifications. Stating these rules could be achieved by adding axioms (i.e., unproven facts) during the encoding process, which is definitively easier to implement. Instead, our datatype package generates entirely conservative definitions and derives these rules from them; this also includes the definition of recursive class invariants, which are in itself not conservative (see section 4.11 for details).

This strategy, i.e., stating entirely conservative definitions and formally proving the datatype properties for them, ensures two very important properties:

1. our encoding fulfills the required properties, otherwise the proofs would fail, and
2. doing all definitions conservatively together with proving all properties ensures the consistency of our model (provided that HOL is consistent and Isabelle/HOL is a correct implementation).

To get a feeling for the amount of work needed, the import of the "Company" model (including the OCL specification) presented in the OCL standard [41, Chapter 7] generates 1147 conservative definitions and proven theorems, the larger "Royals and Loyals" model [52] model generates 2472 conservative definitions and proven theorems.

[1] inherited from

```
fun cast_class_id class parent thy = let
  val pname = name_of parent
  val cname = name_of class
  val thmname = "cast_"^(cname)^"_id"
  val goal_i = mkGoal_cterm
             (Const(is_class_of class,dummyT)$Free("obj",dummyT))
             (Const("op␣=",dummyT)$(Const(parent2class_of class pname,dummyT)
               $(Const(class2get_parent class pname,dummyT)$Free("obj",dummyT)))
                                   $(Free("obj",dummyT)))
  val thm = prove_goalw_cterm thy [] goal_i
            (fn p => [cut_facts_tac p 1,  (* proof script *)
                      asm_full_simp_tac
                         (HOL_ss addsimps
                                [o_def,
                                 get_def thy (parent2class_of class pname),
                                 get_def thy (class2get_parent
                                 class pname )]) 1,
                      stac (get_thm thy (Name mk_get_parent)) 1,
                      asm_full_simp_tac (HOL_ss addsimps [
                            get_def thy (is_class_of class),
                            get_thm thy (Name ("is_"^pname^"_mk_"^(cname)))]) 1,
                      stac (get_thm thy (Name ("get_mk_"^(cname)^"_id"))) 1,
                      ALLGOALS(simp_tac (HOL_ss))])
in
  (fst(PureThy.add_thms [((thmname,thm),[])] (thy)))
end
```

Listing 6.5: Proving Cast and Re-Cast (simplified)

6.4. The HOL-OCL-Library

An important part of HOL-OCL is a collection of Isabelle theory files describing the formalization in detail. These theories also contain new proof procedures (tactics) written in SML. These theory files are extensively documented in Appendix B in the appendix, the developed tactics are also described in section 9.3.

6.5. The Theory-Morpher

The *theory morpher* provides automatic support lifting theorems from the HOL level to the HOL-OCL level. This is based on our organization of thei. e.., library function definitions for our typed shallow embedding in a layered theory morphism. The theory morpher, or lifter is in principle a tactic-based program that lifts meta-level theorems to their object-level counterparts and meta-level prover configurations to object-level ones.

Our approach can be seen as an attempt to liberate the shallow embedding technique from the "point-wise-definition-style" in favor of more global semantic transformations from one language level to another. We abstracted the underlying conceptual notions into a generic framework that shows that the overall technique is applicable in a wide range of embeddings in type systems; embedding-specific dependencies arise only from the specifications of semantic combinator (the *layers*), and technology specific dependencies from the used tactic language.

In the present version, the tactic proof engine cannot handle data adaption invariants which limits its potential to 10%. However, this limitation will be overcome soon.

Part III.

System Description

Chapter 7.

Installing HOL-OCL

7.1. Prerequisites

HOL-OCL is build on top of Isabelle/HOL-Complex, version 2005, thus you need an working installation of *Isabelle 2005*. At the moment, HOL-OCL requires an Isabelle based on sml/NJ [6] to be full functional.[1] We strongly recommend also to install the generic proof assistant front-end *Proof General* [5].[2] The overall system architecture is depicted in Figure 7.1. For doing larger applications, you should install a complete tool-chain comprising a CASE-Tool, e. g., ArgoUML, and an OCL type-checker (from the Dresden OCL 2 toolkit) for generating type-correct UML/OCL models in a comfortable way (see Figure 9.1 for an overview of the proposed HOL-OCL work flow).

7.2. Installation

In the following we will give a brief guide how to install the prerequisites for an HOL-OCL base system (i. e., HOL-OCL and Isabelle), please refer to the respective documentation for details. We will not describe how to install the CASE-tool and the OCL type checker, but we recommend to install them in any case. We have tested our XMI import using the following setup:

[1]The XMI inport uses internally the `Word32` structure which is not supported by Poly/ML [4].
[2]Currently HOL-OCL was only tested on GNU/Linux systems on the i386 architecture, please report your experiences when you try to install HOL-OCL on a different operating system or architecture.

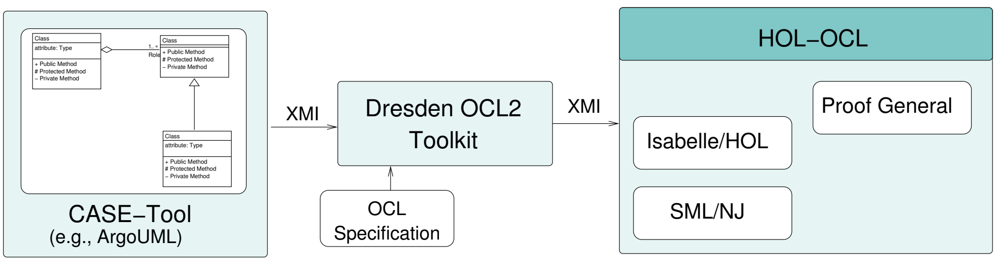


Figure 7.1.: Overview of the High-level System Architecture of HOL-OCL

- A recent version (0.20 or higher) of the CASE-tool "ArgoUML" (`http://argouml.tigris.org/`).
- The type checker "Dresden-OCL 1.1" for OCL 2.0 (`http://dresden-ocl.sf.net`). Our toolchain only relies on the "Parser GUI" which can be started using the command

  ```
  java -jar ocl20parsertools.jar
  ```

  within the top-level directory of the Dresden OCL 2 toolkit. Alternatively, a command-line version of the type-checker is available:

  ```
  java -cp ocl20parsertools.jar tudresden.ocl20.core.parser.OCL20CLI
  ```

7.2.1. Installation from Source

SML and Isabelle

1. Download a recent version of sml/NJ [6] install it following its documentation. We recommend version 110.56 which can be downloaded from `http://www.smlnj.org/dist/working/110.56/index.html`.
2. Download the Isabelle 2005 source code[3] from the from the Isabell web-site (`http://isabelle.in.tum.de/dist/packages.html`), after unpacking the Isabelle source, *replace* the file `src/Pure/defs.ML` with `contrib/defs.ML` which is part of the HOL-OCL distribution.[4] After that you can follow the normal instructions for building Isabelle. Note that you need to build the *HOL-Complex* image, i.e., you have to call

   ```
   $ISABELLE_HOME/build -m HOL-Complex HOL
   ```

3. Finally install a version of Proof General that supports Isabelle 2005, e.g, we strongly recommend a 3.6 pre-release. E.g., download `http://proofgeneral.inf.ed.ac.uk/releases/ProofGeneral-3.6pre051004.tar.gz` and follow the supplied instructions.

Installing HOL-OCL

In the following we assume that you have a running Isabelle 2005 environment including the Proof General based front-end. The installation of HOL-OCL requires the following steps:

1. Unpack the HOL-OCL distribution, e.g.:

   ```
   tar zxvf holocl-0.9.0.tar.gz
   ```

[3] Note that you cannot use the pre-compiled heap images as they are based on Poly/ML.
[4] This fixes a performance bug in Isabelle 2005 which renders our XMI import unusable.

This will create a directory `holocl-0.9.0` containing the HOL-OCL distribution.

2. Check the settings in the configuration file `holocl-0.9.0/make.config`. If you can use the `isatool` tool from Isabelle on the command line, the default settings should work.

3. Change into the `src` directory

   ```
   cd holocl-0.9.0/src
   ```

   and build the HOL-OCL heap image for Isabelle by calling

   ```
   isatool make
   ```

4. HOL-OCL queries the enviornment variable `$HOLOCL_HOME` for finding runtime configurations. Thus you have to set this variable pointing to the top-level directory of your HOL-OCL installation.

5. Finally, you need to extend your `~/.emacs` file with the following lines:

   ```
   (add-to-list 'load-path (concat (getenv "HOLOCL_HOME") "/etc/"))
   (load "x-symbol-holocl-startup") ;; Proof General extension
                                    ;; for HOL-OCL
   ```

   This will make Proof General aware of the new syntax and commands provided by HOL-OCL. Further, if you are using GNU Emacs, you can also add the following lines to you `~/.emacs` file:

   ```
   (require 'ocl)                   ;; simple mode for editing
                                    ;; OCL files
   ```

   This will provide syntax highlighting for plain OCL specifications.

7.2.2. Using Debian Packages

Installing HOL-OCL on an Debian GNU/Linux on a i386 architecture should be straight forward. Just add the IsaMorph apt-repostory to the sources of your package manager, e.g. by adding the following lines

```
# IsaMorph repository
deb-src http://kisogawa.inf.ethz.ch/isamorph/debian stable main
deb     http://kisogawa.inf.ethz.ch/isamorph/debian stable main
```

to `/etc/apt/sources.list` file. Please replace `stable` by the distribution you are using (we provide packages for all three flavours, i.e., `stable`, `testing` or `unstable`). After that, update your package list, i.e., by executing

```
aptitude update
```

Now install a complete (eventually you have to install other logics, if you need them) Isabelle setup by executing

```
aptitude install x-symbol proofgeneral-misc isabelle \
                 isabelle-thy-hol-complex
```

and HOL-OCL by executing

```
aptitude install hol-ocl
```

This should give you a running installation of sml/NJ, Isabelle, Proof General and last but not least, HOL-OCL in its default configuration (based on smashed sets and a referential universe). If you need a different setup, please install HOL-OCL from source and change the configuration in the `make.config` file.

7.3. Starting

Regardless of the installation method you are using, you should now be able to start HOL-OCL using the command:[5]

```
Isabelle -L HOL-OCL
```

As HOL-OCL provides new top-level commands, the `-L HOL-OCL` is *mandatory*. After a few seconds you should see a Emacs window similar to the one shown in Figure 7.2.

[5]If you installed the Debian packages, you can also start HOL-OCL using the command `hol-ocl` or start HOL-OCL using the menu entry of your favourite desktop environment.

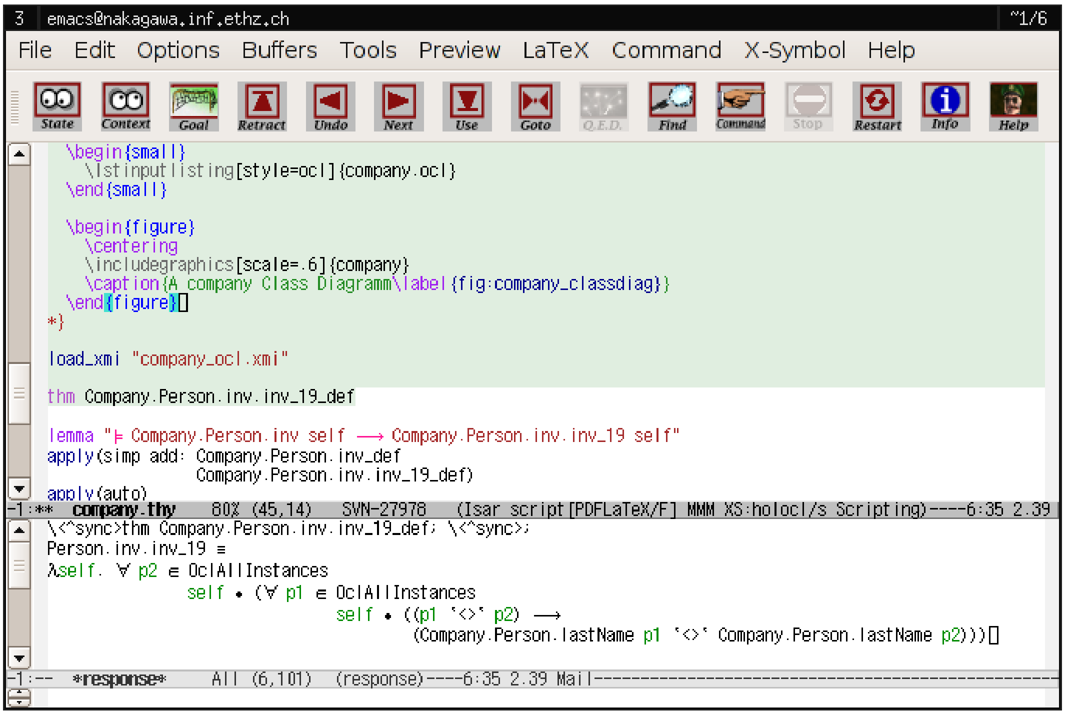


Figure 7.2.: A HOL-OCL session Using the Isar Interface of Isabelle

Chapter 8.

Limitations of HOL-OCL

In this chapter we give a brief overview of the supported OCL/UML subset and current limitations of HOL-OCL.

8.1. The Supported UML Subset

HOL-OCL aims for supporting a relevant subset of the "UML Core" module, i.e., more sloppy: the subset needed to express UML class diagrams.[1] The most restricting limitations are:

- *Enumeration* are not supported, but planned for a future release.
- *Associations* are represented by their associations ends (with additional constraints). Direct association support (i.e., as relations) is planned.
- *Association classes* are not supported, support is planned together with the direct support for associations.
- Qualifiers for *association ends* are not supported.
- *Ordered associations ends* have the type `Sequence` by the Dresden OCL Toolkit which is thus also the type used by HOL-OCL. In contrast, the UML standard [40, p. 3-71] requires the representation as `OrderedSet` (i.e., the elements of the set have an ordering, but duplicates are still prohibited).

Further note that the UML standard [40, p. 5-10] defines the following primitive datatypes: `Integer`, `UnlimitedInteger`, `String`, and `Enumerations`. We consider these datatypes different from the datatypes defined in the OCL standard [41], e.g. HOL-OCL supports the OCL types `Integer`, `Real`, `String`, and `Boolean` (the datatype `enumeration` is not supported at the moment). During modeling, we advise you to place the OCL datatypes in a package called `UML_OCL` (as proposed by the UML standard version 1.4).

[1] Support for dynamic diagram types, i.e., activity charts or state charts is planned for a future release.

8.2. The Supported OCL Subset

8.2.1. OCL Syntactical Variants

In general, the OCL standard introduces a lot of syntactical variants to reduce the amount of text one has to write, i.e., bound variables do not need to be anotated with a concrete type, or can even be ommitted. For example, the following to invariants are only syntactical different:

```
context A:
inv: self.b->select(i=5)
inv: self.b->select(a | a.i =5 )
```

At the moment, our tool chain does not support all these variants, in particular, one has to make bound variable expcplicit and has to anotate them with type information:

```
context A:
inv: self.b->select(b:B| b.i = 5)
```

Furthermore, introducing alternative names for `self`, i.e.:

```
context foo:A:
inv: foo.i = 5
```

is currently not supported.

8.2.2. OCL context declarations

The OCL standard [41, pp. 157] introduces the different context declarations, which are handled by HOL-OCL as follows:

inv: fully supported.

pre: fully supported.

post: fully supported.

init: are supported by converting them into an invariant, e.g.

```
context A:
init: self.x = 5
```

will be converted into

```
context A:
inv: self.oclIsNew() implies self.x = 5
```

Note, this formulae can be considered as non-standard with regards to the OCL standard but is valid in HOL-OCL.

def: not supported due to Dresden OCL (will be supported after Dresden OCL is fixed),i.e., for HOL-OCL there is no difference between statements defined (graphical) within the case tool or as OCL formulae.

derive: not supported due Dresden OCL (will be supported after Dresden OCL is fixed), i.e., for HOL-OCL there is no difference between statements defined (graphical) within the case tool or as OCL formulae.

body: supported by converting them into a post condition, e.g.

```
context A::f():Integer
body: 5
```

will be converted into

```
context A::f():Integer
post: result = 5
```

guard: not supported, support will be developed once we support dynamic diagram types.

8.2.3. OCL Types

OCL introduces a variety of pre-defined types (and meta-types), for some of them certain restrictions apply within HOL-OCL. In more detail:

- The type `OclVoid` is modeled implicit, for details see the lifting construction and the handling of the type class bot.
- The type `OclMessage` is not support, support for it will be eventually developed together with the support for dynamic diagram types (e.g. activity charts).
- *Enumeration* are not supported, but planned for a future release.
- The types `OclModelElementType` and `OclType` are modeled implicit, respectively replaced by the type of the characteristic set (of a type).
- The enumeration `OclModelElement` is not supported, respectively modeled implicit.
- The type `Tuple` (respectively `TypleType`) is not yet supported. Support for tuples is planned for a future release.
- The type `OrderedSet` is not yet supported by the Dresden OCL Toolkit but is supported by HOL-OCL. If you want to use `OrderedSet` you have to enter the coresponding constraints directly into HOL-OCL.

8.2.4. Predefined Properties

OCL introduces a variety of pre-defined properties and operations, for some of them certain restrictions apply within HOL-OCL. In more detail:

- `OclInState` is not support at the moment; it will be eventually developed together with the support for dynamic diagram types (e.g. state charts and activity charts).
- The generic "`->iterate()`" expression is at the moment not supported by the XMI import. Nevertheless, it can be entered directly within HOL-OCL. The predefined iterator expressions, e.g. `->one` or `select`, are fully supported.
- Collection literals, e.g., `Set{1,3,4,9}` or `Sequence{1,..,42}` are at the moment not supported by the XMI import. Nevertheless, they can be entered directly within HOL-OCL.

8.3. Reporting Bugs

If you find a bug in HOL-OCL, please send electronic mail to hol-ocl@brucker.ch. Include the version number, which you can find by running the `info()` in the interactive HOL-OCL-environment. Also include in your message the output that the program produced and the output you expected.

If you have other questions, comments or suggestions about HOL-OCL, contact the author via electronic mail to hol-ocl@brucker.ch. The authors will try to help you out, although they may not have time to fix your problems.

Chapter 9.

A HOL-OCL Reference Manual

9.1. The Basic Workflow

HOL-OCL allows one to reason formally over UML/OCL specifications. The overall HOL-OCL work-flow (see Figure 9.1 for an overview) is divided into two phases:

Modelling Phase: During the modeling and design phase one formalizes the (often informal given) requirements. The result of this phase should be a formal model of the software system being built (in our case, formalized using UML/OCL). Technically, the modelling phase is twofold:

1. design your (data) model using a CASE tool, e.g., ArgoUML and export it in XMI format.

2. refine your data model by adding OCL constraints using the Dresden OCL Toolkit. Export your refined model (including the OCL formulae in XMI format.

Verification Phase: In the analyzing and verification phase, the UML/OCL model is formally explored using a Isar-based [53] environment. Such a formal analysis can for example aim towards proving the consistency (i.e., there exists a system that fulfills the requirements of the model), security of safety. Further, the model can be further improved, i.e., by doing formal refinements.

In the following, we will give a brief overview of the modelling phase and describe the verification phase in more detail.

9.2. The Modelling Phase

In this section, we give a brief overview of the modelling phase, i.e., we concentrate ourselves on describing the most important tasks and pitfalls of this phase. Please refer to the manuals of your CASE tool and Dresden OCL for more information about the technical details of the modelling phase.

Overall, the modeling phase is split into two parts:

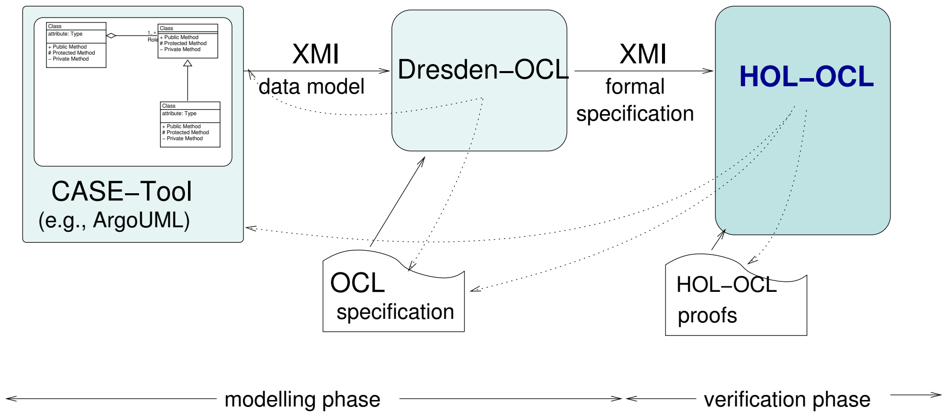


Figure 9.1.: The HOL-OCL Workflow

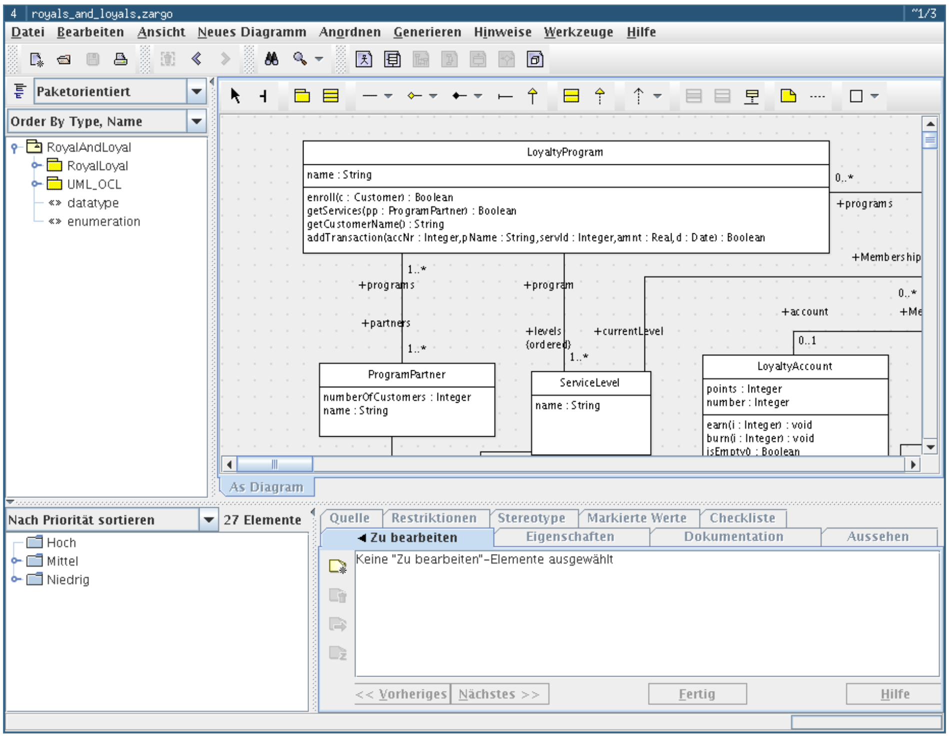


Figure 9.2.: Using ArgoUML for Data Modelling

```
package RoyalLoyal

  context LoyaltyAccount::isEmpty(): Boolean
    pre : true
    post: result = (points = 0)

  context LoyaltyAccount::points : Integer
    init: 0

  context LoyaltyAccount
  inv Transactions: transactions.points
                       ->exists(p : Integer | p = 500 )
endpackage
```

Listing 9.1: A OCL Specification (Excerpt of the Royals and Loyals Model)

1. Modelling the data model (based on the diagrammatic UML notions) using a CASE tool, e. g., ArgoUML (see Figure 9.2). We assume that the reader is familiar with the CASE tool he is using. The result of this process should be an XMI file that can be imported into the Dresden OCL 2 toolkit. Our tool-chain requires the following organization of your model:

   - The complete model should be placed into a top-level UML package. This package is ignored by the Dresden OCL 2 Toolkit.
   - Within the top-level package you can specify arbitrary packages for your models.
   - The top-level package should contain a UML package called `UML_OCL` that contains all OCL datatypes (i.e., the OCL library). All basic datatypes, e.g., Integer, should be taken from this package.

   Please have a look at the examples provided by the HOL-OCL distribution for details.

2. Modelling and type-checking the OCL specification. The Dresden OCL 2 Toolkit uses a simple text format for the OCL specification, e. g., see item 9.1 for an example. The type-checker (included in the "Parser Tool") of the Dresden OCL 2 toolkit can now be used to type-check such a OCL specification against a given data model. This can be either done using a graphical interface (see Figure 9.3) in the following way:

   a) Click on the "Model" tab and load your UML model by clicking on "Load XMI" (either stored as XMI or ArgoUML's native format (`.zargo`).
   b) Click on the "Constraint" tab and load your OCL specification.

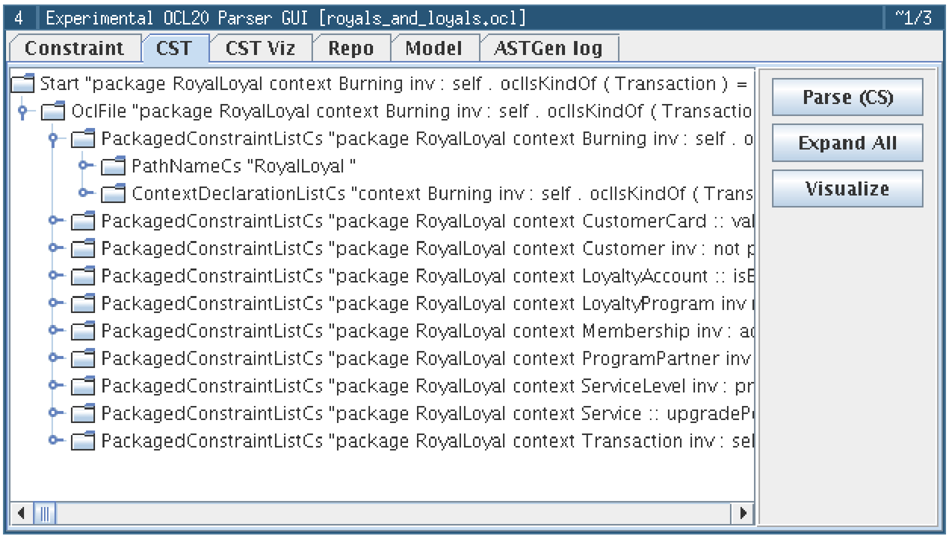


Figure 9.3.: Using the Dresden OCL Parser and Type-checker

c) Click on the "CST" tab and click on "Parse(CS)". This will parse the concrete syntax of your OCL specification, i.e., up to now, no type-checking is involed.

d) If the last step succeeds, i.e., you see a parse tree of the concrete syntax, click on the "ASTGen log" tab and click on "Generate". This will generate type-check your OCL specification.

e) If the type-checking succeeds without errors, you can export (by clicking on "Export XMI") a XMI file conataining both, the UML and OCL part of your model. This XMI file can be loaded into HOL-OCL.

Alternatively, you can type-check your models on the command-line using the following syntax:

```
java -cp ocl20parsertools.jar tudresden.ocl20.core.parser.OCL20CLI \
     --outfile output.xmi input.xmi input.ocl
```

where `input.xmi` is your data model (either in XMI-format or ArgoUML's native format), `input.ocl` is your OCL specification, and `output.xmi` the file where (after successful type-checking) the output is written to.

9.3. Using HOL-OCL: The Verification Phase

In this section we give a brief overview of HOL-OCL related extension of the Isar [53] proof language. We also use a presentation similar to the one in the *Isar Reference*

```
theory royals_and_loyals
imports
  OCL
begin
  load_xmi "royals_and_loyals_ocl.xmi"
end
```

Listing 9.2: A simple HOL-OCL Theory File

Manual [53], e.g. "missing" non-terminals (e.g., ⟨*goalspec*⟩) of our syntax diagrams are defined in [53].

9.3.1. Getting Started

For using HOL-OCL you have to build your Isabelle theories (i.e. test specifications) on top of the theory UML_OCL (plain UML/OCL) or OCL (UML/OCL with extensions) instead of Main. A sample theory is shown in Listing 9.2.

9.3.2. Loading XMI files

A well-typed UML/OCL models can be imported using the **load_xmi** command, e.g.:

```
  load_xmi "royals_and_loyals_ocl.xmi"
```

`load_xmi` – ⟨*filename*⟩

9.3.3. Canonizing Hypotheses

Similar to hypsubst_tac on the HOL level, HOL-OCL provides `ocl_hypsubst_tac` for the elimination of variables A within OCL equality assumptions. The format of these equalities is quite general here, a number of equalities stated implicitly is also handled:

1. $A = t$ and $t = A$ (as in standard hyp_subst_tac),
2. $A\ \tau = t\ \tau$, $t\ \tau = A\ \tau$ (local congruences),
3. $\tau \vDash \not\partial\, A$, $\vDash \not\partial\, A$ (local and global undefinedness),
4. $\tau \vDash \neg\, A$, $\vDash \neg\, A$ (local and global falsities),
5. $\tau \vDash A \triangleq t$, $\tau \vDash t \triangleq A$, $\vDash A \triangleq t$, $\vDash t \triangleq A$ (local and global strong equalities), and
6. $\tau \vDash A \doteq t$, $\tau \vDash A \doteq t \vDash A \doteq t$, $\vDash A \doteq t$ (local and global strict equalities).

The syntax for the command reads as follows:

`ocl_hybsubst_tac` [[- ⟨*goalspec*⟩ -]]

The optional goal specification allows for

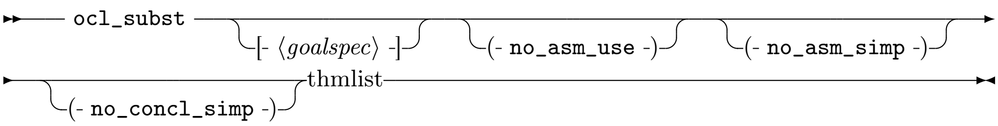
9.3.4. One-Step-Rewriting

The major limitation of canonization as described in the previous section is that it is restricted to variables - on the other hand, the canonization can apply an equality from left to right and from right to left. In contrast, rewriting (actually in this context: narrowing) proceeds only from left to right, but admitting simplification of non-variable terms.

The possible formats of implicit equalities look similarly as the list in `ocl_hypsubst`:

1. $\tau \vDash \not\partial\, lhs$, $\vDash \not\partial\, lhs$ (local and global undefinedness; replaces lhs by $\bot$),
2. $\tau \vDash \partial\, lhs$, $\vDash \partial\, lhs$ (local and global definedness; replaces $\partial\ lhs$ by T),
3. $\tau \vDash \neg\, lhs$, $\vDash \neg\, lhs$ (local and global falsities; replaces $\not\partial\ lhs$ by F),
4. $\tau \vDash lhs \vDash lhs$ (local and global validities; replaces lhs by T),
5. $lhs\ \tau = t\ \tau$ (local congruences),
6. $\vDash lhs \triangleq t$ (local and global strong equalities), and
7. $\tau \vDash lhs \doteq t$, (local and global strict equalities),

The syntax for the command reads as follows:

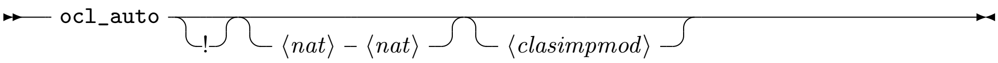


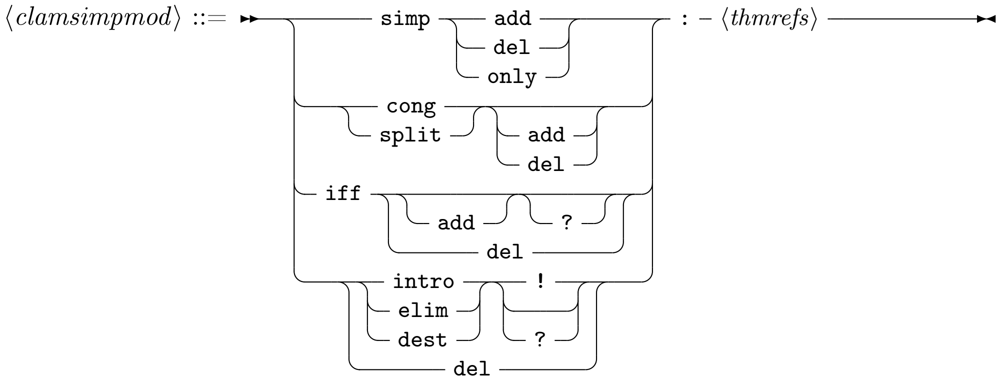
With the attribute specification, the range of the possible rewrite-step can be limited. Specifying `no_asm_use` excludes assumptions in a goal from rewriting, `no_asm_simp` excludes them from being rewritten. By `no_asm_concl`, the rewriting of the conclusion can be prevented. The method accepts a direct specification of a goal to be rewritten, and a list of theorems hat can be used for rewriting (possibly empty).

The method fails if no redex can be found; thus, by using the repetition operator, a simple exhaustive rewriting process can be formulated.

9.3.5. Automated Proof-Procedures

As an equivalent to auto . . .





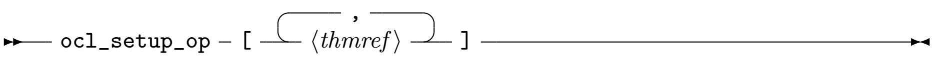
The procedure performs at present no sophisticated ocl-specific machinery.

9.4. Extending HOL-OCL

In this section we explain commands that are usefull for extending HOL-OCL, i.e., they are more thought for developers than for users.

Exploiting theory morphisms: The theory lifting mechanism is used on a “per operator” basis, e.g.:

```
ocl_setup_op [OclIncludes, OclExcludes]
```



Chapter 10.

Case Studies

10.1. Encoding a Stack in UML/OCL

theory *stack*
imports
  OCL
begin

We use the class diagram presented in Fig. 10.1 together with the following OCL formulae shown in Listing 10.1.

load_xmi *stack_ocl.xmi*

An intersting system property would be

lemma $[\![\ \models \ \partial \ s0 \ ]\!] \Longrightarrow \models \ (push\ s0\ e0\ s1) \longrightarrow ((top\ s1\ e1) \wedge (e1 \triangleq e0\ ))$
oops

or

lemma $[\![\ \models \ \partial \ s0 \ ]\!] \Longrightarrow \ size\ (s0) \geq \ size\ (pop\ s0)$
oops

Note that such system properties specified in HOL-OCL allow the use of operations that are not side-effect free (e.g. push)

end

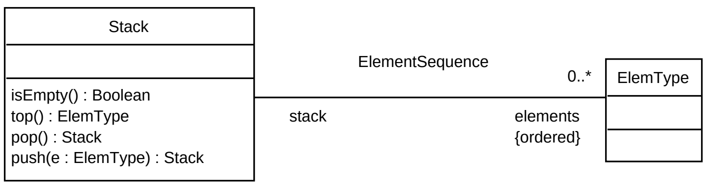


Figure 10.1.: Modelling a Stack: Data Model

```
package Stack

context Stack::pop(): Stack
    pre notEmpty: isEmpty() = false
    post topElementRemoved: top() <> self.top@pre()
    post: elements = elements@pre->subSequence(2,self@pre->size())
    post: elements->size() = elements@pre->size() -1

context Stack::top(): ElemType
    pre   notEmpty: isEmpty() = false
    post: result = elements->first()
--     post: self=self@pre

context Stack::push(e: ElemType): Stack
    post pushedElemTypeIsOnTop: top() = e
    post: elements = elements@pre->prepend(e)
    post: result = self

context Stack::isEmpty(): Boolean
    post: result = (elements->size() <> 0)

endpackage
```

Listing 10.1: Modelling a Stack: OCL specification

Part IV.

Appendix

Appendix A.

The Syntax of OCL

OCL, being advertised with the slogan "Mathematical Foundation, But No Mathematical Symbols" [52], is normally written using a concrete syntax that is inspired by object-oriented programming languages. To give a first impression of this syntax to readers unfamiliar with it, we present a core fragment of OCL using Extended Backus-Naur Form (EBNF) notation (see Tab. A.1). This fragment in particular omits many syntactic variants resulting from naming expressions. This simplified concrete syntax used to denote our examples contains some redundancies: the variant `expr->simpleName` is semantically equivalent with dereferencing `expr.simpleName`, it is a tribute to the OCL convention to distinguish the application of operations on collections such as `X->union(Y)`. In principle, this also holds for the prefix and infix operators. However, the semantics of this applications may be call-by-name or call-by-need; this is handled for each operator individually.

Whereas this textual notation pleases the people coming from object-oriented programming languages, it looks awkward for people coming from the mathematics and formal methods field. Especially for proof work, there seems a need for a compact, mathematical notation. Thus we developed a mathematics-oriented OCL syntax, as an alternative to the programming-language like notation used in the OCL 2.0 standard. For example, compare the textual presentation of the proof rule:

$$\frac{\tau \vDash S\texttt{->includes}(x) \quad \tau \vDash \texttt{not}(P\,x) \quad \mathrm{cp}\,P}{\tau \vDash \big(S\texttt{->forall}\big(x \mid P(x)\big)\texttt{->IsDefined()}} \tag{A.1}$$

to its presentation in mathematical notation:

$$\frac{\tau \vDash x \in S \quad \tau \vDash \neg(P\,x) \quad \mathrm{cp}\,P}{\tau \vDash \partial\big(\forall\, x \in S \,.\, P(x)\big)}\,. \tag{A.2}$$

Clearly, both syntax's have their advantages and disadvantages and therefore we support both of them in HOL-OCL.

In Table A.2 we provide a brief comparison between the different concrete OCL syntax's, namely the syntax as proposed in the OCL standard, our textual notation that tries to follows the standard syntax as close as possible, and finally our new mathematical syntax. The table follows the OCL library presentation from the standard [41, Chapter 11], constructs that are not supported by HOL-OCL are written in a gray typeface, e.g. `o.oclIsInState(s)`.

$$
\begin{array}{rcl}
\mathit{invSpec} & ::= & \mathtt{context}\ \mathit{pathName}\ \mathtt{inv}\ :\ \mathit{expr} \\
\mathit{opSpec} & ::= & \mathtt{context}\ \mathit{operation}\ \mathtt{pre}\ :\ \mathit{expr}\ \mathtt{post}\ :\ \mathit{expr} \\
\mathit{operation} & ::= & [\mathit{pathName}\ ::]\ \mathit{NAME}\ (\ [\mathit{varDecl}\ \{,\ \mathit{varDecl}\}]\ )\ [:\ \mathit{type}] \\
\mathit{varDecl} & ::= & \mathit{NAME}\ [:\ \mathit{type}]\ [=\ \mathit{expr}] \\
\mathit{type} & ::= & \mathit{pathName} \mid \mathit{collKind}\ (\ \mathit{type}\ ) \\
\mathit{expr} & ::= & \mathit{literal} \mid -\mathit{expr} \mid \mathtt{not}\ \mathit{expr} \mid \mathit{expr}\ \mathit{infixOp}\ \mathit{expr} \\
 & & \mid \mathit{pathName}\ [@\mathtt{pre}] \mid \mathit{expr}.\mathit{NAME}\ [@\mathtt{pre}] \mid \mathit{expr}-{>}\mathit{NAME} \\
 & & \mid \mathit{expr}([\{\mathit{expr},\}\mathit{expr}]) \mid \mathit{expr}(\mathit{varDecl}|\mathit{expr}) \\
 & & \mid \mathit{expr}-{>}\mathit{bindOp}(\mathit{varDecl}[;\mathit{varDecl}]|\mathit{expr}) \\
 & & \mid \mathtt{if}\ \mathit{expr}\ \mathtt{then}\ \mathit{expr}\ \mathtt{else}\ \mathit{expr}\ \mathtt{endif} \\
 & & \mid \mathtt{let}\ \mathit{varDecl}\ \{,\ \mathit{varDecl}\}\ \mathtt{in}\ \mathit{expr} \\
\mathit{infixOp} & ::= & * \mid / \mid \mathtt{div} \mid \mathtt{mod} \mid + \mid - \mid < \mid > \mid <= \mid >= \mid = \mid <> \\
 & & \mid \mathtt{and} \mid \mathtt{or} \mid \mathtt{xor} \mid \mathtt{implies} \\
\mathit{bindOp} & ::= & \mathtt{iterate} \mid \mathtt{forall} \mid \mathtt{exists} \\
\mathit{literal} & ::= & \mathtt{Integer} \mid \mathtt{Real} \mid \mathtt{String} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{OclUndefined} \\
 & & \mid \mathit{collKind}\{\{\mathit{collLitPart},\}\mathit{collLitPart}\} \\
\mathit{collKind} & ::= & \mathtt{Set} \mid \mathtt{Bag} \mid \mathtt{Sequence} \mid \mathtt{Collection} \mid \mathtt{OrderedSet} \\
\mathit{collLitPart} & ::= & \mathit{expr} \mid \mathit{expr}..\mathit{expr} \\
\mathit{pathName} & ::= & [\mathit{pathName}::]\mathit{NAME}
\end{array}
$$

Table A.1.: Formal Grammar of OCL (fragment)

Table A.2.: Comparison of different concrete syntax variants for OCL

	OCL (standard)	mathematical HOL-OCL	textual HOL-OCL
OclAny	`x = y`	$x \doteq y$	x `==` y
	`x <> y`	$x \dot{\neq} y$	x `<>` y
		$x \triangleq y$	x `=` y
		$x \dot{\simeq} y$	$x \sim== y$
		$x \overset{\triangle}{\simeq} y$	$x \sim= y$
		$x \dot{\cong} y$	$x \sim==\sim y$
		$x \overset{\triangle}{\cong} y$	$x \sim=\sim y$
	`o.oclIsNew()`		
	`o.oclIsUndefined()`	$\not\partial\, o$	o`->oclIsUndefined()`
	`o.oclAsType(t)`	o`->AsType` (t)	o`->AsType`(t)
	`o.oclIsType(t)`	o`->IsType` (t)	o`->IsType`(t)
	`o.oclIsKindOf(t)`	o`->IsKindOf` (t)	o`->IsKindOf`(t)
	`o.oclIsInState(s)`		
	`o.allInstances()`	o`->AllInstances()`	o`->AllInstances()`
OclMessage	`o.hasReturned()`		
	`o.result()`		
	`o.isSignalSent()`		
	`o.isOperationCall()`		
OclVoid	`OclUndefined`	$\perp$	`OclUndefined`
	`o.oclIsUndefined()`	$\not\partial\, o$	o`->oclIsUndefined()`
		$\partial\, o$	o`->IsDefined()`
Real	`x + y`	$x + y$	$x + y$
	`x - y`	$x - y$	$x - y$
	`x * y`	$x * y$	$x * y$
	`-x`	$-x$	$-x$
	`x / y`	x/y	x/y
	`x.abs()`	$\lvert s \rvert$	s`->abs()`
	`x.floor()`	$\lfloor s \rfloor$	s`->floor()`
	`x.round()`	$\lceil s \rfloor$	s`->round()`
	`x.max(y)`	$\max(x, y)$	x`->max`(y)
	`x.min(y)`	$\min(x, y)$	x`->min`(y)
	`x < y`	$x < y$	$x < y$
	`x > y`	$x > y$	$x > y$
	`x <= y`	$x \leq y$	$x <= y$
	`x >= y`	$x \geq y$	$x >= y$
Integer	`x - y`	$x - y$	$x - y$
	`x + y`	$x + y$	$x + y$
	`-x`	$-x$	$-x$
	`x * y`	$x * y$	$x * y$
	`x / y`	x/y	x/y
	`x.abs(y)`	$\lvert s \rvert$	s`->abs()`

Continued on next page

	OCL	mathematical HOL-OCL	textual HOL-OCL
	`x.div(y)`	x div y	x->div(y)
	`x.mod(y)`	x mod y	x->mod(y)
	`x.max(y)`	max(x, y)	x->max(y)
	`x.min(y)`	min(x, y)	x->min(y)
String	`s.size()`	$\lVert s \rVert$	s->size()
	`s.concat(z)`	$s \frown z$	s->concat(z)
	`s.substring(i,j)`	s->substring(i, j)	s->substring(i, j)
	`s.toInteger()`	s->toInteger()	s->toInteger()
	`s.toReal()`	s->toReal()	s->toReal()
	`s.toUpper()`	s->toUpper()	s->toUpper()
	`s.toLower()`	s->toLowert()	s->toLowert()
Boolean		$\models p$	Valid p
		$\tau \models p$	τ Valid p
	`true`	T	true
	`false`	F	false
	`x or y`	$x \vee y$	x or y
	`x xor y`	$x \oplus y$	x xor y
	`x and y`	$x \wedge y$	x and y
	`not x`	$\neg x$	not x
	`x implies y`	$x \longrightarrow y$	x implies y
		$x \xrightarrow{1} y$	x implies1 y
		$x \xrightarrow{2} y$	x implies2 y
		$x \mathbin{\dot\vee} y$	x sor y
		$x \mathbin{\dot\oplus} y$	x sxor y
		$x \mathbin{\dot\wedge} y$	x sand y
		$x \mathbin{\dot\longrightarrow} y$	x simplies y
	`if c then x else y endif`	if c then x else y endif	if c then x else y endif
Collection	`X->size()`	$\lVert X \rVert$	X->size()
	`X->includes(y)`	$y \in X$	X->includes(y)
	`X->excludes(y)`	$y \notin X$	X->excludes(y)
	`X->count(y)`	X->count()	X->count()
	`X->includesAll(Y)`	$X \subseteq Y$	X->includesAll(Y)
	`X->excludesAll(Y)`	$X \not\subseteq Y$	X->excludesAll(Y)
	`X->isEmpty()`	$X \doteq \emptyset$	X->isEmpty()
	`X->notEmpty()`	$X \mathrel{\dot{\neq}} \emptyset$	X->notEmpty()
	`X->sum()`	X->sum()	X->sum()
	`X->product(Y)`	$X \times Y$	X->product(Y)
	`X->exists(e:T|P(e))`	$\exists\, e \in X. P(e)$	X->exists($e : T \mid P(e)$)
	`X->forAll(e:T|P(e))`	$\forall\, e \in X. P(e)$	X->forall($e : T \mid P(e)$)
	`X->isUnique(e:T|P(e))`	X->isUnique ($e : T \mid P(e)$)	X->isUnique($e : T \mid P(e)$)
	`X->any(e:T|P(e))`	X->any($e : T \mid P(e)$)	X->any($e : T \mid P(e)$)
	`X->one(e:T|P(e))`	X->one($e : T \mid P(e)$)	X->one($e : T \mid P(e)$)
	`X->collect(e:T|P(e))`	$\{\!\mid e \in X \mid P(e) \mid\!\}$	X->collect($e : T \mid P(e)$)

Continued on next page

<table>
<tr><th></th><th>OCL</th><th>mathematical HOL-OCL</th><th>textual HOL-OCL</th></tr>
<tr><td rowspan="19">Set</td><td><code>Set{}</code></td><td>$\emptyset$</td><td><code>{}</code></td></tr>
<tr><td><code>X->union(Y)</code></td><td>$X \cup Y$</td><td>X<code>->union</code>(Y)</td></tr>
<tr><td><code>X = Y</code></td><td>$X \doteq Y$</td><td>$X == Y$</td></tr>
<tr><td><code>X->intersection(Y)</code></td><td>$X\cap$</td><td>X<code>->intersection</code></td></tr>
<tr><td><code>X->complement(Y)</code></td><td>X^{-1}</td><td>X<code>->complement()</code></td></tr>
<tr><td><code>X - Y</code></td><td>$X - Y$</td><td>$X - Y$</td></tr>
<tr><td><code>X->including(y)</code></td><td>$y \in X$</td><td>X<code>->includes</code>(y)</td></tr>
<tr><td><code>X->excluding(y)</code></td><td>$y \notin X$</td><td>X<code>->excludes</code>(y)</td></tr>
<tr><td><code>X->symmetricDifference(Y)</code></td><td>$X \ominus Y$</td><td>X<code>->symmetricDifference</code>(Y)</td></tr>
<tr><td><code>X->count(y)</code></td><td>X<code>->count</code>(y)</td><td>X<code>->count</code>(y)</td></tr>
<tr><td><code>X->flatten()</code></td><td>⟦X⟧</td><td>X<code>->flatten</code>$()$</td></tr>
<tr><td><code>X->asSet()</code></td><td>X<code>->asSet()</code></td><td>X<code>->asSet()</code></td></tr>
<tr><td><code>X->asOrderedSet()</code></td><td>X<code>->asOrderedSet()</code></td><td>X<code>->asOrderedSet()</code></td></tr>
<tr><td><code>X->asSequence()</code></td><td>X<code>->asSequence()</code></td><td>X<code>->asSequence()</code></td></tr>
<tr><td><code>X->asBag()</code></td><td>X<code>->asBag()</code></td><td>X<code>->asBag()</code></td></tr>
<tr><td><code>X->select(e:T|P(e))</code></td><td>⦇$e \in X \mid P(e)$⦈</td><td>X<code>->select</code>$(e : T \mid P(e))$</td></tr>
<tr><td><code>X->reject(e:T|P(e))</code></td><td>⦈$e \in X \mid P(e)$⦇</td><td>X<code>->reject</code>$(e : T \mid P(e))$</td></tr>
<tr><td><code>X->collectNested(e:T|P(e))</code></td><td>{{|$e \in X \mid P(e)$|}}</td><td>X<code>->collectNested</code>$(e : T \mid P(e))$</td></tr>
<tr><td><code>X->sortedBy(e:T|P(e))</code></td><td>X<code>->sortedBy</code> $(e : T \mid P(e))$</td><td>X<code>->sortedBy</code>$(e : T \mid P(e))$</td></tr>
<tr><td rowspan="9">OrderedSet</td><td><code>OrderedSet{}</code></td><td>$\langle\rangle$</td><td><code>OrderedSet{}</code></td></tr>
<tr><td><code>X->append(y)</code></td><td>$X :: y$</td><td>X<code>->append</code>(y)</td></tr>
<tr><td><code>X->prepend(y)</code></td><td>$y : X$</td><td>X<code>->prepend</code>(y)</td></tr>
<tr><td><code>X->insertAt(i,y)</code></td><td>X<code>->insertAt</code>(i, y)</td><td>X<code>->insertAt</code>(i, y)</td></tr>
<tr><td><code>X->subOrderedSet(i,j)</code></td><td>X<code>->subOrderedSet</code>(i, j)</td><td>X<code>->subOrderedSet</code>(i, j)</td></tr>
<tr><td><code>X->at(i)</code></td><td>$\natural i\ X$</td><td>X<code>->at</code>(i)</td></tr>
<tr><td><code>X->indexOf(y)</code></td><td>$X\ \natural?(y)$</td><td>X<code>->indexOf</code>(y)</td></tr>
<tr><td><code>X->first()</code></td><td>$\natural 1\ X$</td><td>X<code>->first()</code></td></tr>
<tr><td><code>X->last()</code></td><td>$\natural\$\ X$</td><td>$X$<code>->last()</code></td></tr>
<tr><td rowspan="16">Bag</td><td><code>Bag{}</code></td><td>⟅⟆</td><td><code>Bag{}</code></td></tr>
<tr><td><code>X = Y</code></td><td>$X = Y$</td><td>$X = Y$</td></tr>
<tr><td><code>X->union(Y)</code></td><td>$X \cup Y$</td><td>X<code>->union</code>(Y)</td></tr>
<tr><td><code>X->intersection(Y)</code></td><td>X^{-1}</td><td>X<code>->complement()</code></td></tr>
<tr><td><code>X->including(y)</code></td><td>$y \in X$</td><td>X<code>->includes</code>(y)</td></tr>
<tr><td><code>X->excluding(y)</code></td><td>$y \notin X$</td><td>X<code>->excludes</code>(y)</td></tr>
<tr><td><code>X->count(y)</code></td><td>X<code>->count</code>(y)</td><td>X<code>->count</code>(y)</td></tr>
<tr><td><code>X->flatten()</code></td><td>⟦X⟧</td><td>X<code>->flatten</code>$()$</td></tr>
<tr><td><code>X->asBag()</code></td><td>X<code>->asBag()</code></td><td>X<code>->asBag()</code></td></tr>
<tr><td><code>X->asSequence()</code></td><td>X<code>->asSequence()</code></td><td>X<code>->asSequence()</code></td></tr>
<tr><td><code>X->asSet()</code></td><td>X<code>->asSet()</code></td><td>X<code>->asSet()</code></td></tr>
<tr><td><code>X->asOrderedSet()</code></td><td>X<code>->asOrderedSet()</code></td><td>X<code>->asOrderedSet()</code></td></tr>
<tr><td><code>X->select(e:T|P(e))</code></td><td>⦇$e \in X \mid P(e)$⦈</td><td>X<code>->select</code>$(e : T \mid P(e))$</td></tr>
<tr><td><code>X->reject(e:T|P(e))</code></td><td>⦈$e \in X \mid P(e)$⦇</td><td>X<code>->reject</code>$(e : T \mid P(e))$</td></tr>
<tr><td><code>X->collectNested(e:T|P(e))</code></td><td>{{|$e \in X \mid P(e)$|}}</td><td>X<code>->collectNested</code>$(e : T \mid P(e))$</td></tr>
<tr><td><code>X->sortedBy(e:T|P(e))</code></td><td>X<code>->sortedBy</code> $(e : T \mid P(e))$</td><td>X<code>->sortedBy</code>$(e : T \mid P(e))$</td></tr>
</table>

Continued on next page

	OCL	mathematical HOL-OCL	textual HOL-OCL
Sequence	`Sequence{}`	$[]$	`[]`
	`X->count()`	$X\,\texttt{->count}(y)$	$X\,\texttt{->count}(y)$
	`X = Y`	$X = Y$	$X = Y$
	`X->union(Y)`	$X \cup Y$	$X\,\texttt{->union}(Y)$
	`X->flatten()`	$[\![X]\!]$	$X\,\texttt{->flatten}()$
	`X->append(y)`	$X :: y$	$X\,\texttt{->append}\,y$
	`X->prepend(y)`	$y : X$	$X\,\texttt{->prepend}\,y$
	`X->insertAt(i,y)`	$X\,\texttt{->insertAt}(i, y)$	$X\,\texttt{->insertAt}(i, y)$
	`X->subSequence(i,j)`	$X\,\texttt{->subSequence}(i, j)$	$X\,\texttt{->subSequence}(i, j)$
	`X->at(i)`	$\natural i\ X$	$X\,\texttt{->at}(i)$
	`X->indexOf(y)`	$X\ \natural?(y)$	$X\,\texttt{->indexOf}(y)$
	`X->first()`	$\natural 1\ X$	$X\,\texttt{->first()}$
	`X->last()`	$\natural\$\ X$	$X\,\texttt{->last()}$
	`X->including(y)`	$y \in X$	$X\,\texttt{->includes}(y)$
	`X->excluding(y)`	$y \notin X$	$X\,\texttt{->excludes}(y)$
	`X->asBag()`	$X\,\texttt{->asBag()}$	$X\,\texttt{->asBag()}$
	`X->asSequence()`	$X\,\texttt{->asSequence()}$	$X\,\texttt{->asSequence()}$
	`X->asSet()`	$X\,\texttt{->asSet()}$	$X\,\texttt{->asSet()}$
	`X->asOrderedSet()`	$X\,\texttt{->asOrderedSet()}$	$X\,\texttt{->asOrderedSet()}$
	`X->select(e:T|P(e))`	$(\!\|e \in X \mid P(e)\|\!)$	$X\,\texttt{->select}(e : T \mid P(e))$
	`X->reject(e:T|P(e))`	$\|\!)e \in X \mid P(e)(\!\|$	$X\,\texttt{->reject}(e : T \mid P(e))$
	`X->collectNested(e:T|P(e))`	$\{\{\!\|e \in X \mid P(e)\|\!\}\}$	$X\,\texttt{->collectNested}(e : T \mid P(e))$
	`X->sortedBy(e:T|P(e))`	$X\,\texttt{->sortedBy}\,(e : T \mid P(e))$	$X\,\texttt{->sortedBy}(e : T \mid P(e))$
	`let e=x in P(s) end`	$\mathrm{let}\ e = x\ \mathrm{in}\ P(s)\ \mathrm{end}$	$\texttt{let}\ e = x\ \texttt{then}\ P(s)\ \texttt{end}$
		$\overset{x}{\llcorner\ \lrcorner}$ $\underset{x}{\ulcorner\ \urcorner}$	

Appendix B.

Isabelle Theories

B.1. Introduction

In this chapter we present the Isabelle theories of HOL-OCL. The main dependencies are shown in Fig. B.1 on the next page. HOL-OCL is build on top of Isabelle/HOL.[1] This chapter was automatically generated by Isabelle, i.e., it is a *formal* document: all definitions and theorems presented here were checked and formally proven by Isabelle. Nevertheless, we took the privilege of omitting theory some parts, in particular part's concerning the setup of the styntax engine, the interface to Isar the implementation of several tactics, the lifter, and the datatype package. Thus, in this chapter we present the semantic foundation of HOL-OCL. If you are also interested in the technical details of the implementation, please read the source code which is included in the HOL-OCL distribution.

B.1.1. HOL-OCL Configurations

Note, HOL-OCL can be built in different configurations, e.g., one can choose between non-referential and referntial universes and between smashed and unsmashed collection types:

Universe: One issue to be raised here is the semantics of equality; are two objects equal only if their object identifier is equal or are two objects equal if their values are equal? The OCL semantics is not specific here since equality is defined as equality over values [41, Sec. A.2.2], and since objects are values, but object identifiers are not distinguished from object values [41, Definition A.10]. However, since many object-oriented programming languages are centered around referential equality (which gave the motivation to opt for the latter in [7]), HOL-OCL can be configured such that the above definition leads to referential equality.

- **non-referential:** here objects are just identified with their values, i.e., two objects are equal if and only if all their attributes are equal.
- **referential universe:** here a referential equality is present, which is is the strong equality for "boxed types" (that is Real, Boolean, String) and an equality on the reference to a value.

[1] More precise, HOL-OCL is build on top of Isabelle/HOL-Complex but only to support real numbers. If you dont't need them, HOL-OCL should be build just fine on top of a plain Isabelle/HOL.

Figure B.1.: Session Graph

This manual describes HOL-OCL with referential universes. We recommend the use of *referential* universes.

This option has effects on the theory `OCL_OclAny_type`.

Collections: OCL [41, Sec. A.2.5.2] allows for collections to include $\perp$, i.e. the constructor of sets and the membership tests are non-strict. This has several undesired consequences for executability and proof support. Thus we provide two collection libraries:

smashed collections: smashing data-structures is a key-concept in denotational semantics [35, 54]. For example, pairs are smashed if $(a, \perp)$ is identified with $\perp$ as in e.g. Java or SML, or, with respect to sets, $\{a, \perp\} = \perp$.

unsmashed collections: In this configuration all collection types can contain the elment $\perp$.

This manual describes HOL-OCL with smashed collections. We recommend the use of *smashed* collections.

This option has effects on the following theories: `OCL_Bag.thy`, `OCL_OrderedSet.thy`, `OCL_Set.thy`, `OCL_Bag_type.thy`, `OCL_OrderedSet_type.thy`, `OCL_Set_type.thy`, `OCL_CharactersticSet.thy`, `OCL_Sequence.thy`, `OCL_Collection_requirements.thy`, and `OCL_Sequence_type.thy`.

The “recommended configuration” is more developed and contains more deductive support.

B.1.2. Notational Remarks

For presentational reasons, we used in the previous chapters for a few constructs a “Formal Syntax” that looks somewhat more “mathematical” than the syntax of Isabelle. To help readers that are not familiar with the syntax of Isabelle/HOL we compare the (already introduced) “Formal Syntax” with the Isabelle syntax used in this chapter in Table B.1.

B.2. Overview

The remainder of this chapter are the Isabelle theories using the LaTeX-based presentation facilities of Isabelle. We structured the theory presentation in this chapter into the following subsections:

Foundations: In this section, we introduce the foundations of HOL-OCL, namely the concept of *lifting* and the basic datatype definitions.

Library: In this section, we introduce the OCL datatypes presented in [41, Chapter 11] and prove basic properties over them. Especially, we prove that our definitions fulfill the requirements presented in the OCL standard [41, Chapter 11].

Formal Syntax	**Isabelle Syntax**
$\dfrac{A_1 \quad \dots \quad A_n}{C}$	$[\![A_1; \dots; A_n]\!] \Longrightarrow C$
$\mathrm{sumCase}(f, g, x) = \begin{cases} f(k) & \text{if } x = \mathrm{Inl}\, k \\ g(k) & \text{if } x = \mathrm{Inr}\, k \end{cases}$	$\mathrm{case}\, x \,\mathrm{of}\, \mathrm{Inl}\, k \Rightarrow f(k) \vert\, \mathrm{Inr}\, k \Rightarrow g(k)$
$\mathrm{upCase}(f, c, x) = \begin{cases} c & \text{if } x = \bot\,; \\ f(k) & \text{if } x = \llcorner k \lrcorner \end{cases}$	$\mathrm{case}\, x \,\mathrm{of} \llcorner k \lrcorner \Rightarrow f(k) \vert \bot \Rightarrow c\,.$
$V_\tau(\alpha)$	$(\tau, \alpha)\, VAL$
$\mathrm{Sem}[\![(\_)]\!]$	`semfun _`

Table B.1.: Comparison of the formal syntax and the Isabelle syntax

State: In this section, the concept of state and constructive concepts like proof calculi are presented.

Requirements: In this section, the proof (or disproof) of the requirements of [41, Chapter 11].

OCL: This section just contains the final OCL theory that should be used as basis for case studies, i. e., it it the HOL-OCL main theory.

B.3. Foundations

B.3.1. The Theory of Lifting and its Combinators

theory *Lifting*
imports
Main
begin

Besides the infrastructure for handling undefinedness, this theory also provides the basis for data type embedding smashing and context lifting including the generic core of theorems.

The main purpose of this theory is to provide a generic theory of undefinedness. The Isabelle standard mechanism for such a generic data type is the class mechanism. The standard method is to declare a class to which all related operations are associated.

A Generic Theory of Undefinedness

Since the very first OCL publictions [39] explicit undefinedness is part of the language, both for the logic and the basic values, e. g., [39, 7.4.9 Undefined Values] states (this is similary stated in [41, page 15]):

Whenever an OCL expressionis beeing evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined.
There are two exceptions to this for the Boolean operators

- True OR-ed with anything is True
- False AND-ed with anything is False

The above two rules are valid irrespective of the order of the arguments and the above rules are valid wether or not the value of the other sub-expression is known.
(OCL Specification [39], page 7-11)

This requirement postulates the strictness of all operations (except for the logic) and rules out a modeling of undefinedness via Hilbert-Operators and underspecification (e.g., as done for Z [26]in Isabelle/HOL-Z [11]).

classes *bot0* $\subseteq$ *ord*

All bottom types posses a constant $\perp$

consts $UU :: {}'\alpha{::}bot0$
syntax $UU :: {}'\alpha{::}bot0 \quad (\perp)$

axclass $bot \subseteq bot0$
$nonEmpty : \exists\ x.\ x \neq \perp$

constdefs
$DEF \quad :: {}'\alpha{::}bot \Rightarrow bool$
$DEF\ x \quad \equiv (x \neq \perp)$

$isStrict \quad :: ({}'\alpha{::}bot \Rightarrow {}'\beta{::}bot) \Rightarrow bool$
$isStrict\ f \quad \equiv (f \perp = \perp)$

$strictify \quad :: ({}'\alpha{::}bot \Rightarrow {}'\beta{::}bot) \Rightarrow {}'\alpha \Rightarrow {}'\beta$
$strictify\ f\ x \equiv if\ x{=}\perp\ then\ \perp\ else\ f\ x$

$smash \quad :: [[{}'\beta{::}bot,\ {}'\alpha{::}bot] \Rightarrow bool,\ {}'\alpha] \Rightarrow {}'\alpha$
$smash\ f\ X \quad \equiv if\ f \perp X\ then\ \perp\ else\ X$

Semantic Constructions: Liftings, Functions, Products

We introduce now the lifting construction (see Winskel [54, p.131]) by a type constructor defined as free data type:

datatype ${}'\alpha\ up = lift\ {}'\alpha \mid down$

constdefs
$drop :: {}'\alpha\ up \Rightarrow {}'\alpha$

$drop\ x\ \equiv case\ x\ of\ lift\ v \Rightarrow v \mid down \Rightarrow \epsilon\ \ x\ .\ True$

syntax
$@lift\ \ :: {}'\alpha \Rightarrow {}'\alpha\ up\ \ (|.(\_).|)$
$Lifting.drop :: {}'\alpha\ up \Rightarrow {}'\alpha\ \ (|\hat{}(\_)\hat{}|)$

syntax (*xsymbols*)
$@lift\ \ :: {}'\alpha \Rightarrow {}'\alpha\ up\ \ (\llcorner\_\lrcorner)$
$Lifting.drop\ :: {}'\alpha\ up \Rightarrow {}'\alpha\ \ (\ulcorner\_\urcorner)$

translations
$|.a.| \rightleftharpoons lift\ a$
$|\hat{}a\hat{}| \rightleftharpoons drop\ a$

instance *up* :: (*zero*) *zero*
by *intro_classes*
instance *up* :: (*one*) *one*
by *intro_classes*
instance *up* :: (*plus*) *plus*
by *intro_classes*
instance *up* :: (*minus*) *minus*
by *intro_classes*
instance *up* :: (*times*) *times*
by *intro_classes*
instance *up* :: (*inverse*) *inverse*
by *intro_classes*

Semantic Constructions: Liftings of Type Constructors

The class is then propagated across lifting, function space and cartesian products.

instance *up* :: (*type*) *ord*
by *intro_classes*
arities *up* :: (*type*) *bot*
instance *fun* :: (*type*,*ord*) *ord*
by *intro_classes*
arities *fun* :: (*type*,*bot*) *bot*
instance $*$ :: (*ord*,*ord*) *ord*
by *intro_classes*
instance $+$ :: (*ord*,*ord*) *ord*
by *intro_classes*

arities $*$ :: (*bot*,*bot*) *bot*
arities $+$ :: (*bot*,*bot*) *bot*

defs
UU_up_def[*simp*]: $\perp \equiv down$

$UU\_fun\_def[simp]$: $\perp \equiv (\lambda x.\ \perp)$
$UU\_pair\_def[simp]$: $\perp \equiv (\perp, \perp)$
$UU\_sum\_def[simp]$: $\perp \equiv (Inl\ \perp)$

instance *up* :: (*type*) *bot*
by *intro_classes*
instance *fun* :: (*type*,*bot*) *bot*
by *intro_classes*
instance $*$ :: (*bot*,*bot*) *bot*
by *intro_classes*
instance $+$ :: (*bot*,*bot*) *bot*
by *intro_classes*

Semantic Constructions Dealing with Context Lifting

types $('\tau,\ 'b)\ VAL = '\tau \Rightarrow 'b$

instance *fun*::(*type*,*zero*) *zero*
by *intro_classes*
instance *fun*::(*type*,*one*) *one*
by *intro_classes*
instance *fun*:: (*type*,*plus*) *plus*
by *intro_classes*
instance *fun*:: (*type*,*minus*) *minus*
by *intro_classes*
instance *fun*:: (*type*,*times*) *times*
by *intro_classes*
instance *fun*:: (*type*,*inverse*) *inverse*
by *intro_classes*
instance *fun*:: (*type*,*ord*) *ord*
by *intro_classes*

constdefs
$lift0\ ::\ '\alpha \Rightarrow ('\tau,'\alpha)\ VAL$
$lift0 \equiv \lambda c.\ \lambda s.\ c$
$lift1\ ::\ ('\alpha \Rightarrow '\beta) \Rightarrow ('\tau,\ '\alpha)\ VAL\ \Rightarrow ('\tau,\ '\beta)\ VAL$
$lift1 \equiv (\lambda f\ X\ \tau\ .\ f\ (X\ \tau))$
$lift2\ ::\ (['\alpha,'\beta]\Rightarrow'\gamma) \Rightarrow [('\tau,'\alpha)\ VAL,('\tau,'\beta)\ VAL]\Rightarrow('\tau,'\gamma)\ VAL$
$lift2 \equiv (\lambda f\ X\ Y\ \tau.\ f\ (X\ \tau)(Y\ \tau))$
$lift3\ ::\ (['\alpha,'\beta,\ '\gamma]\Rightarrow'\delta) \Rightarrow [('\tau,'\alpha)\ VAL,('\tau,'\beta)\ VAL,\ ('\tau,'\gamma)\ VAL]\Rightarrow('\tau,'\delta)\ VAL$
$lift3 \equiv (\lambda f\ X\ Y\ Z\ \tau.\ f\ (X\ \tau)(Y\ \tau)(Z\ \tau))$

$cp\ ::\ ([('\tau,'\alpha)\ VAL,\ '\tau] \Rightarrow '\beta) \Rightarrow bool$
$cp(P) \equiv (\exists\ f.\ \forall\ X\ \tau.\ P\ X\ \tau = f\ (X\ \tau)\ \tau)$

syntax

```
lift0 :: 'α ⇒ ('τ,'α) VAL
         (lift₀)
lift1 :: ('α ⇒ 'β) ⇒ ('s, 'α) VAL  ⇒ ('s, 'β) VAL
         (lift₁)
lift2 :: (['α,'β]⇒'γ) ⇒ [('τ,'α) VAL,('τ,'β) VAL]⇒('τ,'γ) VAL
         (lift₂)
lift3 :: (['α,'β, 'γ]⇒'δ) ⇒ [('τ,'α) VAL,('τ,'β) VAL, ('τ,'γ) VAL]⇒('τ,'δ) VAL
         (lift₃)
```

A Generic Theory of Undefinedness, Strictness, and Smashing

```
lemma not_DEF_UU [simp]: ¬ DEF(⊥)
  by (simp (no_asm) add: DEF_def)

lemma not_DEF_X: (¬ DEF X) = (X = ⊥)
  by (simp (no_asm) add: DEF_def)

lemma exists_DEF: ∃ x. DEF(x)
  by (simp add: DEF_def nonEmpty)


lemma isStrict_strictify [simp]: isStrict(strictify f)
  by (simp add: isStrict_def strictify_def)


lemma strict2a_UU [simp]: strictify f ⊥ = ⊥
  by (simp (no_asm) add: strictify_def)

lemma strict2b_UU [simp]: strictify f ⊥ X = ⊥
  by (simp (no_asm) add: strictify_def UU_fun_def)
```

Strictness versus Definedness

```
lemma DEF_strictify_DEF_args2 :
  DEF (strictify (λx. strictify (f x)) X Y) ⟹ DEF X ∧ DEF Y
  apply (simp add: strictify_def DEF_def)
  apply (case_tac X=⊥)
  apply auto
  done


lemma DEF_strictify_DEF_fun :
  ⟦ ⋀X. DEF(f X) ; DEF X ⟧ ⟹ DEF(strictify f X)
  by (simp add: strictify_def DEF_def )


lemma DEF_strictify_DEF_args :
  DEF(strictify f X) ⟹  DEF f ∧ DEF X
  by (simp add: strictify_def DEF_def, auto)
```

lemma *isStrict_compose* :
$\llbracket$ *isStrict f*; *isStrict g* $\rrbracket \Longrightarrow$ *isStrict* (*f o g*)
by (*simp add*: *isStrict_def o_def*)

lemma *smash_strict* [*simp*]: *smash f* $\bot = \bot$
by (*simp* (*no_asm*) *add*: *smash_def*)

lemma *smashed_sets_nonempty* [*simp*]: $\bot$: $\{X.$ *smash f X* $= X\}$
by (*simp* (*no_asm*))

This lemmas are useful for proofs of non-emptyness of type-definition based on smashed collection types.

A Theory of Undefinedness and Strictness in Lifted Types

lemma *not_down_exists_lift* : $x \neq$ *down* $= (\exists\, y.\ x = \llcorner y \lrcorner)$
by (*induct_tac x*,*auto*)

lemma *not_down_exists_lift2* : $x \neq \bot = (\exists\ y.\ x = \llcorner y \lrcorner)$
by (*induct_tac x*, *auto*)

lemma *drop_lift* [*simp*]: $\ulcorner \llcorner f \lrcorner \urcorner = f$
by (*simp add*: *drop_def UU_up_def*)

lemma *drop_down* [*simp*]: $\ulcorner \bot \urcorner = (\epsilon\ x.\ True)$
by (*simp add*: *drop_def*)

lemma *drop_down2* [*simp*]: $\ulcorner down \urcorner = (\epsilon\ x.\ True)$
by (*simp add*: *drop_def*)

lemma *not_DEF_down* [*simp*]: $\neg$*DEF*(*down*)
by (*simp* (*no_asm*) *add*: *DEF_def*)

lemma *DEF_lift* [*simp*]: *DEF*($\llcorner x \lrcorner$)
by (*simp* (*no_asm*) *add*: *DEF_def*)

lemma *DEF_X_up* : *DEF*(X::$'\alpha$ *up*) $= (\exists\ x.\ X = \llcorner x \lrcorner)$
by (*simp add*: *not_down_exists_lift DEF_def*)

lemma *not_DEF_X_up* : ($\neg$*DEF*(X::$'\alpha$ *up*)) $= (X = \bot)$
by (*simp* (*no_asm*) *add*: *DEF_def*)

lemma *DEF_fun_lift* [*simp*]: *DEF*($\lambda x.\ \llcorner f\ x \lrcorner$)

apply (*simp* (*no_asm*) *add*: *DEF_def*)
apply (*rule notI*)
apply (*drule fun_cong*)
apply *simp*
done

lemma *DEF_fun_fun_lift* [*simp*]: $DEF(\lambda x\ y.\ \llcorner f\ x\ y \lrcorner)$
apply (*simp* (*no_asm*) *add*: *DEF_def*)
apply (*rule notI*)
apply (*drule fun_cong*)
apply (*drule fun_cong*)
apply *simp*
done

lemma *lift_defined* : $\perp \notin X \longrightarrow \llcorner X \lrcorner \neq \perp$
by(*auto*)

A Generic Theory of Strictness

lemma *strict2c_UU* [*simp*]: *strictify f down* = *down*
by (*simp* (*no_asm*) *add*: *strictify_def*)

lemma *strict2d_UU* [*simp*]: *strictify f down X* = *down*
by (*simp* (*no_asm*) *add*: *strictify_def*)

lemma *strict2e_UU* [*simp*]: $strictify\ (\lambda x.\ strictify\ (f\ x))\ (X{::}'\alpha{::}bot)\ \perp = \perp$
by (*simp add*: *strictify_def*)

lemma *strict2f_UU* [*simp*]: $strictify\ (\lambda x.\ strictify\ (f\ x))\ (X{::}'\alpha{::}bot)\ down = down$
by (*simp add*: *strictify_def*)

lemma *strict2_DEF* [*simp*]: $DEF\ X \Longrightarrow strictify\ f\ X = f\ X$
by (*simp add*: *strictify_def*, *auto*)

lemma *strict3_DEF* [*simp*]:
$[\![\ DEF\ X;\ DEF\ Y\ ]\!] \Longrightarrow strictify(\lambda\ X.\ strictify\ (f\ X))\ X\ Y = f\ X\ Y$
by (*simp add*: *strictify_def*, *auto*)

Generic Undefinedness-Reduction Rules for 2-Lifted Strict Operations

lemma *lift1b_undef* [*simp*]: $lift_1\ (strictify\ f)\ \perp = \perp$
by (*simp add*: *lift1_def*)

lemma *lift2_undef1a* [*simp*]: $lift_2\ (strictify(\lambda x.\ strictify(f\ x)))\ \perp\ (X{::}('a{\Rightarrow}('b{::}bot))) = \perp$
by (*simp add*: *lift1_def lift2_def*)

lemma *lift2_undef2a* [*simp*]: $lift_2\ (strictify(\lambda x.\ strictify(f\ x)))\ (X{::}('\tau{\Rightarrow}('b{::}bot)))\ \perp = \perp$

by (*simp add: lift1_def lift2_def*)

lemma *lift1_undef_fw*: $g \equiv lift_1\ (strictify\ f) \Longrightarrow g\ \bot = \bot$
by (*simp del: UU_fun_def*)

lemma *lift2_undef1_fw*: $g \equiv lift_2\ (strictify(\lambda x.\ strictify(f\ x))) \Longrightarrow g\ \bot\ X = \bot$
by (*simp del: UU_fun_def*)

lemma *lift2_undef2_fw*: $g \equiv lift_2\ (strictify(\lambda x.\ strictify(f\ x))) \Longrightarrow g\ X\ \bot = \bot$
by (*simp del: UU_fun_def*)

Generic Theorems on Context Lifted Combinators

lemma *cp_charn* : $[\![\ A\ \tau = B\ \tau;\ cp\ P\ ]\!] \Longrightarrow P\ A\ \tau = P\ B\ \tau$
by (*auto simp: cp_def*)

lemma *cp_by_cpify*: $cp\ P = (\forall\ X\ \tau.\ P\ X\ \tau\ = P\ (lift0\ (X\ \tau))\ \tau)$
by(*auto simp: cp_def lift0_def, rule exI, assumption*)

lemma *cp_lift0* [*simp, intro!*]: $cp(lift_0\ c)$
by (*simp add: cp_def lift0_def, fast*)

lemma *cp_eq* [*simp, intro!*]:
$[\![\ cp\ P;\ cp\ P']\!] \Longrightarrow cp\ (\lambda u\ ua.\ P\ u\ ua = P'\ u\ ua)$
by(*auto simp: cp_def*)

lemma *cp_const* [*simp, intro!*]: $cp(\lambda X.\ c)$
by (*simp add: cp_def, fast*)

lemma *cp_id* [*simp, intro!*]: $cp(\lambda X.\ X)$
by (*simp add: cp_def, fast*)

lemma *cp_compose* [*simp,intro!*]:
$[\![\ cp\ P;\ cp\ P'\ ]\!] \Longrightarrow cp(P\ o\ P')$
apply (*simp add: cp_def o_def*)
apply (*erule exE*)
apply (*erule exE*)
apply (*simp* (*no_asm_simp*))
apply *fast*
done

lemma *cp_compose2*:
$[\![cp\ P;\ cp\ P']\!] \Longrightarrow cp\ (\lambda x.\ P\ (P'\ x))$
by (*simp add: cp_def o_def, auto*)

lemma *cp_lift1* [*simp*]:
$cp\ P \Longrightarrow cp\ (\lambda X.\ lift_1\ f\ (P\ X))$
apply (*simp add: cp_def lift1_def*)

apply (*erule exE*)
apply (*simp* (*no_asm_simp*))
apply *fast*
done

lemma *cp_lift2* [*simp*]:
$[\![cp\ P;\ cp\ P']\!] \Longrightarrow cp\ (\lambda X.\ lift_2\ f\ (P\ X)\ (P'\ X))$
apply (*simp add*: *cp_def lift2_def*)
apply (*erule exE*)
apply (*erule exE*)
apply (*simp* (*no_asm_simp*))
apply *fast*
done

lemma *cp_lift3* [*simp*]:
$[\![cp\ P;\ cp\ P';\ cp\ P'']\!] \Longrightarrow cp\ (\lambda X.\ lift3\ f\ (P\ X)\ (P'\ X)\ (P''\ X))$
apply (*simp add*: *cp_def lift3_def*)
apply (*erule exE*)+
apply (*simp* (*no_asm_simp*))
apply *fast*
done

lemma *cp_lift0_fw* : $f \equiv lift_0\ g \Longrightarrow cp\ f$
by (*simp add*: *lift0_def cp_def*)

lemma *cp_lift1_fw* [*simp*]:
$[\![f \equiv lift_1\ g;\ cp\ P]\!] \Longrightarrow cp\ (\lambda X.\ f\ (P\ X))$
by (*simp add*: *cp_def lift1_def*, *auto*)

lemma *cp_lift2_fw* :
$[\![f \equiv lift_2\ g;\ cp\ P;\ cp\ P']\!]$
$\Longrightarrow cp\ (\lambda X.\ f\ (P\ X)\ (P'\ X))$
by (*simp add*: *cp_def lift2_def*, *auto*)

lemma *cp_lift3_fw*:
$[\![f \equiv lift3\ g;\ cp\ P;\ cp\ P';\ cp\ P'']\!]$
$\Longrightarrow cp\ (\lambda X.\ f\ (P\ X)\ (P'\ X)\ (P''\ X))$
by (*simp add*: *lift3_def cp_def*,*auto*)

lemma *ocl_undef_split* : $X\ \tau = \bot \lor (\exists\, a.\ X\ \tau = (\llcorner a \lrcorner))$
by (*simp add*: *UU_up_def*,*simp add*: *not_down_exists_lift* [*symmetric*])

lemma *ocl_cp_undef_split*:
$[\![cp\ P;\ cp\ P';\ P\ \bot = P'\ \bot;\ X \neq \bot \Longrightarrow P\ X = P'\ X]\!]$
$\Longrightarrow P\ X = P'\ X$

```
  by (simp add: UU_fun_def, auto)


lemma ocl_cp_subst :
  ⟦X τ = X' τ; P X τ = C; cp P⟧ ⟹ P X' τ = C
  by (simp add: cp_def, auto)


lemma ocl_cp_subst2 :
  ⟦X τ = X' τ; P X' τ = P' X' τ; cp P; cp P'⟧ ⟹ P X τ = P' X' τ
  by (simp add: cp_def, auto)


lemma ocl_cp_subst3 :
  ⟦X τ = X' τ; P X' τ = P' X' τ; cp P; cp P'⟧ ⟹ P X τ = P' X τ
  by (simp add: cp_def, auto)


lemma drop_lift_idem [simp]:( ⌈⌊ x ⌋⌉) = x
  by (simp_all split add: up.split)

lemma lift_drop_idem [simp]: ⟦DEF(x)⟧ ⟹ (⌊⌈x⌉⌋) = x
  apply (simp_all add: drop_def DEF_def not_down_exists_lift)
  apply (erule exE, auto)
  done

lemma DEF_set_charn : (∀x∈X. DEF x) = (⊥ ∉ X)
  by (simp add: DEF_def, auto)


lemma cp_option_case:
⟦cp P; cp P'; cp (λ X τ.(λx. P'' x X τ))⟧
  ⟹ cp (λX τ . case (P X τ ) of None ⇒ P' X τ  | Some x  ⇒ P'' x X τ)
apply(simp add: cp_def)
apply(auto)

apply(rule_tac x=λ C τ. case (f C τ) of None ⇒ (fa C τ) | Some x ⇒ (fb  C τ) x in exI)
apply(auto)
apply(rule_tac t=P' X τ in subst)
apply(rotate_tac 1)
apply(erule allE)
apply(rotate_tac −1)
apply(erule allE)
apply(rule sym)
apply(assumption)

apply(rule_tac t=λ x. P'' x X τ in subst)
apply(rotate_tac 2)
apply(erule allE)
```

```
apply(rotate_tac -1)
apply(erule allE)
apply(rule sym)
apply(assumption)

apply(rule refl)
done
```

Higher-order Context-passingness

This theory is inteded to develop higher-order rewriting for LJE-rewriting, i. e., rewriting in the body of iterators or quantifiers with respect tostrong equality.This part of the theory is still under development.

```
constdefs

  cp0    :: ([('τ,'α) VAL, 'τ] ⇒ 'β) ⇒ bool
  cp0 P ≡ ∀ Q τ. P Q τ = P (lift₀ (Q τ)) τ

  cp1    :: ((('τ,'α) VAL ⇒ ('τ,'β) VAL) ⇒ ('τ,'γ) VAL) ⇒ bool
  cp1 P ≡ ∀ Q τ. P Q τ = P (λ x. (lift₀ (Q x τ))) τ

  cp2    :: ((('τ,'α) VAL ⇒ ('τ,'β) VAL ⇒ ('τ,'γ) VAL) ⇒ ('τ,'δ) VAL) ⇒ bool
  cp2 P ≡ ∀ Q τ. P Q τ = P (λ x y. (lift₀ (Q x y τ))) τ
```

Building a ruleset to unfold all cp-ness definitions

```
lemmas ss_cp_defs = cp_def cp0_def cp1_def cp2_def
```

The functional behaviour of $lift_0$. This rule is sometimes useful in a proof.

```
lemma lift0_apply: lift₀ X τ = X
  by(simp add: lift0_def)
```

The rules to "cp-unfold" a term by hand

```
lemma cp0_unfold:
  ⟦ cp0 P; Q τ = R τ ⟧ ⟹ P Q τ = P (lift₀ (R τ)) τ
  apply(unfold cp0_def, erule_tac x=Q in allE, erule_tac x=τ in allE)
  by(simp add: lift0_apply)

lemma cp1_unfold:
  ⟦ cp1 P; ⋀ x. Q x τ = R x τ ⟧ ⟹ P Q τ = P (λ x. lift₀ (R x τ)) τ
  apply(unfold cp1_def, erule_tac x=Q in allE, erule_tac x=τ in allE)
  by(simp add: lift0_apply)

lemma cp2_unfold:
  ⟦ cp2 P; ⋀ x y. Q x y τ = R x y τ ⟧ ⟹ P Q τ = P (λ x y. lift₀ (R x y τ)) τ
  apply(unfold cp2_def, erule_tac x=Q in allE, erule_tac x=τ in allE)
  by(simp add: lift0_apply)
```

The rules to "cp-fold" a term by hand

lemma *cp0_fold*:
  $\llbracket$ *cp0 P*; $Q\ \tau = R\ \tau$ $\rrbracket \Longrightarrow P\ (lift_0\ (Q\ \tau))\ \tau\ = P\ R\ \tau$
  by(*erule cp0_unfold*[*symmetric*], *simp*)

lemma *cp1_fold*:
  $\llbracket$ *cp1 P*; $\bigwedge x.\ Q\ x\ \tau = R\ x\ \tau$ $\rrbracket \Longrightarrow P\ (\lambda\ x.\ lift_0\ (Q\ x\ \tau))\ \tau\ = P\ R\ \tau$
  by(*erule cp1_unfold*[*symmetric*], *simp*)

lemma *cp2_fold*:
  $\llbracket$ *cp2 P*; $\bigwedge x\ y.\ Q\ x\ y\ \tau = R\ x\ y\ \tau$ $\rrbracket \Longrightarrow P\ (\lambda\ x\ y.\ lift_0\ (Q\ x\ y\ \tau))\ \tau\ = P\ R\ \tau$
  by(*erule cp2_unfold*[*symmetric*], *simp*)

Rules to Establish cp

cp0 rules

lemma *cp0_const* [*simp*]: *cp0* $(\lambda\ x\ \tau.\ P)$
  by(*simp add*: *cp0_def*)

lemma *cp0_id* [*simp*]: *cp0* $(\lambda\ x.\ x)$
  by(*simp add*: *cp0_def lift0_apply*)

lemma *cp0_app*:
  assumes *cp0_1_P*: $\bigwedge y.$ *cp0* $(\lambda\ x.\ (P\ x\ y))$
  and *cp0_2_P*: $\bigwedge x.$ *cp0* $(P\ x)$
  shows *cp0* $(\lambda\ x.\ P\ x\ x)$
  apply(*unfold cp0_def*, *clarify*)
  apply(*rule trans*, *rule_tac* $P{=}(\lambda\ x.\ P\ x\ ?Y)$ **and** $\tau{=}\tau$ **in** *cp0_unfold*)
  apply(*rule cp0_1_P*, *rule refl*)
  apply(*rule trans*, *rule_tac* $P{=}(\lambda\ x.\ P\ ?X\ x)$ **and** $\tau{=}\tau$ **in** *cp0_unfold*)
  apply(*rule cp0_2_P*, *rule refl*)
  apply(*rule refl*)
  done

lemma *cp0_of_cp0*:
  assumes *cp0_P*: *cp0 P*
  and *cp0_Q*: *cp0 Q*
  shows *cp0* $(\lambda\ x.\ P\ (Q\ x))$
  apply(*unfold cp0_def*, *clarify*)
  apply(*rule trans*, *rule_tac* $P{=}(\lambda\ x.\ P\ x)$ **and** $\tau{=}\tau$ **in** *cp0_unfold*)
  apply(*rule cp0_P*, *rule refl*)
  apply(*rule sym*, *rule trans*, *rule_tac* $P{=}(\lambda\ x.\ P\ x)$ **and** $\tau{=}\tau$ **in** *cp0_unfold*)
  apply(*rule cp0_P*)
  apply(*rule_tac* $P{=}(\lambda\ x.\ Q\ x)$ **and** $\tau{=}\tau$ **in** *cp0_fold*)
  apply(*rule cp0_Q*, *rule refl*)
  apply(*rule refl*)
  done

```
lemma cp0_of_cp1:
  assumes cp1_P: cp1 P
  and     cp0_1_Q: ⋀ a. cp0 (λ x. Q x a)
  shows          cp0 (λ x. P (Q x))
  apply(unfold cp0_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
  apply(rule cp1_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
  apply(rule cp1_P)
  apply(rule_tac P=(λ x. Q x ?X) and τ=τ in cp0_fold)
  apply(rule cp0_1_Q, rule refl)
  apply(rule refl)
  done

lemma cp0_of_cp2:
  assumes cp2_P: cp2 P
  and     cp0_1_Q: ⋀ a b. cp0 (λ x. Q x a b)
  shows          cp0 (λ x. P (Q x))
  apply(unfold cp0_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
  apply(rule cp2_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
  apply(rule cp2_P)
  apply(rule_tac P=(λ x. Q x ?X ?Y) and τ=τ in cp0_fold)
  apply(rule cp0_1_Q, rule refl)
  apply(rule refl)
  done
```

In the following, we prove the cp1 rules.

```
lemma cp1_const [simp]: cp1 (λ x τ. P)
  by(simp add: cp1_def)
```

This rule corresponds to *cp0_id*. It has the same structure except for the constant evaluation which is of course not needed for a nullary argument.

```
lemma cp1_const_eval [simp]: cp1 (λ x. x X)
  by(simp add: cp1_def lift0_apply)

lemma cp1_app:
  assumes cp1_1_P: ⋀ y. cp1 (λ x. (P x y))
  and     cp1_2_P: ⋀ x. cp1 (P x)
  shows          cp1 (λ x. P x x)
  apply(unfold cp1_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x ?Y) and τ=τ in cp1_unfold)
  apply(rule cp1_1_P, rule refl)
  apply(rule trans, rule_tac P=(λ x. P ?X x) and τ=τ in cp1_unfold)
  apply(rule cp1_2_P, rule refl)
  apply(rule refl)
  done
```

```
lemma cp1_of_cp0:
  assumes cp0_P: cp0 P
  and     cp1_Q: cp1 Q
  shows          cp1 (λ x. P (Q x))
  apply(unfold cp1_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp0_unfold)
  apply(rule cp0_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp0_unfold)
  apply(rule cp0_P)
  apply(rule_tac P=(λ x. Q x) and τ=τ in cp1_fold)
  apply(rule cp1_Q, rule refl)
  apply(rule refl)
  done

lemma cp1_of_cp1:
  assumes cp1_P: cp1 P
  and    cp1_1_Q: ⋀ a. cp1 (λ x. Q x a)
  shows          cp1 (λ x. P (Q x))
  apply(unfold cp1_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
  apply(rule cp1_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
  apply(rule cp1_P)
  apply(rule_tac P=(λ x. Q x ?X) and τ=τ in cp1_fold)
  apply(rule cp1_1_Q, rule refl)
  apply(rule refl)
  done

lemma cp1_of_cp2:
  assumes cp2_P: cp2 P
  and    cp1_1_Q: ⋀ a b. cp1 (λ x. Q x a b)
  shows          cp1 (λ x. P (Q x))
  apply(unfold cp1_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
  apply(rule cp2_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
  apply(rule cp2_P)
  apply(rule_tac P=(λ x. Q x ?X ?Y) and τ=τ in cp1_fold)
  apply(rule cp1_1_Q, rule refl)
  apply(rule refl)
  done
```

cp2 rules

```
lemma cp2_const [simp]: cp2 (λ x τ. P)
  by(simp add: cp2_def)

lemma cp2_const_eval [simp]: cp2 (λ x. x X Y)
  by(simp add: cp2_def lift0_apply)
```

```
lemma cp2_app:
  assumes cp2_1_P: ⋀ y. cp2 (λ x. (P x y))
  and     cp2_2_P: ⋀ x. cp2 (P x)
  shows           cp2 (λ x. P x x)
  apply(unfold cp2_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x ?Y) and τ=τ in cp2_unfold)
  apply(rule cp2_1_P, rule refl)
  apply(rule trans, rule_tac P=(λ x. P ?X x) and τ=τ in cp2_unfold)
  apply(rule cp2_2_P, rule refl)
  apply(rule refl)
  done

lemma cp2_of_cp0:
  assumes cp0_P: cp0 P
  and     cp2_Q: cp2 Q
  shows          cp2 (λ x. P (Q x))
  apply(unfold cp2_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp0_unfold)
  apply(rule cp0_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp0_unfold)
  apply(rule cp0_P)
  apply(rule_tac P=(λ x. Q x) and τ=τ in cp2_fold)
  apply(rule cp2_Q, rule refl)
  apply(rule refl)
  done

lemma cp2_of_cp1:
  assumes cp1_P: cp1 P
  and   cp2_1_Q: ⋀ a. cp2 (λ x. Q x a)
  shows          cp2 (λ x. P (Q x))
  apply(unfold cp2_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
  apply(rule cp1_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
  apply(rule cp1_P)
  apply(rule_tac P=(λ x. Q x ?X) and τ=τ in cp2_fold)
  apply(rule cp2_1_Q, rule refl)
  apply(rule refl)
  done

lemma cp2_of_cp2:
  assumes cp2_P: cp2 P
  and   cp2_1_Q: ⋀ a b. cp2 (λ x. Q x a b)
  shows          cp2 (λ x. P (Q x))
  apply(unfold cp2_def, clarify)
  apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
  apply(rule cp2_P, rule refl)
  apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
  apply(rule cp2_P)
```

apply(*rule_tac P*=(λ *x*. *Q x ?X ?Y*) **and** τ=τ **in** *cp2_fold*)
apply(*rule cp2_1_Q*, *rule refl*)
apply(*rule refl*)
done

Deciding cp using *cp_unfold*

By converting a term into explicit cp-representation one can also decide all context-passingness questions. Since the tactic uses a stepwise unfolding it can deal with cases where simp combined with chain rules for cp-ness does not work. These are cases where currently the rule *cp_compose2* is applied by hand. The decision using *cp_unfold* works as follows:

1. apply *cp_unfold*
2. if one of the rules *cp0_cp_unfolded*, *cp1_cp_unfolded*, or *cp2_cp_unfolded* matches were done otherwise the cp-ness proposition is false (assuming that the simpset of *cp_unfold* is complete with respect to the set of OCL operations)

lemma *cp0_cp_unfolded*:
cp0 (λ *x* τ. *P* ($lift_0$ (*x* τ)) τ)
by(*simp add*: *cp0_def lift0_apply*)

lemma *cp1_cp_unfolded*:
cp1 (λ *Q* τ. *P* (λ *x*. $lift_0$ (*Q x* τ)) τ)
by(*simp add*: *cp1_def lift0_apply*)

lemma *cp2_cp_unfolded*:
cp2 (λ *Q* τ. *P* (λ *x y*. $lift_0$ (*Q x y* τ)) τ)
by(*simp add*: *cp2_def lift0_apply*)

Higher-Order lifting construction

constdefs
lift1' :: [('γ ⇒ 'η ⇒ 'α), 'α ⇒ 'β] ⇒ ['γ, 'η] ⇒ 'β
lift1' a1 ≡ (λ *f x* τ. *f* (*a1 x* τ))

lift2' :: [('α ⇒ 'γ ⇒ 'η), ('β ⇒ 'γ ⇒ 'f), ('η ⇒ 'f ⇒ 'δ)] ⇒ ['α, 'β, 'γ] ⇒ 'δ
lift2' a1 a2 ≡ (λ *f x y* τ. *f* (*a1 x* τ) (*a2 y* τ))

lift3' :: ['η⇒'h⇒'α, 'f⇒'h⇒'β, 'g⇒'h⇒'γ, 'α⇒'β⇒'γ⇒'δ] ⇒ ['η, 'f, 'g, 'h] ⇒ 'δ
lift3' a1 a2 a3 ≡ (λ *f x y z* τ. *f* (*a1 x* τ) (*a2 y* τ) (*a3 z* τ))

lift_arg0 :: ('τ ⇒ 'β) ⇒ ('τ ⇒ 'β)
lift_arg0 ≡ *id*

lift_arg1 :: (('τ ⇒ 'γ) ⇒ ('τ ⇒ 'δ)) ⇒ ('τ ⇒ ('γ ⇒ 'δ))
lift_arg1 ≡ (λ *f* τ *x*. *f* ($lift_0$ *x*) τ)

$lift\_arg2 \quad :: ([{'}\alpha{\Rightarrow}{'}\beta,\ {'}\gamma{\Rightarrow}{'}\delta] \Rightarrow ({'}\eta{\Rightarrow}{'}f)) \Rightarrow ([{'}\eta,\ {'}\beta,\ {'}\delta] \Rightarrow {'}f)$
$lift\_arg2 \quad \equiv (\lambda\ f\ \tau\ x\ y.\ f\ (lift_0\ x)\ (lift_0\ y)\ \tau)$

Rulesets to unfold the whole lifting constructions

lemmas *ss_lifting* = *lift1_def lift2_def lift3_def*
lift1′_def lift2′_def lift3′_def
lift_arg0_def lift_arg1_def lift_arg2_def
DEF_def strictify_def isStrict_def
smash_def

lemmas *ss_lifting′* = *lift0_def ss_lifting*

The Link to the old Combinator Set.

Note, *lift1*, *lift2*, *lift3* are special cases of these general combinators.

lemma *lift1_by_lift1′*: $lift_1$ = (*lift1′ lift_arg0*)
by(*simp add*: *lift1_def lift1′_def lift_arg0_def*)

lemma *lift2_by_lift2′*: $lift_2$ = (*lift2′ lift_arg0 lift_arg0*)
by(*simp add*: *lift2_def lift2′_def lift_arg0_def*)

lemma *lift3_by_lift3′*: *lift3* = (*lift3′ lift_arg0 lift_arg0 lift_arg0*)
by(*simp add*: *lift3_def lift3′_def lift_arg0_def*)

Forward rules for the currently used constructions:

As we have in the section about *cp_unfold*, transforming a term into explicit cp-representation is very powerful. Its foundation is *cp_unfold* which relies on a simplifier set containing all context-passingess predicates for every argument of every OCL operation. These set can be created by putting all forward rules from below whose first assumption is unifyable with the definition of the OCL function into this set.

The *fw_OCL_cp0_chain* rules are the old *OCL_cp* rules who can be put into the standard simplifier set.

Context-passingness by construction

ML ⟪
```
local
  val cpR_prover_simpset = simpset() addsimps ((thms ss_lifting′)
                                              @ (thms ss_cp_defs))
in
  fun cpR_prover mk_name (goal,arity,position,lifting) =
    let val (rname, name_prefix) = mk_name arity position lifting
        val thm  = prove_goalw (the_context()) [] goal
                   (fn prems => [ALLGOALS(simp_tac
                                 (cpR_prover_simpset
```

```
                                        addsimps prems))
                                ])
    in (store_thm(rname,thm), rname, name_prefix)
    end
end
⟫
```

pointwise context-passingness rules

```
ML ⟪
local
  fun mk_name_cpR arity position lifting =
      let val name_prefix = OCL_cp ^ (Int.toString arity) ^ _
                              ^ (Int.toString position) ^ _
      in  (fw_ ^ name_prefix ^ lifting, name_prefix) end
in
  val cpR_thms = map (cpR_prover mk_name_cpR) [
    (f ≡ lift1 P ⟹ cp0 f, 0, 1, lift1),
    (f ≡ (lift1′ lift_arg0) P ⟹ cp0 f, 0, 1, lift1′),

    (f ≡ lift2 P ⟹ cp0 (λ x. f x Y), 0, 1, lift2),
    (f ≡ lift2 P ⟹ cp0 (λ x. f X x), 0, 2, lift2),
    (f ≡ (lift2′ lift_arg0 aY) P ⟹ cp0 (λ x. f x Y), 0, 1, lift2′),
    (f ≡ (lift2′ aX lift_arg0) P ⟹ cp0 (λ x. f X x), 0, 2, lift2′),
    (f ≡ (lift2′ aX lift_arg1) P ⟹ cp1 (λ x. f X x), 1, 2, lift2′),
    (f ≡ (lift2′ aX lift_arg2) P ⟹ cp2 (λ x. f X x), 2, 2, lift2′),

    (f ≡ lift3 P ⟹ cp0 (λ x. f x Y Z), 0, 1, lift3),
    (f ≡ lift3 P ⟹ cp0 (λ x. f X x Z), 0, 2, lift3),
    (f ≡ lift3 P ⟹ cp0 (λ x. f X Y x), 0, 3, lift3),
    (f ≡ (lift3′ lift_arg0 aY aZ) P ⟹ cp0 (λ x. f x Y Z), 0, 1, lift3′),
    (f ≡ (lift3′ aX lift_arg0 aZ) P ⟹ cp0 (λ x. f X x Z), 0, 2, lift3′),
    (f ≡ (lift3′ aX aY lift_arg0) P ⟹ cp0 (λ x. f X Y x), 0, 3, lift3′),
    (f ≡ (lift3′ aX lift_arg1 aZ) P ⟹ cp1 (λ x. f X x Z), 1, 2, lift3′),
    (f ≡ (lift3′ aX lift_arg2 aZ) P ⟹ cp2 (λ x. f X x Z), 2, 2, lift3′)
  ]
end
⟫
```

pointwise cp_fold rules

```
ML ⟪
local
  fun mk_name_cpR_fold arity position lifting =
      let val name_prefix = OCL_cp ^ (Int.toString arity) ^ _fold_
                              ^ (Int.toString position) ^ _
      in  (fw_ ^ name_prefix ^ lifting, name_prefix) end
in
  val cpR_fold_thms = map (cpR_prover mk_name_cpR_fold) [
    (f ≡ lift1 P ⟹ f (lift₀ (X τ)) τ = f X τ, 0, 1, lift1),
```

$(f \equiv (\mathit{lift1}'\ \mathit{lift\_arg0})\ P \Longrightarrow f\ (\mathit{lift}_0\ (X\ \tau))\ \tau = f\ X\ \tau,\ 0,\ 1,\ \mathit{lift1}'),$

$(f \equiv \mathit{lift2}\ P \Longrightarrow f\ (\mathit{lift}_0\ (X\ \tau))\ Y\ \tau = f\ X\ Y\ \tau,\ 0,\ 1,\ \mathit{lift2}),$
$(f \equiv \mathit{lift2}\ P \Longrightarrow f\ X\ (\mathit{lift}_0\ (Y\ \tau))\ \tau = f\ X\ Y\ \tau,\ 0,\ 2,\ \mathit{lift2}),$

$(f \equiv (\mathit{lift2}'\ \mathit{lift\_arg0}\ aY)\ P \Longrightarrow f\ (\mathit{lift}_0\ (X\ \tau))\ Y\ \tau = f\ X\ Y\ \tau,\ 0,\ 1,\ \mathit{lift2}'),$
$(f \equiv (\mathit{lift2}'\ aX\ \mathit{lift\_arg0})\ P \Longrightarrow f\ X\ (\mathit{lift}_0\ (Y\ \tau))\ \tau = f\ X\ Y\ \tau,\ 0,\ 2,\ \mathit{lift2}'),$
$(f \equiv (\mathit{lift2}'\ aX\ \mathit{lift\_arg1})\ P \Longrightarrow f\ X\ (\lambda\ x.\ \mathit{lift}_0\ (Y\ x\ \tau))\ \tau = f\ X\ Y\ \tau,\ 1,\ 2,\ \mathit{lift2}'),$
$(f \equiv (\mathit{lift2}'\ aX\ \mathit{lift\_arg2})\ P \Longrightarrow f\ X\ (\lambda\ x\ y.\ \mathit{lift}_0\ (Y\ x\ y\ \tau))\ \tau = f\ X\ Y\ \tau,\ 2,\ 2,\ \mathit{lift2}'),$

$(f \equiv \mathit{lift3}\ P \Longrightarrow f\ (\mathit{lift}_0\ (X\ \tau))\ Y\ Z\ \tau = f\ X\ Y\ Z\ \tau,\ 0,\ 1,\ \mathit{lift3}),$
$(f \equiv \mathit{lift3}\ P \Longrightarrow f\ X\ (\mathit{lift}_0\ (Y\ \tau))\ Z\ \tau = f\ X\ Y\ Z\ \tau,\ 0,\ 2,\ \mathit{lift3}),$
$(f \equiv \mathit{lift3}\ P \Longrightarrow f\ X\ Y\ (\mathit{lift}_0\ (Z\ \tau))\ \tau = f\ X\ Y\ Z\ \tau,\ 0,\ 3,\ \mathit{lift3}),$

$(f \equiv (\mathit{lift3}'\ \mathit{lift\_arg0}\ aY\ aZ)\ P \Longrightarrow f\ (\mathit{lift}_0\ (X\ \tau))\ Y\ Z\ \tau = f\ X\ Y\ Z\ \tau,\ 0,\ 1,\ \mathit{lift3}'),$
$(f \equiv (\mathit{lift3}'\ aX\ \mathit{lift\_arg0}\ aZ)\ P \Longrightarrow f\ X\ (\mathit{lift}_0\ (Y\ \tau))\ Z\ \tau = f\ X\ Y\ Z\ \tau,\ 0,\ 2,\ \mathit{lift3}'),$
$(f \equiv (\mathit{lift3}'\ aX\ aY\ \mathit{lift\_arg0})\ P \Longrightarrow f\ X\ Y\ (\mathit{lift}_0\ (Z\ \tau))\ \tau = f\ X\ Y\ Z\ \tau,\ 0,\ 3,\ \mathit{lift3}'),$
$(f \equiv (\mathit{lift3}'\ aX\ \mathit{lift\_arg2}\ aZ)\ P \Longrightarrow f\ X\ (\lambda\ x\ y.\ \mathit{lift}_0\ (Y\ x\ y\ \tau))\ Z\ \tau = f\ X\ Y\ Z\ \tau,\ 2,\ 2,\ \mathit{lift3}')$
]
end

⟫

Chain rules for cp0 usable by simp or fast

Unary OCL operations: old lifting construction.

lemma *fw_OCL_cp0_chain0_lift1*:
assumes *f_def*: $f \equiv \mathit{lift1}\ P$
and *cp0_X*: *cp0 X*
shows *cp0* $(\lambda\ x.\ f\ (X\ x))$
by(*insert cp0_X*, *simp add*: *f_def cp0_def lift1_def lift0_apply*)

lemma *fw_OCL_cp0_chain0_lift1'*:
assumes *f_def*: $f \equiv (\mathit{lift1}'\ \mathit{lift\_arg0})\ P$
and *cp0_X*: *cp0 X*
shows *cp0* $(\lambda\ x.\ f\ (X\ x))$
by(*insert cp0_X*, *simp add*: *f_def cp0_def lift_arg0_def lift1'_def lift0_apply*)

Binary OCL operations: old lifting construction.

lemma *fw_OCL_cp0_chain00_lift2*:
assumes *f_def*: $f \equiv \mathit{lift2}\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: *cp0 Y*
shows *cp0* $(\lambda\ x.\ f\ (X\ x)\ (Y\ x))$
by(*insert cp0_X cp0_Y*,
simp add: *f_def cp0_def lift2_def lift0_apply*)

lemma *fw_OCL_cp0_chain00_lift2'*:
assumes *f_def*: $f \equiv (lift2'\ lift\_arg0\ lift\_arg0)\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: *cp0 Y*
shows $cp0\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x))$
by(*insert cp0_X cp0_Y*,
simp add: *f_def cp0_def lift_arg0_def lift2'_def lift0_apply*)

lemma *fw_OCL_cp0_chain01_lift2'*:
assumes *f_def*: $f \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: $\bigwedge\ a.\ cp0\ (\lambda\ x.\ Y\ x\ a)$
shows $cp0\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x))$
by(*insert cp0_X cp0_Y*,
simp add: *f_def cp0_def lift_arg0_def lift_arg1_def lift2'_def lift0_apply*)

lemma *fw_OCL_cp0_chain02_lift2'*:
assumes *f_def*: $f \equiv (lift2'\ lift\_arg0\ lift\_arg2)\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: $\bigwedge\ a\ b.\ cp0\ (\lambda\ x.\ Y\ x\ a\ b)$
shows $cp0\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x))$
by(*insert cp0_X cp0_Y*,
simp add: *f_def cp0_def lift_arg0_def lift_arg2_def lift2'_def lift0_apply*)

Tertiary OCL operations: `->iterate` and if.

lemma *fw_OCL_cp0_chain000_lift3*:
assumes *f_def*: $f \equiv lift3\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: *cp0 Y*
and *cp0_Z*: *cp0 Z*
shows $cp0\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x)\ (Z\ x))$
by(*insert cp0_X cp0_Y cp0_Z*,
simp add: *f_def cp0_def lift3_def lift0_apply*)

lemma *fw_OCL_cp0_chain000_lift3'*:
assumes *f_def*: $f \equiv (lift3'\ lift\_arg0\ lift\_arg0\ lift\_arg0)\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: *cp0 Y*
and *cp0_Z*: *cp0 Z*
shows $cp0\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x)\ (Z\ x))$
by(*insert cp0_X cp0_Y cp0_Z*,
simp add: *f_def cp0_def lift_arg0_def lift3'_def lift0_apply*)

lemma *fw_OCL_cp0_chain020_lift3'*:
assumes *f_def*: $f \equiv (lift3'\ lift\_arg0\ lift\_arg2\ lift\_arg0)\ P$
and *cp0_X*: *cp0 X*
and *cp0_Y*: $\bigwedge\ a\ b.\ cp0\ (\lambda\ x.\ Y\ x\ a\ b)$
and *cp0_Z*: *cp0 Z*
shows $cp0\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x)\ (Z\ x))$

```
  by(insert cp0_X cp0_Y cp0_Z,
    simp add: f_def cp0_def lift_arg0_def lift_arg2_def lift3'_def)
```

Chain rules for cp usable by simp or fast.

Unary OCL operations: old lifting construction.

```
lemma fw_OCL_cp_chain0_lift1':
  assumes f_def: f ≡ (lift1' lift_arg0) P
  and      cp_X: cp X
  shows           cp (λ x. f (X x))
  by(insert cp_X, simp add: f_def cp_by_cpify lift_arg0_def lift1'_def lift0_apply)
```

Binary OCL operations: old lifting construction.

```
lemma fw_OCL_cp_chain00_lift2':
  assumes f_def: f ≡ (lift2' lift_arg0 lift_arg0) P
  and      cp_X: cp X
  and      cp_Y: cp Y
  shows           cp (λ x. f (X x) (Y x))
  by(insert cp_X cp_Y,
    simp add: f_def cp_by_cpify lift_arg0_def lift2'_def lift0_apply)

lemma fw_OCL_cp_chain01_lift2':
  assumes f_def: f ≡ (lift2' lift_arg0 lift_arg1) P
  and      cp_X: cp X
  and      cp_Y: ⋀ a. cp (λ x. Y x a)
  shows           cp (λ x. f (X x) (Y x))
  by(insert cp_X cp_Y,
    simp add: f_def cp_by_cpify lift_arg0_def lift_arg1_def lift2'_def lift0_apply)

lemma fw_OCL_cp_chain02_lift2':
  assumes f_def: f ≡ (lift2' lift_arg0 lift_arg2) P
  and      cp_X: cp X
  and      cp_Y: ⋀ a b. cp (λ x. Y x a b)
  shows           cp (λ x. f (X x) (Y x))
  by(insert cp_X cp_Y,
    simp add: f_def cp_by_cpify lift_arg0_def lift_arg2_def lift2'_def lift0_apply)
```

Tertiary OCL operations: `->iterate` and if.

```
lemma fw_OCL_cp_chain000_lift3':
  assumes f_def: f ≡ (lift3' lift_arg0 lift_arg0 lift_arg0) P
  and      cp_X: cp X
  and      cp_Y: cp Y
  and      cp_Z: cp Z
  shows           cp (λ x. f (X x) (Y x) (Z x))
  by(insert cp_X cp_Y cp_Z,
    simp add: f_def cp_by_cpify lift_arg0_def lift3'_def lift0_apply)

lemma fw_OCL_cp_chain020_lift3':
```

assumes *f_def*: $f \equiv (lift3'\ lift\_arg0\ lift\_arg2\ lift\_arg0)\ P$
and *cp_X*: $cp\ X$
and *cp_Y*: $\bigwedge a\ b.\ cp\ (\lambda\ x.\ Y\ x\ a\ b)$
and *cp_Z*: $cp\ Z$
shows $cp\ (\lambda\ x.\ f\ (X\ x)\ (Y\ x)\ (Z\ x))$
by(*insert cp_X cp_Y cp_Z,*
simp add: f_def cp_by_cpify lift_arg0_def lift_arg2_def lift3'_def)

Note, ll of these rules are registered for the use with ocl_setup_op in the theory OCLL_Kernel.

cp0 is equivalent to what we already have with *cp*. Thus it is currently added to the simpset to allow for a decision of *cp0*-questions based on the machinery already built for deciding *cp*-questions. This will change when the theories are cleaned up.

lemma *cp0_eq_cp*:
$cp0\ P = cp\ P$
by(*simp add: cp0_def cp_by_cpify*)

Accessing Type Sums

constdefs
$FromL\ :: ('\alpha + '\beta)\ up \Rightarrow '\alpha\ up$
$FromL\ x\ \equiv case\ x\ of\ lift\ v \Rightarrow (\ sum\_case\ (\lambda\ x.\ \llcorner x \lrcorner)\ (\lambda\ x.\ down)\ v)$
$|\ down \Rightarrow down$

constdefs
$FromR\ :: ('\alpha + '\beta)\ up \Rightarrow '\beta\ up$
$FromR\ x\ \equiv case\ x\ of\ lift\ v \Rightarrow (\ sum\_case\ (\lambda x.\ down\ )\ (\lambda\ x.\ \llcorner x \lrcorner)\ v)$
$|\ down \Rightarrow down$

lemma *FromL_UU* [*simp*]: $FromL\ \bot = \bot$
by (*simp add: FromL_def*)

lemma *FromR_UU* [*simp*]: $FromR\ \bot = \bot$
by (*simp add: FromR_def*)

lemma *FromL_down* [*simp*]: $FromL\ down = down$
by (*simp add: FromL_def*)

lemma *FromR_down* [*simp*]: $FromR\ down = down$
by (*simp add: FromR_def*)

lemma *FromL_lift_Inl_id* [*simp*]: $FromL\ \llcorner Inl\ x \lrcorner = \llcorner x \lrcorner$
by (*simp add: FromL_def*)

lemma *FromR_lift_Inr_id* [*simp*]: $FromR\ \llcorner Inr\ x \lrcorner\ = lift\ x$
by (*simp add: FromR_def*)

lemma *FromR_Inl_UU* [*simp*]: $FromR\ \llcorner Inl\ x \lrcorner = \bot$

by (*simp add*: *FromR_def*)

lemma *FromL_Inr_UU* [*simp*]: *FromL* $\llcorner$*Inr x*$\lrcorner$ = $\bot$
by (*simp add*: *FromL_def*)

lemma *h1*: *DEF* (*FromL*(*x*)) $\Longrightarrow$ $\llcorner$*Inl*$\ulcorner$*FromL x*$\urcorner$$\lrcorner$ = *x*
apply(*simp add*: *FromL_def lift_drop_idem*)
apply(*case_tac x*, *simp add*: *lift_drop_idem drop_lift_idem*,*auto*)
apply(*case_tac* α, *simp add*: *lift_drop_idem drop_lift_idem*,*auto*)
done

lemma *h2*: *DEF* (*FromL*(*x*)) $\Longrightarrow$ *Inl*$\ulcorner$*FromL x*$\urcorner$ = $\ulcorner$*x*$\urcorner$
apply(*simp add*: *FromL_def lift_drop_idem*)
apply(*case_tac x*, *simp add*: *lift_drop_idem drop_lift_idem*, *auto*)
apply(*case_tac* α, *simp add*: *lift_drop_idem drop_lift_idem*, *auto*)
done

lemma *DEF_FromL_exists_Inl*: *DEF*(*FromL x*) = ($\exists$ *y*. *x* = $\llcorner$*Inl y*$\lrcorner$)
apply (*simp add*: *FromL_def*)
apply (*case_tac x*, *simp*, *auto*)
apply (*case_tac* α, *simp*, *auto*)
done

lemma *DEF_FromR_exists_drop_Inr*: *DEF* (*FromR*(*x*)) $\Longrightarrow$ $\llcorner$*Inr* $\ulcorner$*FromR x*$\urcorner$$\lrcorner$ = *x*
apply(*simp add*: *FromR_def lift_drop_idem*)
apply(*case_tac x*, *simp add*: *lift_drop_idem drop_lift_idem*, *auto*)
apply(*case_tac* α,*simp add*: *lift_drop_idem drop_lift_idem*, *auto*)
done

lemma *FromL_Inl_inv* [*simp*]: (*FromL* $\llcorner$*xb*$\lrcorner$ = $\llcorner$*X*$\lrcorner$) = (*xb* = *Inl X*)
apply (*auto*)
apply (*cases xb*)
apply (*auto*)
done

lemma *FromR_Inr_inv* [*simp*]: (*FromR* $\llcorner$*xb*$\lrcorner$ = $\llcorner$*X*$\lrcorner$) = (*xb* = *Inr X*)
apply (*auto*)
apply (*cases xb*)
apply (*auto*)
done

lemma *Inr_FromR_drop*: *DEF* (*FromR*(*x*)) $\Longrightarrow$ *Inr* $\ulcorner$*FromR x*$\urcorner$ = $\ulcorner$*x*$\urcorner$
apply(*simp add*: *FromR_def lift_drop_idem*)
apply(*case_tac x*, *simp add*: *lift_drop_idem drop_lift_idem*,*auto*)
apply(*case_tac* α, *simp add*: *lift_drop_idem drop_lift_idem*,*auto*)
done

lemma *DEF_FromR_exits_Inr*: $DEF\ (FromR\ x) = (\exists\, y.\ x = \llcorner Inr\ y \lrcorner)$
  apply (*simp add*: *FromR_def*)
  apply (*case_tac x*,*simp*,*auto*)
  apply (*case_tac* α,*simp*,*auto*)
  done

lemma *DEF_FromLD* : $DEF(FromL\ x) \Longrightarrow\ DEF(x)$
  apply (*simp add*: *DEF_def FromL_def not_down_exists_lift*)
  apply (*case_tac x*, *simp*, *simp*)
  done

lemma *DEF_FromRD* : $DEF(FromR\ x) \Longrightarrow\ DEF(x)$
  apply (*simp add*: *DEF_def FromR_def not_down_exists_lift*)
  apply (*case_tac x*, *simp*, *simp*)
  done

end

B.3.2. The OCL Kernel

theory *OCL_Kernel*
imports
  $HOLOCL_HOME/src/Lifting
begin

Global Syntax Infrastructure

In the theory files, this section contains the SML code for setting up the infra-strucure for switching between mathematical and other notations for OCL.

ocl_setup_op

In the theory files, this section contains the SML code that implements the rulebase for the context-passingness rules that are registered to the simplifier to decide cp-ness in proofs *without* the help of *ocl_cp_unfold* or *ocl_simp*.

Definition of the Isar interface to `ocl_setup_op`

In the theory files, this section contains the SML code that sets up the isar syntax for command setup_op.

Setting up the Rulebase for cp-reasoning.

In the theory files, this section contains the SML code that sets up the rulebase for the context-passingness rules that are registered to the simplifier to decide cp-ness questions in interactive proofs *without* the help of *ocl_cp_unfold* or *ocl_simp*.

Registering the cp-rules used to decide cp-ness questions in *cp_unfold* and the cp-fold rules used to convert a term in explicit cp-representation to a term in implicit cp-representation.

end

B.3.3. Type Definition for OclUndefined

theory *OCL_Undefined*
imports
$HOLOCL_HOME/src/OCL_Kernel$
begin

As previously mentioned, all types, let it be basic types or class types, my contain the element ⊥ denoting undefinedness. In classifier types, this can be interpreted as equivalent in unboxed types to a null-reference. Therefore, there is a universal constant in OCL, called ⊥, that is constructed as follows:

constdefs
OclUndefined :: ('a,'b::bot) VAL
OclUndefined ≡ $lift_0$ ⊥

syntax
_OclUndefined_std :: ('a,'b::bot) VAL (`OclUndefined`)

syntax
_OclUndefined_ascii :: ('a,'b::bot) VAL (*undef*)

syntax
_OclUndefined_math :: ('a,'b::bot) VAL (⊥)

Rules describing the effect of strictify on the OCL level

lemma *fw_OCL_undef_1_lift1'_0*:
assumes *f_def*: *f* ≡ (*lift1' lift_arg0*) (*strictify P*)
shows *f OclUndefined* = *OclUndefined*
by(*simp add*: *f_def OclUndefined_def ss_lifting'*)

binary rules

lemma *fw_OCL_undef_1_lift2'_00*:
assumes *f_def*: *f* ≡ (*lift2' lift_arg0 lift_arg0*) (*strictify P*)
shows *f OclUndefined Y* = *OclUndefined*

```
  by(simp add: f_def OclUndefined_def ss_lifting')

lemma fw_OCL_undef_1_lift2'_01:
  assumes f_def: f ≡ (lift2' lift_arg0 lift_arg1) (strictify P)
  shows          f OclUndefined Y = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')

lemma fw_OCL_undef_2_lift2'_00:
  assumes f_def: f ≡ (lift2' lift_arg0 lift_arg0) (λ x. strictify (P x))
  shows          f X OclUndefined = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')
```

tertiary rules

```
lemma fw_OCL_undef_1_lift3'_000:
  assumes f_def: f ≡ (lift3' lift_arg0 lift_arg0 lift_arg0) (strictify P)
  shows          f OclUndefined Y Z = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')

lemma fw_OCL_undef_1_lift3'_020:
  assumes f_def: f ≡ (lift3' lift_arg0 lift_arg2 lift_arg0) (strictify P)
  shows          f OclUndefined Y Z = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')

lemma fw_OCL_undef_3_lift3'_000:
  assumes f_def: f ≡ (lift3' lift_arg0 lift_arg0 lift_arg0) (λ x y. strictify (P x y))
  shows          f X Y OclUndefined = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')

lemma fw_OCL_undef_3_lift3'_020:
  assumes f_def: f ≡ (lift3' lift_arg0 lift_arg2 lift_arg0) (λ x y. strictify (P x y))
  shows          f X Y OclUndefined = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')

end
```

B.3.4. Type Definition for Boolean

```
theory OCL_Boolean_type
imports
  $HOLOCL_HOME/src/OCL_Kernel
begin
```

This theory is the prototype for a basic type definition in OCL. The value types are defined by extending the basic HOL types with the bottom element ⊥.

```
types
      Boolean_0    = bool up
      'a Boolean   = ('a, Boolean_0) VAL
end
```

B.3.5. Type Definition for Integer

theory *OCL_Integer_type*
imports
 $*HOLOCL_HOME/src/OCL_Kernel*
begin

The value types are defined by extending the basic HOL types with the bottom element $\perp$.

types
 $Integer\_0 = int\ up$
 $'a\ Integer = ('a,\ Integer\_0)\ VAL$

translations
 $a\ Integer \leftharpoondown (a,\ Integer\_0)\ VAL$
end

B.3.6. Type Definition for Real

theory
OCL_Real_type
imports
 $*HOLOCL_HOME/src/OCL_Kernel*
 Complex_Main
begin

The value types are defined by extending the basic HOL types with the bottom element $\perp$.

types
 $Real\_0 = real\ up$
 $'a\ Real = ('a,\ Real\_0)\ VAL$

translations
 $a\ Real \leftharpoondown (a,\ Real\_0)\ VAL$
end

B.3.7. Type Definition for String

theory *OCL_String_type*
imports
 $*HOLOCL_HOME/src/OCL_Kernel*
begin

The value types are defined by extending the basic HOL types with the bottom element $\perp$.

types
  String_0 = *string up*
  'a String = (*'a*, *String_0*) *VAL*

translations
  a String ↽ (*a*, *String_0*) *VAL*
end

B.3.8. Type Definition for OclAny

theory *OCL_OclAny_type*
imports
  $*HOLOCL_HOME/src/Lifting*
begin typedecl
  oid

arities
  oid :: *bot*

datatype *OclAny_key* = *OclAny*

types *OclAny_type* = *OclAny_key* × *oid*

This definition of *OclAny_type* results in a "referential universe", i.e., a universe where a referential equality is present. A referential equality is the strong equality for "boxed types" (that is Real, Boolean, String) and an equality on the reference to a value. Since our universe construction is designed to mirror the subtype discipline of OCL, i.e., objects are records of a certain type, the reference to an object must be stored in the object itself in our model, namely at the first position of the record.

If OCL expressions refer to states, where all objects contain the reference to itself in the state, the logical equality co-incides with a referential equality.

types
  'a OclAny_ext = (*OclAny_type* × *'a up*)
  'a OclAny_0 = (*'a OclAny_ext*) *up*

Alternative we could define which would allow an additional 'exception' element with a non-strict behavior. This would model something like a null reference (or pointer) known from object-oriented programming languages.

translations
  a OclAny_0 <= (*type*) ((*OclAny_key* ∗ *oid*) ∗ *a up*)*up*

types (*'st*,*'a*)*OclAny* = (*'st*, *'a OclAny_0*) *VAL*

lemma [*simp*]: *DEF* (*obj*::((*'a*)*OclAny_0*)) ⟹
    (*fst*(*fst*(*drop*(*obj*))) = *OclAny_key.OclAny*)
  apply (*auto simp*: *DEF_def not_down_exists_lift*)
  apply (*case_tac a*)

```
  apply (assumption)
  done

end
```

B.3.9. Type Definition for Collections

```
theory OCL_Collection_type
imports
  $HOLOCL_HOME/src/OCL_Kernel
begin axclass
  collection < bot
end
```

B.3.10. Type Definition for Set

```
theory OCL_Set_type
imports
  $HOLOCL_HOME/src/library/collection/OCL_Collection_type
begin typedef 'a Set_0 = {X::('a::bot) set up.
                  (smash (λx X. DEF X ∧ x ∈ drop X) X) = X}
apply (rule_tac x=⊥ in exI)
  by (simp add: smash_def)

instance Set_0 :: (type) ord
  by intro_classes

arities Set_0 :: (type) collection

defs     UU_Set_def: ⊥ ≡ Abs_Set_0 down

end
```

B.3.11. Type Definition for Sequence

```
theory OCL_Sequence_type
imports
  $HOLOCL_HOME/src/library/collection/OCL_Collection_type
  List
begin
```

In contrast to other collections, the collection Sequence is indeed implemented by *finite* lists. Consequently, *size* is defined for all defined sequences.

```
typedef 'a Sequence_0 = {X::('a::bot) list up.
                         (smash (λ x X. DEF X ∧ x ∈ set ⌜X⌝) X) = X}
  by (rule_tac x=⊥ in exI, simp add: smash_def)
```

```
arities Sequence_0 :: (type) collection

defs    UU_Sequence_def: ⊥ ≡ Abs_Sequence_0 down
```

The HOL type constructor for representing OCL sequence types ist defined as follows:

```
types   ('st,'b) Sequence  = ('st, 'b Sequence_0) VAL


end
```

B.3.12. Type Definition for Bag

```
theory OCL_Bag_type
imports
  $HOLOCL_HOME/src/library/collection/OCL_Collection_type
  Multiset
begin typedef 'a Bag_0 = {X::('a::bot) multiset up.
                      (smash (λx X. DEF X & x :# drop X) X) = X}
apply (rule_tac x=⊥ in exI)
by (simp add: smash_def)


instance Bag_0 :: (type) ord
  by intro_classes

arities Bag_0 :: (type) bot
arities Bag_0 :: (type) collection

defs    UU_Bag_def: ⊥ ≡ Abs_Bag_0 down

types   ('st,'b) Bag  = ('st, 'b Bag_0) VAL

end
```

B.3.13. Type Definition for OrderedSet

```
theory OCL_OrderedSet_type
imports
  $HOLOCL_HOME/src/library/collection/OCL_Collection_type
  List
begin typedef 'a OrderedSet_0 = {X::('a::bot) list up.
                     (smash (λ x X. DEF X ∧ (distinct ⌜X⌝ ⟶ x ∈ set ⌜X⌝)) X) = X}
  by (rule_tac x=⊥ in exI, simp add: smash_def)


instance OrderedSet_0 :: (type) ord
```

```
  by intro_classes

arities  OrderedSet_0 :: (type) bot
arities  OrderedSet_0 :: (type) collection

defs     UU_OrderedSet_def: ⊥ ≡ Abs_OrderedSet_0 down

types       ('a,'b) OrderedSet  = ('a, 'b OrderedSet_0) VAL

end
```

B.3.14. The OCL Universe

```
theory OCL_Universe
imports
  Datatype
  $HOLOCL_HOME/src/library/basic/OCL_Boolean_type
  $HOLOCL_HOME/src/library/basic/OCL_String_type
  $HOLOCL_HOME/src/library/basic/OCL_Real_type
  $HOLOCL_HOME/src/library/basic/OCL_Integer_type
  $HOLOCL_HOME/src/library/basic/OCL_Undefined
  $HOLOCL_HOME/src/library/OclAny/$OCLANY/OCL_OclAny_type
  $HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_Set_type
  $HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_Bag_type
  $HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_OrderedSet_type
  $HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_Sequence_type
begin hide const class
```

Note, the definitions of `OclAny` differs, depending if HOL-OCL is was built with a referential universe or not.[2]

Foundations of the Universe Construction

The *universe* $\mathcal{U}$ is a family of *universe types* which comprise all value types (Real, Integer,String, Boolean, ...) and an extendible *class type representation* induced by a class hierarchy. The concepts of the universe construction has been presented in Chapt. 4.

The "initial" universe type $\mathcal{U}_0$ is built as sum of the built-in value types and the type OclAny (including all its extensions representing subtypes of OclAny.

```
types
  values  = Real_0 + Integer_0 + Boolean_0 + String_0
  'a U    = ('a OclAny_0) + values

types
```

[2]This manual describes HOL-OCL with referential universes.

$('a,'T)\ cl\_ext = ('T \times 'a\ up)$
$('a,'b,'T)\ ext = ('a,'T)cl\_ext + 'b$

The following setup of Isabelles syntax engine helps when presenting types corresponding to OCL types:

translations
a U <= (*type*) (*OclAny_0 a*) + (*Real_0* + *Integer_0* + *Boolean_0* + *String_0*)
Real_0 <= (*type*) *real up*
Integer_0 <= (*type*) *int up*
String_0 <= (*type*) *char list up*
Boolean_0 <= (*type*) *bool up*

Constants in and Operations on the Universe

The test operations for "beeing a Real" in a universe type were provided as follows:

constdefs
$sup :: (('a \times 'b\ )up) \Rightarrow 'a$
$sup\ obj \equiv fst((drop(obj)))$
$base :: ('a \times 'b\ up)\ up \Rightarrow 'b\ up$

base x ≡ *case x of lift* (*a*,*b*) => *b*
| *down* => *down*

constdefs
$refbase :: ('a \times 'b \times 'c\ up)\ up \Rightarrow 'c\ up$
refbase x ≡ *case x of lift* (*a*,(*b*,*c*)) => *c* | *down* => *down*

constdefs
$ref\_of :: ('a \times 'b \times 'c\ up)\ up \Rightarrow 'b$
$ref\_of\ x \equiv fst(snd(drop(x)))$

lemma *DEF_baseD* [*simp*]: $DEF(base(obj)) \Longrightarrow DEF(obj)$
apply (*simp add*: *DEF_def base_def not_down_exists_lift*)
apply (*cases obj*, *simp*)
apply (*auto*)
done
lemma *DEF_baseD2* [*simp*]: $DEF(base(obj)) \Longrightarrow DEF(snd(drop(obj)))$
apply (*simp add*: *DEF_def base_def not_down_exists_lift*)
apply (*cases obj*, *simp*)
apply (*auto*)
done
lemma *base_snd_drop*: $DEF(base(obj)) \Longrightarrow base(obj) = snd(drop(obj))$
apply (*simp add*: *DEF_def base_def not_down_exists_lift*)
apply (*cases obj*, *simp*)
apply (*auto*)
done

lemma *DEF_refbaseD* [*simp*]: $DEF(refbase(obj)) \Longrightarrow DEF(obj)$
  apply (*simp add*: *DEF_def refbase_def not_down_exists_lift*)
  apply (*cases obj*, *simp*)
  apply (*auto*)
  done
lemma *DEF_refbaseD2* [*simp*]: $DEF(refbase(obj)) \Longrightarrow DEF(snd(drop(obj)))$
  apply (*simp add*: *DEF_def refbase_def not_down_exists_lift*)
  apply (*cases obj*, *simp*)
  apply (*auto*)
  done
lemma *refbase_snd_drop*: $DEF(refbase(obj)) \Longrightarrow refbase(obj) = snd(snd(drop(obj)))$
  apply (*simp add*: *DEF_def refbase_def not_down_exists_lift*)
  apply (*cases obj*, *simp*)
  apply (*auto*)
  done

lemma *base_UU* [*simp*]: $base \perp = \perp$
  by (*simp add*: *base_def*)

lemma *base_down* [*simp*]: $base\ down = down$
  by (*simp add*: *base_def*)

lemma *base_snd_lift* [*simp*]: $base\ (lift(a,X)) = X$
  by (*simp add*: *base_def*)

lemma *refbase_UU* [*simp*]: $refbase \perp = \perp$
  by (*simp add*: *refbase_def*)

lemma *refbase_down* [*simp*]: $refbase\ down = down$
  by (*simp add*: *refbase_def*)

lemma *refbase_snd_lift* [*simp*]: $refbase\ (lift(a,(b,X))) = X$
  by (*simp add*: *refbase_def*)

types $'a\ OCLtype\ = 'a\ U\ set$

constdefs $noext\ :: 'a \Rightarrow ('a \times 'b\ up)$
  $noext\ x \equiv (x,\perp)$

lemma *fst_noext_id* [*simp*]: $fst(noext\ x)=x$

```
  apply (simp (no_asm) add: noext_def)
  done

lemma snd_noext_UU [simp]: snd(noext x)=⊥
  apply (simp (no_asm) add: noext_def)
done


end
```

B.4. Library

B.4.1. The OCL let expression

```
theory OCL_Let
imports
  $HOLOCL_HOME/src/Lifting
begin
```

The OCL let expression is a purely syntactical abbrevation (see [41, p. A-26]) therefore it can be directly mapped to the HOL let.

```
constdefs
  OclLet    :: [('a,'b) VAL, ('a,'b) VAL ⇒ ('a,'b::bot) VAL] ⇒ ('a,'b) VAL
 OclLet     ≡  Let

syntax
    _OclLet_std         :: [('a,'b) VAL,('a,'b) VAL ⇒ ('a,'b::bot) VAL] ⇒ ('a,'b) VAL
                        ((let (_)/ in (_)/end) 15)

syntax
    _OclLet_ascii         :: [('a,'b) VAL,('a,'b) VAL ⇒ ('a,'b::bot) VAL] ⇒ ('a,'b) VAL
                        ((let (_)/ in (_)/end) 15)


syntax (xsymbols)
    _OclLet_math         :: [('a,'b::bot) VAL] ⇒ ('a,'b) VAL
                        ((let (_)/ in (_)/end) 15)
end
```

B.4.2. OCL Boolean

```
theory OCL_Boolean
  imports
    $HOLOCL_HOME/src/OCL_Universe
    $HOLOCL_HOME/src/OCL_Let
begin
```

The Characteristic Set of the Type Boolean

Case distinction over a boolean

lemma *boolean_cases_sem*:
  assumes *undefC*: $x = \bot \Longrightarrow R$
  and *trueC*: $x = \llcorner True \lrcorner \Longrightarrow R$
  and *falseC*: $x = \llcorner False \lrcorner \Longrightarrow R$
  shows R
  apply(*rule disjE*)
  prefer *2* **apply**(*erule undefC*)
  prefer *2* **apply**(*rule disjE*)
  prefer *2* **apply**(*rotate_tac −1, erule trueC*)
  prefer *2* **apply**(*rotate_tac −1, erule falseC*)
  apply(*auto simp: not_down_exists_lift*)
  done

constdefs
  $is\_Boolean\_0 :: {}'\alpha\ U \Rightarrow bool$
  $is\_Boolean\_0 \equiv sum\_case\ (\lambda x.\ False)$
  $(sum\_case\ (\lambda x.\ False)$
  $(sum\_case\ (\lambda x.\ False)$
  $(sum\_case\ (\lambda x.\ True)(\lambda x.\ False))))$
  $get\_Boolean\_0 :: {}'\alpha\ U \Rightarrow Boolean\_0$
  $get\_Boolean\_0 \equiv sum\_case\ (\lambda x.\ \epsilon\ x.\ True)$
  $(sum\_case\ (\lambda x.\ \epsilon\ x.\ True)$
  $(sum\_case\ (\lambda x.\ \epsilon\ x.\ True)$
  $(sum\_case\ (\lambda x.\ x)(\lambda x.\ \epsilon\ x.\ True))))$
  $mk\_Boolean\_0 :: Boolean\_0 \Rightarrow {}'\alpha\ U$
  $mk\_Boolean\_0 \equiv Inr \circ Inr \circ Inr \circ Inl$

lemma *get_mk_Boolean_id_0*: $get\_Boolean\_0(mk\_Boolean\_0\ x) = x$
  apply (*simp add: get_Boolean_0_def mk_Boolean_0_def*)
  done
lemma *is_mk_Boolean_0*: $is\_Boolean\_0(mk\_Boolean\_0\ x) = True$
  apply (*simp add: is_Boolean_0_def mk_Boolean_0_def*)
  done
lemma *mk_get_Boolean_id_0*: $is\_Boolean\_0\ x \Longrightarrow mk\_Boolean\_0(get\_Boolean\_0\ x) = x$
  apply (*simp add: get_Boolean_0_def mk_Boolean_0_def is_Boolean_0_def*)
  apply (*case_tac x, simp, simp*)
  apply (*case_tac b, simp, simp*)
  apply (*case_tac ba, simp, simp*)
  apply (*case_tac bb, simp, simp*)
  done

constdefs
  $Boolean\_0 :: ({}'\tau,\ Boolean\_0\ Set\_0)\ VAL$
  $Boolean\_0 \equiv lift_0(Abs\_Set\_0\ \llcorner lift\ `\ UNIV \lrcorner)$

Semantics of Boolean expressions, i.e., the logical operators of the language. They build a Kleene-Logic,i.e., the usual algebraic laws hold. Moreover, all operators are context passing. This results in proof support based on extensive ternary case distinctions ($P(X)$ holds iff $P(\bot)$ and $P(\mathsf{T})$ and $P(\mathsf{F})$).

OCL Boolean expressions

The Core of the Language

These operators are defined via lifting principles from HOL (see [13]). A more textbook-like "definition" of OCL (following the standard) is derived in the theorem section.

constdefs

$$
\begin{array}{ll}
\mathit{OclIsDefined} & :: (\prime\tau,\prime b{::}bot)\ \mathit{VAL} \Rightarrow \prime\tau\ \mathit{Boolean} \\
\mathit{OclIsDefined} & \equiv \mathit{lift}_1(\mathit{lift} \circ \mathit{DEF})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{OclTrue} & :: \prime\tau\ \mathit{Boolean} \\
\mathit{OclTrue} & \equiv \mathit{lift}_0(\lfloor \mathit{True} \rfloor)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{OclFalse} & :: \prime\tau\ \mathit{Boolean} \\
\mathit{OclFalse} & \equiv \mathit{lift}_0(\lfloor \mathit{False} \rfloor)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{OclNot} & :: \prime\tau\ \mathit{Boolean} \Rightarrow \prime\tau\ \mathit{Boolean} \\
\mathit{OclNot} & \equiv \mathit{lift}_1(\mathit{strictify}(\mathit{lift} \circ \mathit{Not} \circ \mathit{drop}))
\end{array}
$$

constdefs

$$
\begin{array}{ll}
\mathit{OclAnd} & :: [\prime\tau\ \mathit{Boolean}, \prime\tau\ \mathit{Boolean}] \Rightarrow \prime\tau\ \mathit{Boolean} \\
\mathit{OclAnd} & \equiv \mathit{lift}_2(\lambda x\ y.\ \mathit{if}\ \mathit{DEF}\ x \\
& \quad \mathit{then}\ \mathit{if}\ \mathit{DEF}\ y\ \mathit{then}\ \lfloor(\ulcorner x \urcorner) \wedge (\ulcorner y \urcorner)\rfloor \\
& \qquad \mathit{else}\ (\mathit{if}\ \ulcorner x \urcorner\ \mathit{then}\ \bot\ \mathit{else}\ \lfloor \mathit{False} \rfloor) \\
& \quad \mathit{else}\ \mathit{if}\ \mathit{DEF}\ y \\
& \qquad \mathit{then}\ (\mathit{if}\ \ulcorner y \urcorner\ \mathit{then}\ \bot\ \mathit{else}\ \lfloor \mathit{False} \rfloor) \\
& \qquad \mathit{else}\ \bot)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{OclIf} & :: [\prime\tau\ \mathit{Boolean}, (\prime\tau,\prime\alpha{::}bot)\,\mathit{VAL}, (\prime\tau,\prime\alpha)\,\mathit{VAL}] \Rightarrow (\prime\tau,\prime\alpha)\,\mathit{VAL} \\
\mathit{OclIf} & \equiv \mathit{lift}_3(\lambda x\ y\ z.\ \mathit{strictify}(\lambda a.\ \mathit{if}\ \ulcorner a \urcorner\ \mathit{then}\ y\ \mathit{else}\ z)\ x)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{OclSand} & :: [\prime\tau\ \mathit{Boolean}, \prime\tau\ \mathit{Boolean}] \Rightarrow \prime\tau\ \mathit{Boolean} \\
\mathit{OclSand} & \equiv \mathit{lift}_2(\mathit{strictify}(\lambda\ x.\ \mathit{strictify}(\lambda\ y. \\
& \qquad \lfloor(\ \ulcorner x \urcorner\ ) \wedge (\ \ulcorner y \urcorner\ )\rfloor\ )))
\end{array}
$$

The following statements represent the technical setup of Isabelle's syntax engine for this core-language. All these technical setups can be ignored in first reasing.

syntax
```
_OclIsDefined_std :: ('τ,'α::bot) VAL ⇒ 'τ Boolean
                     (->IsDefined() (_) [60]60)
_OclIsUndefined_std::('τ,'α::bot) VAL ⇒ 'τ Boolean
                     (->oclIsUndefined() (_) [60]60)
_OclTrue_std      :: 'τ Boolean
                     (true)
_OclFalse_std     :: 'τ Boolean
                     (false)
_OclNot_std       :: 'τ Boolean ⇒ 'τ Boolean
                     (not (_) [59]59)
_OclAnd_std       :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                     ((_ and/ _)[57,58]57)
_OclIf_std        :: ['τ Boolean,('τ,'b) VAL,('τ,'b::bot) VAL] ⇒ ('τ,'b) VAL
                     ((if (_)/ then (_)/else (_)/endif) 15)
_OclSand_std      :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                     ((_ sand/ _)[57,58]57)
```

syntax
```
_OclIsDefined_ascii :: ('τ,'α::bot) VAL ⇒ 'τ Boolean
                     (isdef (_) [60]60)
_OclIsUndefined_ascii:: ('τ,'α::bot) VAL ⇒ 'τ Boolean
                     (isundef (_) [60]60)
_OclTrue_ascii      :: 'τ Boolean
                     (TRUE)
_OclFalse_ascii     :: 'τ Boolean
                     (FALSE)
_OclNot_ascii       :: 'τ Boolean ⇒ 'τ Boolean
                     (not (_) [59]59)

_OclAnd_ascii       :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                     ((_ and _)[57,58]57)
_OclIf_ascii        :: ['τ Boolean,('τ,'b) VAL,('τ,'b::bot) VAL] ⇒ ('τ,'b) VAL
                     ((if (_)/ then (_)/else (_)/endif) 15)
_OclSand_ascii      :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                     ((_ sand _)[57,58]57)
```

syntax (xsymbols)
```
_OclIsDefined_math:: ('τ,'α::bot) VAL ⇒ 'τ Boolean
                     (∂ (_) [60]60)
_OclIsUndefined_math:: ('τ,'α::bot) VAL ⇒ 'τ Boolean
                     (∂̸ (_) [60]60)
_OclTrue_math     :: 'τ Boolean
                     (T)
_OclFalse_math    :: 'τ Boolean
                     (F)
_OclNot_math      :: 'τ Boolean ⇒ 'τ Boolean
                     (¬ (_) [59]59)
```

_OclAnd_math :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
((_ ∧/ _)[57,58]57)
_OclIf_math :: ['τ Boolean,('τ,'α::bot) VAL,('τ,'α) VAL] ⇒ ('τ,'α) VAL
((if (_)/ then (_)/else (_)/endif) 15)
_OclSand_math :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
((_ $\dot{\wedge}$/ _)[57,58]57)

Is OCL Boolean Faithful to the OCL Standard?

In this section, we present background material and the proof work for the section 4.5.

constdefs
semfun :: 'α ⇒ 'α (*Sem* [[(_)]])
Sem[[x]] ≡ x

We show for our first strict operation in OCL, the `not` operator, that it is in fact an instance of the standards definition scheme:

lemma *not_faithfully_represented*:
Sem[[¬ *X*]] γ = (*if Sem*[[*X*]] γ ≠ ⊥
then $\llcorner$¬ $\ulcorner$ *Sem*[[*X*]]γ $\urcorner\lrcorner$
else ⊥)
by(*simp add*:*OclNot_def*
DEF_def o_def
lift0_def lift1_def lift2_def
semfun_def)

The proof is trivial and canonic: it consists of the unfolding of all combinator definitions (they are just abbreviations of re-occuring patterns in the textbook style definitions used to make these patterns explicit!) and the semantic function which is a mere syntactic marker in our context.

The definition of basic operations for the special case of boolean operations is done by a truth table [41, table A 2, page A-12]. We represent this table for the elementary "and" case as following:

lemma *and_faithfully_represented*:
Sem[[*X* ∧ *Y*]] γ =
(*if* *Sem*[[*X*]] γ = $\llcorner$*False*$\lrcorner$ ∧ *Sem*[[*Y*]] γ = $\llcorner$*False*$\lrcorner$ *then* $\llcorner$*False*$\lrcorner$
else if Sem[[*X*]] γ = $\llcorner$*False*$\lrcorner$ ∧ *Sem*[[*Y*]] γ = $\llcorner$*True*$\lrcorner$ *then* $\llcorner$*False*$\lrcorner$
else if Sem[[*X*]] γ = $\llcorner$*True*$\lrcorner$ ∧ *Sem*[[*Y*]] γ = $\llcorner$*False*$\lrcorner$ *then* $\llcorner$*False*$\lrcorner$
else if Sem[[*X*]] γ = $\llcorner$*True*$\lrcorner$ ∧ *Sem*[[*Y*]] γ = $\llcorner$*True*$\lrcorner$ *then* $\llcorner$*True*$\lrcorner$
else if Sem[[*X*]] γ = $\llcorner$*False*$\lrcorner$ ∧ *Sem*[[*Y*]] γ = ⊥ *then* $\llcorner$*False*$\lrcorner$
else if Sem[[*X*]] γ = $\llcorner$*True*$\lrcorner$ ∧ *Sem*[[*Y*]] γ = ⊥ *then* ⊥
else if Sem[[*X*]] γ = ⊥ ∧ *Sem*[[*Y*]] γ = $\llcorner$*False*$\lrcorner$ *then* $\llcorner$*False*$\lrcorner$
else if Sem[[*X*]] γ = ⊥ ∧ *Sem*[[*Y*]] γ = $\llcorner$*True*$\lrcorner$ *then* ⊥
else if Sem[[*X*]] γ = ⊥ ∧ *Sem*[[*Y*]] γ = ⊥ *then* ⊥
else ⊥)

apply(*simp add:OclAnd_def*
DEF_def o_def strictify_def
lift0_def lift1_def lift2_def
semfun_def)

apply (*cases X* γ = *down*)
apply (*cases Y* γ = *down*)
apply (*simp_all add: not_down_exists_lift*)
by *auto*

For the operators `or`, `xor`, `and implies`, we refrain from an explicit proof of conformance with [41, table A 2, page A-12]. Instead, we preferred to define these operators as usual in terms of `and` and `not`. However, the subsequent lemmas *OclUndefined__or__True* represent another way to directly check that our definitions meet the standard.

The Extended Core of the Language

constdefs
OclOr :: $['\tau\ Boolean, '\tau\ Boolean] \Rightarrow '\tau\ Boolean$
OclOr S T $\equiv \neg\ (\neg\ S \wedge \neg\ T)$

OclSor :: $['\tau\ Boolean, '\tau\ Boolean] \Rightarrow '\tau\ Boolean$
OclSor S T $\equiv \neg\ (\neg\ S\ \dot{\wedge}\ \neg\ T)$

Alternative definitions for the use with ocl_setup_op:

lemma *OclOr_alt_def*: $OclOr \equiv lift_2\ (\lambda\ x\ y.$
$(if\ (DEF\ x)$
$then\ (if\ (DEF\ y)\ then\ \llcorner(\ulcorner x\urcorner \vee \ulcorner y\urcorner)\lrcorner$
$else\ (if\ \ulcorner x\urcorner\ then\ \llcorner True\lrcorner\ else\ \bot))$
$else\ (if\ (DEF\ y)\ then\ (if\ \ulcorner y\urcorner\ then\ \llcorner True\lrcorner\ else\ \bot)\ else\ \bot)))$
apply(*rule eq_reflection*, (*rule ext*)+)
by(*simp add: OclOr_def OclAnd_def OclNot_def ss_lifting*)

lemma *OclSor_alt_def*:
$OclSor \equiv lift_2\ (strictify\ (\lambda\ x.\ strictify\ (\lambda\ y. \llcorner(\ulcorner x\urcorner \vee \ulcorner y\urcorner)\lrcorner)))$
apply(*rule eq_reflection*, (*rule ext*)+)
by(*simp add: OclSor_def OclNot_def OclSand_def ss_lifting'*)

syntax
_OclOr_std :: $['\tau\ Boolean, '\tau\ Boolean] \Rightarrow '\tau\ Boolean$
((_ `or`/ _) [53,54]53)
_OclSor_std :: $['\tau\ Boolean, '\tau\ Boolean] \Rightarrow '\tau\ Boolean$
((_ `sor`/ _) [53,54]53)

syntax
_OclOr_ascii :: $['\tau\ Boolean, '\tau\ Boolean] \Rightarrow '\tau\ Boolean$
((_ *or*/ _) [53,54]53)

$\_OclSor\_ascii \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$
$((\_\ sor/\ \_)\ [53,54]53)$

syntax
$\_OclOr\_math \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$
$((\_\ \vee/\ \_)\ [53,54]53)$
$\_OclSor\_math \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$

$((\_\ \dot{\vee}/\ \_)\ [53,54]53)$

constdefs
$OclXor \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$
$OclXor\ S\ T \quad \equiv (S \vee T) \wedge \neg\ (S \wedge T)$

$OclImplies \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$
$OclImplies\ S\ T \equiv\ (\neg\ S) \vee T$

alternative definitions for the use with ocl_setup_op

lemma $OclXor\_alt\_def$: $OclXor \equiv lift_2\ (\lambda\ x\ y.$
$(if\ (DEF\ x)$
$then\ (if\ (DEF\ y)\ then\ \llcorner(\ulcorner x \urcorner \neq \ulcorner y \urcorner)\lrcorner\ else\ \bot)$
$else\ \bot))$
apply(*rule eq_reflection*, (*rule ext*)+)
by(*simp add*: *OclXor_def OclOr_alt_def OclAnd_def OclNot_def*
ss_lifting)

lemma $OclImplies\_alt\_def$: $OclImplies \equiv lift_2\ (\lambda\ x\ y.$
$(if\ (DEF\ x)$
$then\ (if\ (DEF\ y)\ then\ \llcorner(\ulcorner x \urcorner \longrightarrow \ulcorner y \urcorner)\lrcorner$
$else\ (if\ \ulcorner x \urcorner\ then\ \bot\ else\ \llcorner True \lrcorner))$
$else\ (if\ (DEF\ y)\ then\ (if\ \ulcorner y \urcorner\ then\ \llcorner True \lrcorner\ else\ \bot)\ else\ \bot)))$
apply(*rule eq_reflection*, (*rule ext*)+)
by(*simp add*: *OclImplies_def OclAnd_def OclNot_def*
OclOr_alt_def OclIsDefined_def ss_lifting)

There is an inconsistency between the normative part [41, Chapter 11] and the informative [41, Appendix A] the standard. For reasons discussed below, we chose the `implies` version defined by [41, Table A-2, page A-12] and define the other version later.

The truth for `implies` does not fulfill the normative requirement.

constdefs
$OclSxor \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$
$OclSxor\ S\ T \quad \equiv (S\ \dot{\vee}\ T)\ \dot{\wedge}\ \neg\ (S\ \dot{\wedge}\ T)$

$OclSimplies \quad :: [{'\tau}\ Boolean, {'\tau}\ Boolean] \Rightarrow {'\tau}\ Boolean$
$OclSimplies\ S\ T \equiv (\neg\ S)\ \dot{\vee}\ T$

Alternative definitions for the use with ocl_setup_op

lemma $OclSxor\_alt\_def$:
$OclSxor \equiv lift_2\ (strictify\ (\lambda\ x.\ strictify\ (\lambda\ y.\llcorner(\ulcorner x \urcorner \neq \ulcorner y \urcorner)\lrcorner)))$

```
  apply(rule eq_reflection, (rule ext)+)
  by(auto simp: OclSxor_def OclSor_def OclNot_def OclSand_def
             ss_lifting')

lemma OclSimplies_alt_def:
  OclSimplies ≡ lift₂ (strictify (λ x. strictify (λ y.⌊(⌈x⌉ ⟶ ⌈y⌉)⌋)))
  apply(rule eq_reflection, (rule ext)+)
  by(simp add: OclSimplies_def OclSxor_def OclNot_def OclSand_def
            OclSor_def ss_lifting')


syntax
   _OclXor_std      :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ xor/ _) [55,56]55)
   _OclImplies_std  :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ implies/ _)[53,52]52)
   _OclSxor_std      :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ sxor/ _) [55,56]55)
   _OclSimplies_std  :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ simplies/ _)[53,52]52)

syntax
   _OclXor_ascii     :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ xor/ _) [55,56]55)
   _OclImplies_ascii:: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ implies/ _)[53,52]52)
   _OclSxor_ascii    :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ sxor/ _) [55,56]55)
   _OclSimplies_ascii:: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ simplies/ _)[53,52]52)


syntax
   _OclXor_math        :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ ⊕/ _) [55,56]55)
   _OclImplies_math :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ ⟶/ _)[53,52]52)
   _OclSxor_math       :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ ⊕̇/ _) [55,56]55)
   _OclSimplies_math :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
                       ((_ ⟶̇/ _)[53,52]52)
```

An `implies` variant for three-valued logics introduced by Hähnle [18]:

```
constdefs
  OclImplies1      :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
 OclImplies1 S T ≡ ∂ S ∨ ¬ S ∨ T
```

An `implies` variant for three-valued logics introduced by Hennicker et al [23]. This variant is also the version actually used in the normative part of the standard.

constdefs
  OclImplies2 :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
  OclImplies2 S T ≡ (¬ S) ∨ (S ∧ T)

Alternative definitions for the use with ocl_setup_op

lemma *OclImplies1_alt_def*: *OclImplies1* ≡ *lift*$_2$ (λ x y.
    (if (DEF x)
     then (if ⌜x⌝ then y else ⌞True⌟)
     else ⌞True⌟))
  apply(*rule eq_reflection*, (*rule ext*)+)
  by(*simp add*: *OclImplies1_def OclAnd_def OclNot_def*
        OclOr_alt_def OclIsDefined_def ss_lifting)

lemma *OclImplies2_alt_def*: *OclImplies2* ≡ *lift*$_2$ (λ x y.
    (if (DEF x)
     then (if ⌜x⌝ then y else ⌞True⌟)
     else ⊥))
  apply(*rule eq_reflection*, (*rule ext*)+)
  by(*simp add*: *OclImplies2_def OclAnd_def OclNot_def*
        OclOr_alt_def OclIsDefined_def ss_lifting)

syntax
  _OclImplies1_std :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ implies1/ _)[53,52]52)
  _OclImplies2_std :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ implies2/ _)[53,52]52)

syntax
  _OclImplies1_ascii:: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ implies1/ _)[53,52]52)
  _OclImplies2_ascii:: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ implies2/ _)[53,52]52)

syntax
  _OclImplies1_math :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ $\xrightarrow{1}$/ _)[53,52]52)
  _OclImplies2_math :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ $\xrightarrow{2}$/ _)[53,52]52)

syntax
  _OclImplies1_std :: 'τ Boolean ⇒ 'τ Boolean
              (implies1 (_) [59]59)
  _OclImplies2_std :: ['τ Boolean, 'τ Boolean] ⇒ 'τ Boolean
              ((_ implies2/ _)[57,58]57)
  _OclIf_std   :: ['τ Boolean,('τ,'α::bot) VAL,('τ,'α::bot) VAL] ⇒ ('τ,'α) VAL

((if (_)/ then (_)/else (_)/endif) 15)

Setup of the Basics

lemma *OclIsDefined_vs_DEF*: ∂ *x* = $\top$ $\Longrightarrow$ *DEF* (*x St*)
apply (*simp add*: *OclIsDefined_def lift1_def o_def OclTrue_def lift0_def*)
apply (*drule fun_cong* [*of _ _ St*])
apply (*auto*)
done

lemma *DEF_vs_OclIsDefined*:
assumes *1*: ($\bigwedge$ *St*. *DEF* (*x St*))
shows $\top$ = ∂ *x*
apply (*rule ext*)
apply (*simp add*: *OclIsDefined_def lift1_def o_def OclTrue_def lift0_def*)
apply (*rule 1*)
done

ML_setup ⟪
```
val OclUndefined_def = get_def (theory OCL_Undefined) OclUndefined
```
⟫
ML ⟪
```
val lift3_def = get_def (theory Lifting) lift3
val lift2_def = get_def (theory Lifting) lift2
val lift1_def = get_def (theory Lifting) lift1
val lift0_def = get_def (theory Lifting) lift0
val cp_def = get_def (theory Lifting) cp
```
⟫

ML ⟪

```
val strictify_def = get_def (theory Lifting) strictify
val isStrict_def  = get_def (theory Lifting)  isStrict

val DEF_def = get_def  (theory Lifting)  DEF
val not_down_exists_lift = thm not_down_exists_lift
val up_split = thm up.split
val cp_lift0_fw = thm cp_lift0_fw
val cp_lift1_fw = thm cp_lift1_fw
val cp_lift2_fw = thm cp_lift2_fw
val cp_lift3_fw = thm cp_lift3_fw
val DEF_strictify_DEF_args  = thm DEF_strictify_DEF_args
val DEF_strictify_DEF_args2 = thm DEF_strictify_DEF_args2
val lift1b_undef = thm lift1b_undef
val lift2_undef1a= thm lift2_undef1a
val lift2_undef2a= thm lift2_undef2a
val cp_compose2  = thm cp_compose2
val UU_fun_def   = get_def  (theory Lifting)  UU_fun
```

⟫

ML ⟪
```
    val OclIsDefined_def= thm OclIsDefined_def
    val OclTrue_def     = thm OclTrue_def
    val OclFalse_def    = thm OclFalse_def
    val OclNot_def      = thm OclNot_def
    val OclAnd_def      = thm OclAnd_def
    val OclIf_def       = thm OclIf_def
    val OclOr_def       = thm OCL_Boolean.OclOr_def
    val OclXor_def      = thm OCL_Boolean.OclXor_def
    val OclImplies_def  = thm OCL_Boolean.OclImplies_def
    val OclImplies1_def = thm OCL_Boolean.OclImplies1_def
    val OclImplies2_def = thm OCL_Boolean.OclImplies2_def
```
⟫

The Basic Laws for the Boolean Operators

ML_setup ⟪
```
fun prover1 (goal,prems,name) =
    let val thm = prove_goalw (the_context()) (DEF_def::strictify_def::o_def::
                               lift0_def::lift1_def::lift2_def::lift3_def
                               ::prems) goal
                 (fn _ => [rtac ext 1,
                           ALLGOALS(asm_full_simp_tac
                                    (simpset()
                                     addsplits [split_if,up_split])),
                           ALLGOALS(asm_full_simp_tac
                                    (simpset()
                                     addsimps [not_down_exists_lift])),
                           auto_tac(claset(),simpset())
                          ])
    in bind_thm(name,thm);
       Addsimps[thm]
    end;
```
⟫

ML_setup ⟪
```
map prover1
[(∂ ⊥ = F, [OclIsDefined_def,OclTrue_def,OclFalse_def,OclUndefined_def],
            OclIsDefined__undef),
 (∂ T = T, [OclIsDefined_def,OclTrue_def,OclFalse_def,OclUndefined_def],
            OclIsDefined__True),
 (∂ F = T, [OclIsDefined_def,OclTrue_def,OclFalse_def,OclUndefined_def],
            OclIsDefined__False),
 (¬ ⊥ = ⊥, [OclNot_def,OclTrue_def,OclFalse_def,OclUndefined_def],
            not__undef),
 (¬ T = F, [OclNot_def,OclTrue_def,OclFalse_def,OclUndefined_def],
            not__True),
```

```
 (¬ F = T, [OclNot_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              not__False)];

⟫
```

This shows the following facts: $\partial \perp = \mathsf{F}$

$\partial\ \mathsf{T} = \mathsf{T}$

$\partial\ \mathsf{F} = \mathsf{T}$

$\neg \perp = \perp$

$\neg\ \mathsf{T} = \mathsf{F}$

$\neg\ \mathsf{F} = \mathsf{T}$

...which also prove the complience with Table A.2 in the standard 2.0.

```
ML ⟪
map prover1
[(⊥ ∧ ⊥ = ⊥, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              undef__and__undef),
 (⊥ ∧ T = ⊥, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              undef__and__True),
 (⊥ ∧ F = F, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              undef__and__False),
 (T ∧ ⊥ = ⊥, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              True__and__undef),
 (T ∧ T = T, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              True__and__True),
 (T ∧ F = F, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              True__and__False),
 (F ∧ ⊥ = F, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              False__and__undef),
 (F ∧ T = F, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              False__and__True),
 (F ∧ F = F, [OclAnd_def,OclTrue_def,OclFalse_def,OclUndefined_def],
              False__and__False)];
⟫
```

```
ML ⟪
map prover1
[(⊥ ∨ ⊥ = ⊥, [OclOr_def], OclUndefined__or__undef),
 (⊥ ∨ T = T, [OclOr_def], OclUndefined__or__True),
 (⊥ ∨ F = ⊥, [OclOr_def], OclUndefined__or__False),
 (T ∨ ⊥ = T, [OclOr_def], True__or__undef),
 (T ∨ T = T, [OclOr_def], True__or__True),
 (T ∨ F = T, [OclOr_def], True__or__False),
 (F ∨ ⊥ = ⊥, [OclOr_def], False__or__undef),
```

($\mathsf{F} \vee \mathsf{T} = \mathsf{T}$, [*OclOr_def*], *False__or__True*),
($\mathsf{F} \vee \mathsf{F} = \mathsf{F}$, [*OclOr_def*], *False__or__False*)];
⟫

ML ⟪
map prover1
[($\bot \oplus \bot = \bot$, [*OclXor_def*], *OclUndefined__xor__undef*),
($\bot \oplus \mathsf{T} = \bot$, [*OclXor_def*], *OclUndefined__xor__True*),
($\bot \oplus \mathsf{F} = \bot$, [*OclXor_def*], *OclUndefined__xor__False*),
($\mathsf{T} \oplus \bot = \bot$, [*OclXor_def*], *True__xor__undef*),
($\mathsf{T} \oplus \mathsf{T} = \mathsf{F}$, [*OclXor_def*], *True__xor__True*),
($\mathsf{T} \oplus \mathsf{F} = \mathsf{T}$, [*OclXor_def*], *True__xor__False*),
($\mathsf{F} \oplus \bot = \bot$, [*OclXor_def*], *OclUndefined__xor__undef*),
($\mathsf{F} \oplus \mathsf{T} = \mathsf{T}$, [*OclXor_def*], *OclUndefined__xor__True*),
($\mathsf{F} \oplus \mathsf{F} = \mathsf{F}$, [*OclXor_def*], *OclUndefined__xor__False*)];
⟫

ML ⟪
map prover1
[($\bot \longrightarrow \bot = \bot$,[*OclImplies_def*], *OclUndefined__implies__undef2*),
($\bot \longrightarrow \mathsf{T} = \mathsf{T}$,[*OclImplies_def*], *OclUndefined__implies__True2*),
($\bot \longrightarrow \mathsf{F} = \bot$,[*OclImplies_def*], *OclUndefined__implies__FALSE2*),
($\mathsf{T} \longrightarrow \bot = \bot$,[*OclImplies_def*], *True__implies__undef2*),
($\mathsf{T} \longrightarrow \mathsf{T} = \mathsf{T}$,[*OclImplies_def*], *True__implies__True2*),
($\mathsf{T} \longrightarrow \mathsf{F} = \mathsf{F}$,[*OclImplies_def*], *True__implies__False2*),
($\mathsf{F} \longrightarrow \bot = \mathsf{T}$,[*OclImplies_def*], *FALSE__implies__undef2*),
($\mathsf{F} \longrightarrow \mathsf{T} = \mathsf{T}$,[*OclImplies_def*], *False__implies__True2*),
($\mathsf{F} \longrightarrow \mathsf{F} = \mathsf{T}$,[*OclImplies_def*], *False__implies__False2*)];
⟫

ML ⟪
map prover1
[($\bot \xrightarrow{1} \bot = \mathsf{T}$,[*OclImplies1_def*], *OclUndefined__implies1__undef1*),
($\bot \xrightarrow{1} \mathsf{T} = \mathsf{T}$,[*OclImplies1_def*], *OclUndefined__implies1__True1*),
($\bot \xrightarrow{1} \mathsf{F} = \mathsf{T}$,[*OclImplies1_def*], *OclUndefined__implies1__False1*),
($\mathsf{T} \xrightarrow{1} \bot = \bot$,[*OclImplies1_def*], *True__implies1__undef1*),
($\mathsf{T} \xrightarrow{1} \mathsf{T} = \mathsf{T}$,[*OclImplies1_def*], *True__implies1__True1*),
($\mathsf{T} \xrightarrow{1} \mathsf{F} = \mathsf{F}$,[*OclImplies1_def*], *True__implies1__False1*),
($\mathsf{F} \xrightarrow{1} \bot = \mathsf{T}$,[*OclImplies1_def*], *False__implies1__undef1*),
($\mathsf{F} \xrightarrow{1} \mathsf{T} = \mathsf{T}$,[*OclImplies1_def*], *False__implies1__undef1*),
($\mathsf{F} \xrightarrow{1} \mathsf{F} = \mathsf{T}$,[*OclImplies1_def*], *False__implies1__undef1*)];
⟫

ML ⟪
map prover1

$[(\bot \xrightarrow{2} \bot = \bot, [OclImplies2\_def], OclUndefined\_\_implies2\_\_undef),$
$(\bot \xrightarrow{2} \mathsf{T} = \bot, [OclImplies2\_def], OclUndefined\_\_implies2\_\_True),$
$(\bot \xrightarrow{2} \mathsf{F} = \bot, [OclImplies2\_def], OclUndefined\_\_implies2\_\_FALSE),$
$(\mathsf{T} \xrightarrow{2} \bot = \bot, [OclImplies2\_def], True\_\_implies2\_\_undef),$
$(\mathsf{T} \xrightarrow{2} \mathsf{T} = \mathsf{T}, [OclImplies2\_def], True\_\_implies2\_\_True),$
$(\mathsf{T} \xrightarrow{2} \mathsf{F} = \mathsf{F}, [OclImplies2\_def], True\_\_implies2\_\_False),$
$(\mathsf{F} \xrightarrow{2} \bot = \mathsf{T}, [OclImplies2\_def], FALSE\_\_implies2\_\_undef),$
$(\mathsf{F} \xrightarrow{2} \mathsf{T} = \mathsf{T}, [OclImplies2\_def], False\_\_implies2\_\_True),$
$(\mathsf{F} \xrightarrow{2} \mathsf{F} = \mathsf{T}, [OclImplies2\_def], False\_\_implies2\_\_False)];$
⟫

ML ⟪
map prover1
$[((\textit{if } \bot \textit{ then } Y \textit{ else } Z \textit{ endif}) = \bot,$
$\quad [OclIf\_def, OclUndefined\_def], if\_undef),$
$((\textit{if } \mathsf{T} \textit{ then } Y \textit{ else } Z \textit{ endif}) = \backslash$
$\quad \backslash(Y::('\tau,'\alpha::bot)\,VAL),$
$\quad [OclIf\_def, OclTrue\_def], if\_TRUE),$
$((\textit{if } \mathsf{F} \textit{ then } Y \textit{ else } Z \textit{ endif}) = \backslash$
$\quad \backslash(Z::('\tau,'\alpha::bot)\,VAL),$
$\quad [OclIf\_def, OclFalse\_def], if\_FALSE)];$
⟫

All operators are contextpassing

lemmas *cp_undef = OclUndefined_def [THEN cp_lift0_fw, standard]*

lemmas *cp_OclIsDefined = OclIsDefined_def [THEN cp_lift1_fw, standard]*
lemmas *cp_not = OclNot_def [THEN cp_lift1_fw, standard]*
lemmas *cp_and = OclAnd_def [THEN cp_lift2_fw, standard]*
lemmas *cp_if = OclIf_def [THEN cp_lift3_fw, standard]*

declare *cp_undef [simp,intro!] cp_OclIsDefined [simp, intro!]*
cp_not [simp, intro!] cp_and [simp, intro!]
cp_if [simp, intro!]

ML ⟪
```
fun prover2 (goal,prems,name) =
    let val thm = prove_goalw (the_context()) (prems) goal
                  (fn _ => [Fast_tac 1])
    in bind_thm(name,thm);
       Addsimps[thm];
       AddSIs [thm]
    end;
```
⟫

⟫

ML ⟪
map prover2
$[(\bigwedge P.\ [\![\ cp\ P;cp\ P']\!] \Longrightarrow cp(\lambda X.\ P\ X \vee P'\ X),\ [OclOr\_def],\quad cp\_or),$
$(\bigwedge P.\ [\![cp\ P;cp\ P']\!] \Longrightarrow cp(\lambda X.\ P\ X \oplus P'\ X),\ [OclXor\_def],\quad cp\_xor),$
$(\bigwedge P.\ [\![cp\ P;cp\ P']\!] \Longrightarrow cp(\lambda X.\ P\ X \longrightarrow P'\ X),[OclImplies\_def],\ cp\_implies),$
$(\bigwedge P.\ [\![cp\ P;cp\ P']\!] \Longrightarrow cp(\lambda X.\ P\ X \xrightarrow{1} P'\ X),\ [OclImplies1\_def],cp\_implies1),$
$(\bigwedge P.\ [\![cp\ P;cp\ P']\!] \Longrightarrow cp(\lambda X.\ P\ X \xrightarrow{2} P'\ X),\ [OclImplies2\_def],cp\_implies2)]$
⟫

This shows the following facts: $[\![cp\ ?P;\ cp\ ?P']\!] \Longrightarrow cp\ (\lambda X.\ ?P\ X \vee ?P'\ X)$ $[\![cp\ ?P;\ cp\ ?P']\!] \Longrightarrow cp\ (\lambda X.\ ?P\ X \oplus ?P'\ X)$ $[\![cp\ ?P;\ cp\ ?P']\!] \Longrightarrow cp\ (\lambda X.\ ?P\ X \longrightarrow ?P'\ X)$ $[\![cp\ ?P;\ cp\ ?P']\!] \Longrightarrow cp\ (\lambda X.\ ?P\ X \xrightarrow{1} ?P'\ X)$ $[\![cp\ ?P;\ cp\ ?P']\!] \Longrightarrow cp\ (\lambda X.\ ?P\ X \xrightarrow{2} ?P'\ X)$

Proof Technique: High-level Trichometry on context passing contexts

lemma *base_distinct*:
$X\ \tau = \bot\ \tau \vee X\ \tau = \mathsf{T}\ \tau \vee X\ \tau = \mathsf{F}\ \tau$
  apply (*simp add: OclUndefined_def lift0_def OclTrue_def OclFalse_def*)
  apply (*auto*)
  apply (*simp add: not_down_exists_lift*)
  apply (*erule exE*)
  apply (*auto*)
  done
  ML_setup ⟪ *val base_distinct = thm base_distinct* ⟫

ML_setup ⟪
```
val prems = goalw (the_context()) []
                  (X τ ~= F τ) = (X τ = ⊥ τ | X τ = T τ);
    by(cut_facts_tac[base_distinct] 1);
    by(Step_tac 1);
    ba 1;
    by(ALLGOALS(asm_full_simp_tac ((simpset()) addsimps
                             [OclFalse_def,OclTrue_def,
                              lift0_def,OclUndefined_def])));
    qeddistinct_1;

val prems = goalw (the_context()) []
                  (X τ ~= T τ) = (X τ = ⊥ τ | X τ = F τ);
    by(cut_facts_tac[base_distinct] 1);
    by(Step_tac 1);
    ba 1;
    by(ALLGOALS(asm_full_simp_tac ((simpset()) addsimps
                             [OclFalse_def,OclTrue_def,lift0_def,OclUndefined_def])));
    qeddistinct_2;
```

```
val prems = goalw (the_context()) []
                  (X τ ~= ⊥) = (X τ = T τ | X τ = F τ);
    by(cut_facts_tac[base_distinct] 1);
    by(Step_tac 1);
    by(ALLGOALS(asm_full_simp_tac ((simpset()) addsimps
                                   [OclFalse_def,OclTrue_def,lift0_def,OclUndefined_def])));
    by(Fast_tac 1);
    qeddistinct_3;

⟫
```

lemma *cp_distinct_core*:
⟦ *cp P*; *cp P'*;
$P \perp \tau = P' \perp \tau$;
$P\ \mathsf{T}\ \tau\ = P'\ \mathsf{T}\ \tau$;
$P\ \mathsf{F}\ \tau = P'\ \mathsf{F}\ \tau$
⟧ $\Longrightarrow P\ X\ \tau = P'\ X\ \tau$
apply (*unfold cp_def*)
apply (*erule exE*)
apply (*erule exE*)
apply (*rotate_tac* −*2*)
apply *simp*
apply (*cut_tac* $X = X$ **and** $\tau = \tau$ **in** *base_distinct*)
apply (*erule disjE*)
apply (*erule_tac* [*2*] *disjE*)
apply (*auto*)
done
ML ⟪ *val cp_distinct_core = thm cp_distinct_core* ⟫

lemma *cp_distinct_core_P*:
⟦ *cp P*; *cp P'*;
$X\ \tau = \perp\ \tau \Longrightarrow P \perp \tau = P' \perp \tau$;
$X\ \tau = \mathsf{T}\ \tau\ \Longrightarrow P\ \mathsf{T}\ \tau\ = P'\ \mathsf{T}\ \tau$;
$X\ \tau = \mathsf{F}\ \tau \Longrightarrow P\ \mathsf{F}\ \tau = P'\ \mathsf{F}\ \tau$
⟧ $\Longrightarrow P\ X\ \tau = P'\ X\ \tau$
apply (*unfold cp_def*)
apply (*erule exE*)
apply (*erule exE*)
apply (*rotate_tac* −*2*)
apply *simp*
apply (*cut_tac* $X = X$ **and** $\tau = \tau$ **in** *base_distinct*)
apply (*erule disjE*)
apply (*erule_tac* [*2*] *disjE*)
apply (*auto*)
done

ML ⟪ *val cp_distinct_core_P = thm cp_distinct_core_P* ⟫

lemma *local_validity_propagation1*:
$[\![$ $X\ \tau = \bot\ \tau$;
$P\ \bot\ \tau = P'\ \bot\ \tau$;
$cp\ P$; $cp\ P'$
$]\!] \Longrightarrow (P{::}('\tau)Boolean \Rightarrow ('\tau,'\alpha)VAL)\ X\ \tau = P'\ X\ \tau$

apply (*rule_tac* $P = P$ **and** $P' = P'$ **in** *cp_distinct_core_P*)
apply *auto*
apply (*simp_all add*: *OclTrue_def OclFalse_def OclUndefined_def lift0_def*)
done

ML ⟪ *val local_validity_propagation1 = thm local_validity_propagation1* ⟫

lemma *local_validity_propagation2*:
$[\![$ $X\ \tau = \mathsf{T}\ \tau$;
$P\ \mathsf{T}\ \tau = P'\ \mathsf{T}\ \tau$;
$cp\ P$; $cp\ P'$
$]\!] \Longrightarrow (P{::}('\tau)Boolean \Rightarrow ('\tau,'\alpha)VAL)\ X\ \tau = P'\ X\ \tau$

apply (*rule_tac* $P = P$ **and** $P' = P'$ **in** *cp_distinct_core_P*)
apply *auto*
apply (*simp_all add*: *OclTrue_def OclFalse_def OclUndefined_def lift0_def*)
done
ML ⟪ *val local_validity_propagation2 = thm local_validity_propagation2* ⟫

lemma *local_validity_propagation3*:
$[\![$ $X\ \tau = \mathsf{F}\ \tau$;
$P\ \mathsf{F}\ \tau = P'\ \mathsf{F}\ \tau$;
$cp\ P$; $cp\ P'$
$]\!] \Longrightarrow (P{::}('\tau)Boolean \Rightarrow ('\tau,'\alpha)VAL)\ X\ \tau = P'\ X\ \tau$

apply (*rule_tac* $P = P$ **and** $P' = P'$ **in** *cp_distinct_core_P*)
apply *auto*
apply (*simp_all add*: *OclTrue_def OclFalse_def OclUndefined_def lift0_def*)
done

ML ⟪ *val local_validity_propagation3 = thm local_validity_propagation3* ⟫

lemma *cp_distinct*:
$[\![$ $cp\ P$; $cp\ P'$;
$P\ \bot = P'\ \bot$;
$P\ \mathsf{T} = P'\ \mathsf{T}$;
$P\ \mathsf{F} = P'\ \mathsf{F}$
$]\!] \Longrightarrow P\ X = P'\ X$
apply (*rule ext*)
apply (*drule_tac* $f = P\ \bot$ **and** $x = x$ **in** *fun_cong*)
apply (*drule_tac* $f = P\ \mathsf{T}$ **and** $x = x$ **in** *fun_cong*)

apply ($drule\_tac\ f = P$ F **and** $x = x$ **in** fun_cong)
apply ($rule\ cp\_distinct\_core$)
back
apply ($auto$)
done

ML ⟪ $val\ cp\_distinct = thm\ cp\_distinct$ ⟫

lemma $cp\_distinct1$:
  ⟦ ∂ $X =$ T; $cp\ P$; $cp\ P'$;
    P T $= P'$ T;
    P F $= P'$ F
  ⟧ $\Longrightarrow P\ X = P'\ X$
apply ($unfold\ cp\_def\ OclIsDefined\_def\ o\_def\ lift1\_def$)
apply ($rule\ ext$)
apply ($drule\_tac\ f = P$ T **and** $x = x$ **in** fun_cong)
apply ($drule\_tac\ f = P$ F **and** $x = x$ **in** fun_cong)
apply ($drule\_tac\ f = \lambda c.\ \llcorner DEF\ (X\ c)\lrcorner$ **and** $x = x$ **in** fun_cong)
apply ($erule\ exE$)
apply ($erule\ exE$)
apply ($rotate\_tac\ -2$)
apply $simp$
apply ($cut\_tac\ X = X$ **and** $\tau = x$ **in** $base\_distinct$)
apply ($erule\ disjE$)
apply ($erule\_tac\ [2]\ disjE$)
apply ($simp$)
apply ($simp\ add$: $DEF\_def\ lift0\_def\ OclUndefined\_def\ OclTrue\_def$)
apply ($auto$)
done

ML ⟪ $val\ cp\_distinct1 = thm\ cp\_distinct1$ ⟫

lemma $cp\_distinct2$:
  ⟦ $cp\ P$; $cp\ P'$;
    $\bigwedge X.\ X =$ ⊥ $\Longrightarrow P\ X = P'\ X$;
    $\bigwedge X.\ X =$ T $\Longrightarrow P\ X = P'\ X$;
    $\bigwedge X.\ X =$ F $\Longrightarrow P\ X = P'\ X$
  ⟧ $\Longrightarrow P\ X = P'\ X$

apply ($rule\ cp\_distinct$)
apply $auto$
done

ML ⟪ $val\ cp\_distinct2 = thm\ cp\_distinct2$ ⟫

The lemmas for *st_transissibility* allow for a new, high-level proof style based on case-distictions *avoiding* unfolding the definition of `and`, ...

ML ⟪

```
fun Ocl_case_tac str n=
    EVERY[(eres_inst_tac [(X,str)] cp_distinct1 n) ORELSE
          (res_inst_tac [(X,str)] cp_distinct n),
           TRY(Fast_tac n), TRY(Fast_tac n),
            Auto_tac ]
⟫
```

ML ⟪

```
fun prover3 (goal,dists,b,name) =
    let val thm = prove_goalw (the_context()) [] goal
                (fn _ => map (fn x => Ocl_case_tac x 1) dists)
    in bind_thm(name,thm);
       if b then Addsimps[thm] else ()
    end;
⟫
```

ML ⟪

```
map prover3
[(¬(¬ X) = X,     [X],true,not_not) ,
 (X ∧ X = X,      [X],true, and_idem),
 (F ∧ X = F,      [X],true,and_false_dominant_1),
 (X ∧ F = F,      [X],true,and_false_dominant_2),
 (T ∧ X = X,      [X],true,and_true_neutral_1),
 (X ∧ T = X,      [X],true,and_true_neutral_2),
 (X ∧ Y = Y∧ X,   [X,Y],false,and_commute),
 (X ∧ (Y ∧ Z) = (X ∧ Y)∧ Z,[X,Y,Z],false,and_assoc)]
⟫
```

ML ⟪

```
map prover3
[(X ∨ X = X,[X],true, or_idem),
 (X ∨ T = T,[X],true, or_true_dominant_1),
 (T ∨ X = T,[X],true, or_true_dominant_2),
 (X ∨ F = X,[X],true, or_false_neutral_1),
 (F ∨ X = X,[X],true, or_false_neutral_2),
 (X ∨ Y = Y ∨ X,[X,Y],false, or_commute),
 ((X ∨ Y) ∨ Z = X ∨ (Y ∨ Z),[X,Y,Z],false,or_assoc)];
⟫
```

ML ⟪

```
map prover3
[((X ∨ Y)∧ Z = (X ∧ Z) ∨ (Y ∧ Z),[Z,Y,X,X],
  false,and_or_distrib1),
 (Z ∧ (X ∨ Y) = (Z ∧ X) ∨ (Z ∧ Y),[Z,Y,X,X],
  false,and_or_distrib2),
 ((X ∧ Y) ∨ Z = (X ∨ Z) ∧ (Y ∨ Z),[Z,Y,X,X],
```

```
 false,or_and_distrib1),
 (Z ∨ (X ∧ Y) = (Z ∨ X) ∧ (Z ∨ Y),[Z,Y,X,X],
 false,or_and_distrib2)];
⟫
```

ML ⟪
```
map prover3
[((X ∧ Y) = ¬(¬(X) ∨ ¬(Y)),[X,Y],false,de_morgan),
 (¬(X ∧ Y) = (¬(X) ∨ ¬(Y)),[X,Y],false,de_morgan1),
 (¬(X ∨ Y) = (¬(X) ∧ ¬(Y)),[X,Y],false,de_morgan2)];
⟫
```

ML ⟪
```
map prover3
[(X ⊕ T = ¬ X,   [X], true, xor_true_inverse_1),
 (T ⊕ X = ¬ X,   [X], true, xor_true_inverse_2),
 (X ⊕ F = X,     [X], true, xor_false_neutral_1),
 (F ⊕ X = X,     [X], true, xor_false_neutral_2),
 (X ⊕ Y = Y ⊕ X, [X,Y],false, xor_commute),
 ((X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z),
                   [X,Y,Z,Z,Y,Z], false, xor_assoc)];
⟫
```

We will not include the or-generating rewrite rules into the rewriter since they do not "simplify" the situation. As rules, they are important, of course.

ML ⟪
```
map prover3
[((X ⟶ F) = ¬ X,[X], true, implies_false),
 ((X ⟶ T) = T, [X], true, implies_true1),
 ((X ⟶ ⊥) = ¬ X ∨ ⊥, [X], false, implies_undef1),
 ((F ⟶ X) = T, [X], true, implies_true2),
 ((T ⟶ X) = X, [X], true, implies_idem),
 ((⊥ ⟶ X) = X ∨ ⊥, [X], false, implies_undef2),
 ((X ⟶ X) = ¬ X ∨ X, [X], false, implies_idem1),
 (X ⟶ (Y ∧ Z) = ((X ⟶ Y) ∧ (X ⟶ Z)),
                   [X,Y,Z,Z],false, implies_andI),
 (X ⟶ (Y ∨ Z) = ((X ⟶ Y) ∨ (X ⟶ Z)),
                   [X,Y,Z,Z],false, implies_orI),
 (((X ∧ Y) ⟶ Z) = X ⟶ (Y ⟶ Z),
                   [Z,Y,X,Y,X],false, implies_andE),
 (((X ∨ Y) ⟶ Z) = (X ⟶ Z) ∧ (Y ⟶ Z),
                   [Z,Y,X,Y,X,X,X,X,Y],false,
                   implies_orE),
 (X ⟶ (Y ⟶ Z) = Y ⟶ (X ⟶ Z),
   [Z,Y,X,Y,X],false, implies_commute)]
⟫
```

ML ⟪

map prover3
$[(\bot \xrightarrow{2} X = \bot, [X],$ *true*, *implies2_undef*),
$(\top \xrightarrow{2} X = X, [X],$ *true*, *implies2_idem*),
$(\mathsf{F} \xrightarrow{2} X = \top, [X],$ *true*, *implies2_true2*),
$((X \xrightarrow{2} X) = \neg X \vee X, [X],$ *false*, *implies2_idem1*),
$(X \xrightarrow{2} \bot = \neg X \vee \bot, [X],$ *false*, *implies2_undef2*),
$(X \xrightarrow{2} \mathsf{F} = \neg X, [X],$ *true*, *implies2_false*),
$(X \xrightarrow{2} \top = \neg X \vee X, [X],$ *false*, *implies2_true1*)
] ⟫

ML⟪
map prover3
$[(X \xrightarrow{2} (Y \wedge Z) = ((X \xrightarrow{2} Y) \wedge (X \xrightarrow{2} Z)),$
$[X]$,*false*,
implies2_andI),
$(X \xrightarrow{2} (Y \vee Z) = ((X \xrightarrow{2} Y) \vee (X \xrightarrow{2} Z)),$
$[X]$,*false*,
implies2_orI)
$(* (((X \wedge Y) \xrightarrow{2} Z) = X \xrightarrow{2} (Y \xrightarrow{2} Z),$
$[X, Y]$,*false*, *implies2_andE*) $*$)
$(* (((X \vee Y) \xrightarrow{2} Z) = (X \xrightarrow{2} Z) \wedge (Y \xrightarrow{2} Z),$
$[X, Y]$,*false*, *implies2_orE*) $*$)
$(* (X \xrightarrow{2} (Y \xrightarrow{2} Z) = Y \xrightarrow{2} (X \xrightarrow{2} Z),$
$[X, Y]$,*false*, *implies2_commute*) $*$)]
⟫

The properties in comments, crucial for deduction, could not be proven. In fact, they do not hold due to the counter-examples:

lemma $[\![X=\bot; Y=\top; Z=\top]\!] \Longrightarrow$
$((X \vee Y) \xrightarrow{2} Z) \neq (X \xrightarrow{2} Z) \wedge (Y \xrightarrow{2} Z)$
apply(*simp*, *auto simp*: *lift0_def OclUndefined_def OclTrue_def*)
apply(*drule fun_cong*, *simp*)
done

lemma $[\![X=\bot; Y=\mathsf{F}]\!] \Longrightarrow$
$((X \wedge Y) \xrightarrow{2} Z) \neq (X \xrightarrow{2} (Y \xrightarrow{2} Z))$
apply(*simp*, *auto simp*: *lift0_def OclUndefined_def OclTrue_def*)
apply(*drule fun_cong*, *simp*)
done

lemma $[\![X=\bot; Y=\mathsf{F}]\!] \Longrightarrow$
$X \xrightarrow{2} (Y \xrightarrow{2} Z) \neq Y \xrightarrow{2} (X \xrightarrow{2} Z)$
apply(*simp*, *auto simp*: *lift0_def OclUndefined_def OclTrue_def*)
apply(*drule fun_cong*, *simp*)
done

All these higher logical arrangement rules *do not hold* for the standards definition $?S \xrightarrow{2} ?T \equiv \neg\ ?S \vee\ ?S\ \wedge\ ?T$ of the implication, in contrast to the "classical definition" $?S \longrightarrow\ ?T \equiv \neg\ ?S \vee\ ?T$. Although the standards choice is feasible, in the light of these dramatic algebraic deficiencies, we qualify it as glitch from the deduction point of view and suggest to apply the definition used in the appendix.

Definition for implies inappropriate

For the completeness of our presentation, we show the properties of `implies` introduced by Hähnle [18]:

ML ⟨⟨
map prover3
$[((X \xrightarrow{1} X) = \mathsf{T}, [X],\ \mathit{true},\ \mathit{implies1\_idem}),$
$((\bot \xrightarrow{1} X) = \mathsf{T}, [X],\ \mathit{true},\ \mathit{implies1\_undef}),$
$(\bigwedge X.\ \partial(X) = \mathsf{T} \Longrightarrow (X \xrightarrow{1} \mathsf{F}) = \neg\ X,$
$\quad [X],\ \mathit{true},\ \mathit{implies1\_false}),$
$((X \xrightarrow{1} \mathsf{T}) = \mathsf{T}, [X],\ \mathit{true},\ \mathit{implies1\_true1}),$
$((\mathsf{F} \xrightarrow{1} X) = \mathsf{T}, [X],\ \mathit{true},\ \mathit{implies1\_true2}),$
$((\mathsf{T} \xrightarrow{1} X) = X, [X],\ \mathit{true},\ \mathit{implies1\_idem2}),$
$(((X \wedge Y) \xrightarrow{1} Z) = X \xrightarrow{1} (Y \xrightarrow{1} Z),$
$\quad [Z,Y,X,Y,X], \mathit{false},$
$\quad \mathit{implies1\_andE}),$
$(((X \vee Y) \xrightarrow{1} Z) = (X \xrightarrow{1} Z) \wedge (Y \xrightarrow{1} Z),$
$\quad [Z,Y,X,Y,X,X,X,X,Y], \mathit{false},$
$\quad \mathit{implies1\_orE}),$
$(X \xrightarrow{1} (Y \xrightarrow{1} Z) = Y \xrightarrow{1} (X \xrightarrow{1} Z),$
$\quad [X], \mathit{false},\ \mathit{implies1\_commute}),$
$(X \xrightarrow{1} (Y \wedge Z) = ((X \xrightarrow{1} Y) \wedge (X \xrightarrow{1} Z)),$
$\quad [X], \mathit{false},\ \mathit{implies1\_andI}),$
$(X \xrightarrow{1} (Y \vee Z) = ((X \xrightarrow{1} Y) \vee (X \xrightarrow{1} Z)),$
$\quad [X], \mathit{false},\ \mathit{implies1\_orI})]$
⟩⟩

ML ⟨⟨
map prover3
$[(\neg(\partial\ X) \vee X \vee \neg\ X = \mathsf{T},\quad [X], \mathit{false}, \mathit{kleenical0}),$
$(\bigwedge X.\ \partial\ X = \mathsf{T} \Longrightarrow (X \vee \neg\ X) = \mathsf{T},\ [X], \mathit{false}, \mathit{kleenical1}),$
$(\bigwedge X.\ \partial\ X = \mathsf{T} \Longrightarrow (\neg\ X \vee X) = \mathsf{T},\ [X], \mathit{false}, \mathit{kleenical2}),$
$((\partial X \wedge X \wedge \neg\ X) = \mathsf{F},\quad [X], \mathit{false}, \mathit{absurd0}),$
$(\bigwedge X.\ \partial\ X = \mathsf{T} \Longrightarrow (X \wedge \neg\ X) = \mathsf{F},\ [X], \mathit{false}, \mathit{absurd1}),$
$(\bigwedge X.\ \partial\ X = \mathsf{T} \Longrightarrow (\neg\ X \wedge X) = \mathsf{F},\ [X], \mathit{false}, \mathit{absurd2}),$
$(\neg(\partial\ X) \vee (X \longrightarrow X) = \mathsf{T},\quad [X], \mathit{false}, \mathit{implies\_assume0}),$
$(\bigwedge X.\ \partial\ X = \mathsf{T} \Longrightarrow (X \longrightarrow X) = \mathsf{T},\ [X], \mathit{false}, \mathit{implies\_assume})]$
⟩⟩

ML ⟨⟨
map prover3
$[(\bigwedge X.\ \partial\ X = \mathsf{T} \Longrightarrow \partial(\neg\ X) = \mathsf{T},\ [X], \mathit{true},\ \mathit{isDefined\_not}),$

```
(⋀Z. ⟦ ∂ X = ⊤;∂ Y = ⊤ ⟧ ⟹  ∂(X ∧ Y) = ⊤,
                              [X],true, isDefined_and),
(⋀Z. ⟦ ∂ X = ⊤;∂ Y = ⊤ ⟧ ⟹  ∂(X ∨ Y) = ⊤,
                              [X],true, isDefined_or),
(⋀Z. ⟦ ∂ X = ⊤;∂ Y = ⊤ ⟧ ⟹  ∂(X ⊕ Y) = ⊤,
                              [X],true, isDefined_xor),
(⋀Z. ⟦ ∂ X = ⊤;∂ Y = ⊤ ⟧ ⟹  ∂(X ⟶ Y) = ⊤,
                              [X],true, isDefined_implies),
(⋀Z. ⟦ ∂ X = ⊤;∂ Y = ⊤ ⟧ ⟹  ∂(X ⟶² Y) = ⊤,
                              [X],true, isDefined_implies2),
(⋀A. ⟦ ∂ X = ⊤;∂ Y = ⊤; ∂ Z = ⊤⟧ ⟹ \
 \ ∂(if X then Y else Z endif) = ⊤,
                              [X],true, isDefined_if)]

⟫
```

```
ML ⟪
map prover3
[((if X then (Y::'τ=>('b::bot)) else Z endif) = \
 \(if ¬ X then Z else Y endif),    [X],false,if_swap),
 (⋀Z. ∂ X = ⊤ ⟹ (if X then Y else Z endif) = ((X ⟶ Y) ∧ (¬ X ⟶ Z)),
                                   [X],false,if_to_implies)]
⟫
```

The Laws for the *is_def* Operator

Some basic laws.

```
lemma isDefined_idem[simp] : ∂(∂ X) = ⊤
  by(simp add: OclIsDefined_def lift0_def
               lift1_def OclUndefined_def o_def OclTrue_def)
```

```
ML ⟪
map prover3
[(∂(∂ X ∧ X) = ⊤,   [X],true, isDefined_cong1),
 (∂(∂ X ∧ ¬ X) = ⊤, [X],true, isDefined_cong2),
 (∂(X ∧ ∂ X) = ⊤,   [X],true, isDefined_cong3),
 (∂(¬ X ∧ ∂ X) = ⊤, [X],true, isDefined_cong4)];

⟫
```

Some generic laws for the *is_def* operator

This section is essentially a rephrasing of the theory Lifting in the newly established terminology of this theory.

```
lemma is_isdef_lift0: ∂(lift₀(⌞ c ⌟)) = ⊤
 apply (unfold OclIsDefined_def lift0_def lift1_def OclTrue_def o_def)
 apply (rule ext)
```

apply (*auto*)
done

lemma *lift1_strictify_is_isdef*:
∂(*lift*$_1$ (*strictify* ($\lambda x.$ $\lfloor f\ x\rfloor$)) *X*) = ∂ *X*
apply (*unfold OclIsDefined_def lift1_def OclUndefined_def o_def*)
apply (*rule ext*)
apply *auto*
apply (*drule DEF_strictify_DEF_args*)
apply *auto*
done

lemma *lift2_strictify_is_isdef*:
∂(*lift*$_2$ (*strictify*($\lambda x.$ *strictify*($\lambda y.$ $\lfloor f\ x\ y\rfloor$))) *X Y*) = (∂ *X* $\wedge$ ∂ *Y*)
apply (*unfold OclIsDefined_def lift1_def lift2_def OclUndefined_def OclAnd_def o_def*)
apply (*rule ext*)
apply *auto*
apply (*drule DEF_strictify_DEF_args2*)
apply (*drule_tac* [*2*] *DEF_strictify_DEF_args2*)
apply *auto*
done

declare *lift1_strictify_is_isdef* [*simp*] *lift2_strictify_is_isdef* [*simp*]

lemma *lift1b_undef_fw*:
$f \equiv$ *lift*$_1$ (*strictify*(*g*)) $\Longrightarrow$ $f\ \bot = \bot$
apply (*unfold OclUndefined_def lift0_def*)
apply (*tactic* ⟪ *fold_tac* [*UU_fun_def*] ⟫)
apply (*rule lift1b_undef*)
done

lemma *lift2_undef1a_fw*:
$f \equiv$ *lift*$_2$ (*strictify*($\lambda x.$ *strictify*(*g x*))) $\Longrightarrow$ $f\ \bot\ (X::'\tau \Rightarrow '\alpha::bot) = \bot$
apply (*unfold OclUndefined_def lift0_def*)
apply (*tactic* ⟪ *fold_tac* [*UU_fun_def*] ⟫)
apply (*rule lift2_undef1a*)
done

lemma *lift2_undef2a_fw*:
$f \equiv$ *lift*$_2$ (*strictify*($\lambda x.$ *strictify*(*g x*))) $\Longrightarrow$ $f(X::'\tau \Rightarrow '\alpha::bot)\ \bot = \bot$
apply (*unfold OclUndefined_def lift0_def*)
apply (*tactic* ⟪ *fold_tac* [*UU_fun_def*] ⟫)
apply (*rule lift2_undef2a*)
done

ML ⟪ *val lift2_undef2a_fw = thm lift2_undef2a_fw* ⟫

lemma *lift1_strict_is_isdef_fw*:
$f \equiv lift_1\ (strictify\ (\lambda x.\ \llcorner g \ulcorner x \urcorner \lrcorner)) \Longrightarrow\ \partial (f\ X) = \partial\ X$
by(*auto*)

lemma *lift1_strictify_is_isdef_fw*:
$f \equiv lift_1\ (strictify(\lambda x.\ \llcorner g\ x \lrcorner)) \Longrightarrow\ \partial (f\ X) = \partial\ X$
by(*auto*)

lemma *lift2_strict_is_isdef_fw*:
$f \equiv lift_2(strictify(\lambda x.\ strictify(\lambda y.\ \llcorner g \ulcorner x \urcorner\ \ulcorner y \urcorner \lrcorner)))\Longrightarrow \partial (f\ X\ Y) = (\partial\ X \wedge \partial\ Y)$
by(*auto*)

lemma *lift2_strictify_is_isdef_fw*:
$f \equiv lift_2\ (strictify\ (\lambda x.\ strictify(\lambda y.\ \llcorner g\ x\ y \lrcorner))) \Longrightarrow \partial (f\ X\ Y) = (\partial\ X \wedge \partial\ Y)$
by(*auto*)

Definedness reasoning for the new lifting construction

lemma *fw_OCL_is_isdef_lift1'_0*:
  assumes *f_def*: $f \equiv (lift1'\ lift\_arg0)\ (strictify\ (\lambda\ x.\ \llcorner g\ x \lrcorner))$
  shows $\partial (f\ X) = \partial\ X$
  by(*rule ext, simp add: f_def OclIsDefined_def ss_lifting'*)

lemma *fw_OCL_is_isdef_lift2'_00*:
  assumes *f_def*: $f \equiv (lift2'\ lift\_arg0\ lift\_arg0)$
$(strictify\ (\lambda\ x.\ (strictify\ (\lambda\ y.\ \llcorner g\ x\ y \lrcorner))))$
  shows $\partial (f\ X\ Y) = (\partial\ X) \wedge (\partial\ Y)$
  by(*rule ext, simp add: f_def OclIsDefined_def OclAnd_def ss_lifting'*)

register rules for ocl_setup_op

ML ⟪
```
val _ = let

  fun is_isdef_rules rname =
    let val rname' = fw_OCL_is_isdef ^ rname;
    in  RULE
          {rule      = (rname', thm rname'),
           simp_tac  = fn _ => fn x => x,
           is_atomic = fn _ => true,
           register  = default_register_simponly,
           name_gen  = default_namegen OCL_is_isdef_
          }
    end

in
  add_setup_rules (
    (map is_isdef_rules [_lift1'_0, _lift2'_00])
```

)
end
⟫

Some basic laws for thenot operator

lemma *not_injective* : $[\![\neg X = \neg Y]\!] \Longrightarrow X = Y$
by(*drule_tac f* = $\lambda x. \neg x$ **in** *arg_cong*, *simp*)

lemma *not_bijective*[*simp*]: $(\neg X = \neg Y) = (X = Y)$
by(*auto intro*!: *not_injective*)

Characterizations of logical operator definedness

lemma *isDefined_notD*: $\partial(\neg X) = \mathsf{T} \Longrightarrow \partial(X)=\mathsf{T}$
by (*simp add*: *OclNot_def o_def lift1_strictify_is_isdef*)

ML_setup ⟪
map prover3
$[(\partial(\neg X) = (\partial X),$ $[X]$,*true*, *isDefined_notD0*),
$(\partial(X \wedge Y) = ((\partial X \wedge \partial Y) \vee (\partial X \wedge \neg X) \vee (\partial Y \wedge \neg Y)),$
$[X, Y, Y]$,*false*, *isDefined_andD0*),
$(\partial(X \vee Y) = ((\partial X \wedge \partial Y) \vee (\partial X \wedge X) \vee (\partial Y \wedge Y)),$
$[X, Y, Y]$,*false*, *isDefined_orD0*),
$(\partial(X \oplus Y) = (\partial X \wedge \partial Y),[X, Y]$,*false*, *isDefined_xorD0*),
$(\partial(X \longrightarrow Y)=((\partial X \wedge \partial Y) \vee (\partial X \wedge \neg X) \vee (\partial Y \wedge Y)),$
$[X, Y, Y]$,*false*, *isDefined_impliesD0*),
$(\partial(X \xrightarrow{2} Y)= \partial X \wedge (\neg(X \wedge \neg \partial Y)),$
$[X]$,*false*, *isDefined_implies2D0*),
$(\partial($*if* X *then* Y *else* Z *endif*$) = (\partial X \wedge ((X \wedge \partial Y) \vee (\neg X \wedge \partial Z))),$
$[X]$,*false*, *isDefined_ifD0*)];
⟫

The set of rules above is of fundamental importance for the forward-reasoning of definedness expressions—it allows for computing for any core logic expression a DNF composed of boolean literals and text *is_def*(X) terms (where X is either a Boolean OCL atom (variable) or a non Boolean expression (`if`)). If the latter are composed by strict OCL operators, it can be further atomized to *is_def*(X) atoms where X occurs in the expression.

Some generic laws for the Ocl if operator

lemma *if_distrib_strict*:
$[\![$ *cp P*; *isStrict P* $]\!] \Longrightarrow$
$P($*if* X *then* $(Y{::}('\tau,'\alpha{::}bot)\,VAL)$ *else* Z *endif*$)$ =
(*if* X *then* $((P\ Y){::}('\tau,'\beta{::}bot)\,VAL)$ *else* $P\ Z$ *endif*)
apply(*rule_tac X* = X **in** *cp_distinct*)
apply(*rule_tac P* = P **in** *cp_compose2*)
apply *auto*

```
  apply(simp add: UU_fun_def OclUndefined_def
                  isStrict_def lift0_def)
  done

lemma if_distrib2 :
  ⟦ cp P; ∂ X = T ⟧ ⟹
   P(if X then (Y::('τ,'α::bot) VAL) else Z endif) =
    (if X then ((P Y)::('τ,'β::bot) VAL) else P Z endif)
   apply(simp add: cp_def OclIf_def lift3_def,
         erule exE, rule ext)
   apply(simp add: OclIsDefined_def DEF_def OclTrue_def
                   lift0_def lift1_def o_def)
   apply(drule_tac x=τ in fun_cong, simp)
   apply(simp only: not_down_exists_lift, erule exE)
   apply simp
   done
```

Final Setup

```
lemma and_left_idem: P ∧ (P ∧ Q) = P ∧ Q
 apply (subst and_assoc )
 apply (subst and_idem )
 apply (auto)
 done
 ML ⟪ val and_left_idem = thm and_left_idem ⟫

lemma or_left_idem:  P ∨ (P ∨ Q) = P ∨ Q
 apply (subst or_assoc [symmetric])
 apply (subst or_idem)
 apply (auto)
 done
 ML ⟪ val or_left_idem = thm or_left_idem  ⟫

lemma and_left_assoc: X ∧ (Y ∧ Z) = Y ∧ (X ∧ Z)
 apply (simp add:and_assoc)
 apply (subst and_commute)
 apply simp
 done

lemma or_left_assoc: X ∨ (Y ∨ Z) = Y ∨ (X ∨ Z)
 apply (simp add:or_assoc[symmetric])
 apply (subst or_commute)
 apply simp
 done
```

Now we prepare a setup for Associativity, commutativity, and idempotency (ACI) rewriting of the boolean operators: These are:

$$a \circ b = b \circ a \tag{B.1}$$
$$(a \circ b) \circ c = a \circ (b \circ c) \tag{B.2}$$
$$a \circ (b \circ c) = b \circ (a \circ c) \tag{B.3}$$
$$a \circ a = a \tag{B.4}$$
$$a \circ (a \circ b) = a \circ b \tag{B.5}$$

lemmas *OCL_logic_ACI =*
and_commute and_assoc[symmetric]
and_left_assoc and_idem and_left_idem
or_commute or_assoc
or_left_assoc or_idem
or_left_idem

Boolean Relations

OCL Equalities

In the non-referential universe the following defines the two default equalities. In the referntial universe, these definitions define the referntial equalities.

constdefs *OclStrongEq* :: $[('\tau,'\alpha{::}bot)\,VAL,('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
OclStrongEq $\equiv$ $lift_2$ $(\lambda\ x\ y.\ \llcorner x = y\lrcorner\ )$

OclStrictEq :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
OclStrictEq $\equiv$ $lift_2$ $(strictify(\lambda\ x.$
$strictify(\lambda\ y.\ \llcorner x = y\lrcorner\ )))$

consts
OclStrongValueEq :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
OclStrictValueEq :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
OclStrongDeepValueEq :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
OclStrictDeepValueEq :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*

syntax
_OclStrongEq_std :: $[('\tau,'\alpha{::}bot)\,VAL,('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
((1_/ = _) [63,64]63)
_OclStrictEq_std :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
((1_/==_) [63,64]63)
_OclStrongValueEq_std :: $[('\tau,'\alpha{::}bot)\,VAL,('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
((1_/ ~= _) [63,64]63)
_OclStrictValueEq_std :: $[('\tau,'\alpha{::}bot)\,VAL,\ ('\tau,'\alpha)\,VAL] \Rightarrow {}'\tau$ *Boolean*
((1_/~==_) [63,64]63)

$\_OclStrongDeepValueEq\_std$ :: $[('\tau,'\alpha::bot)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \sim=\sim \_)\ [63,64]63)$
$\_OclStrictDeepValueEq\_std$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/\sim==\sim\_)\ [63,64]63)$

syntax
$\_OclStrongEq\_ascii$::$[('\tau,'\alpha::bot)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/$ ‘=‘ $\_)\ [63,64]63)$
$\_OclStrictEq\_ascii$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/$‘==‘$\_)\ [63,64]63)$
$\_OclStrongValueEq\_ascii$::$[('\tau,'\alpha::bot)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/$ ‘~=‘ $\_)\ [63,64]63)$
$\_OclStrictValueEq\_ascii$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/$‘~==‘$\_)\ [63,64]63)$
$\_OclStrongDeepValueEq\_ascii$::$[('\tau,'b::bot)\ VAL,('\tau,'b)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/$ ‘~=~‘ $\_)\ [63,64]63)$
$\_OclStrictDeepValueEq\_ascii$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/$‘~==~‘$\_)\ [63,64]63)$

syntax (*xsymbols*)
$\_OclStrongEq\_math$:: $[('\tau,'\alpha::bot)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \triangleq \_)\ [63,64]63)$
$\_OclStrictEq\_math$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \doteq \_)\ [63,64]63)$
$\_OclStrongValueEq\_math$:: $[('\tau,'\alpha::bot)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \overset{\triangle}{\simeq} \_)\ [63,64]63)$
$\_OclStrictValueEq\_math$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \dot{\simeq} \_)\ [63,64]63)$
$\_OclStrongDeepValueEq\_math$:: $[('\tau,'\alpha::bot)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \overset{\triangle}{\approxeq} \_)\ [63,64]63)$
$\_OclStrictDeepValueEq\_math$ :: $[('\tau,'\alpha::bot)\ VAL, ('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$((1\_/ \dot{\approxeq} \_)\ [63,64]63)$**syntax**
$@OclNeq :: ['\tau\ Boolean,'\tau\ Boolean] \Rightarrow '\tau\ Boolean$
$(\_ /$‘<>‘ $\_\ \ [51,52]51)$

translations
X ‘<>‘ $Y == \neg(X \doteq Y)$

Is HOL-OCL Faithful to the OCL standard (Part II)?

lemma *strictEq_faithfully_represented*:
$Sem[[X \doteq Y]]\ \gamma = (if\quad Sem[[X]]\ \gamma \neq \bot \land Sem[[Y]]\ \gamma \neq \bot$
$then\ \llcorner\ulcorner Sem[[X]]\gamma\urcorner = \ulcorner Sem[[Y]]\gamma\urcorner\lrcorner$
$else\ \ \bot)$
apply(*simp add:OclStrictEq_def*
DEF_def o_def

lift0_def lift1_def lift2_def
semfun_def)
apply (*simp_all add*: *not_down_exists_lift*, *auto*)
done

This operator—the first binary operator of a strict operation—represents directly the definition scheme as imposed by [41, Definition A. 16]. The operators for Set, Bag, Integer and Real will follow this scheme.

end

B.4.3. OCL Logic Core

theory *OCL_Logic_core*
imports *$HOLOCL_HOME/src/library/basic/OCL_Boolean*
uses *$HOLOCL_HOME/src/kernel_ext/isabelle_kernel_patch.ML*
begin

Introduction

The purpose of this theory is to introduce the concept of OCL judgement (written $\tau \models \phi$) over a context τ and an OCL formula ϕ and to derive a number of mechanizeable calculi for it. Here, a *context* is usually a system transition in OCL, i.e. a pair of two states $\tau = (\tau_{\text{pre}}, \tau_{\text{post}})$; however, since our theory on OCL does make perfect sense for generalizations such as "sequences of states" of Kripke structures, we prefer the more general notion of context to simple state pairs. A formula ϕ may contain paths such as “`self.a.b`” which were interpreted in τ. The concept of a judgement occurs already in the standard in definition A.33 (Semantics of Operation Specifications). We turn it to the heart of deductive calculi for OCL: It will be the basis for equational calculi (both strict and strong equalities), congruences on formulas and local and global equivalences (terminology explained below) over judgements, and tableaux calculi over judgements.

All of them are integrated into tactic procedures that allow for automated proof state transformations even automated proofs over OCL formula. Finally, these SML-programmed tactics were bound to a specific Isar syntax.

Terminolgy

An OCL formula is *globally valid* (or just *valid* if its evaluation yields for all contexts (e.g., state transitions (s, s'), Kripke-structures, ...) true; an OCL formula is *locally valid* if this is the case for a specific context.

We have three different equational calculi for OCL forming the bare bones of automated deduction in OCL:

1. an *universal congruence* (on OCL formulae) (UC) is a (conditional) formula with an equality $E = E'$ as conclusion where E and E' has type $V_\alpha(\beta)$. These equalities hold for all contexts τ and can be processed directly by Isabelle's rewriter.

2. a *local equivalence* (LE) is a (conditional) formula with an equality $E\ \tau = E'\ \tau$ as conclusion where E and E' has type $V_\alpha(\beta)$. These equalities hold only in some context τ of type τ (?) and can only be processed by the tactic OCL_rewriter tactic in subterms occurring in judgements and in the $E's$ in $E\ \tau = E'\ \tau$.

3. a *local judgement equivalence* (LJE) which states that two judgments over OCL formulae must be equivalent in a particular context τ, so $(\tau \models E) = (\tau \models E')$. Recall that judgements require that the formula *holds*, i.e., evalutates to `true`. In contrast to LE, the formulae must not agree on all three truth values.

The Theory of OCL Judgements

In the following, the notation for validity judgements is introduced:

constdefs
OclValid :: *'a Boolean* ⇒ *bool*
OclValid P ≡ (*P* = T)
OclLocalValid :: ['*a*,'*a Boolean*] ⇒ *bool*
OclLocalValid τ *P* ≡ (*P* τ = T τ)

syntax
_*OclValid*_*std* :: '*a Boolean* ⇒ *bool*
((`Valid` (_)) *50*)
_*OclLocalValid*_*std*:: ['*a*,'*a Boolean*] ⇒ *bool*
((*1*(_)/`OclValid`(_)) *50*)

syntax
_*OclValid*_*ascii* :: '*a Boolean* ⇒ *bool*
(|= (_) *50*)
_*OclLocalValid*_*ascii*:: ['*a*,'*a Boolean*] ⇒ *bool*
((*1*(_)/|=(_)) *50*)

syntax
_*OclValid*_*math* :: '*a Boolean* ⇒ *bool*
(⊨ (_) *50*)
_*OclLocalValid*_*math*:: ['*a*,'*a Boolean*] ⇒ *bool*
((*1*(_)/ ⊨ (_)) *50*)

cp ...
lemma *cp*_*OclLocalValid*[*simp*, *intro*!] :
cp P ⟹ *cp*(λ *X* τ. τ ⊨ *P X*)
by(*simp add*: *OclLocalValid*_*def*)

The rules needed by *cp*_*unfold*.

lemma *OCL*_*cp0*_*OclLocalValid*: *cp0* (λ *x* τ. τ ⊨ *x*)
by(*simp add*: *OclLocalValid*_*def OclTrue*_*def ss*_*cp*_*defs ss*_*lifting'*)

lemma *OCL_cp0_fold_OclLocalValid*: $(\tau \models (\mathit{lift0}\ (x\ \tau))) = (\tau \models x)$
by(*simp add*: *OclLocalValid_def OclTrue_def ss_cp_defs ss_lifting'*)

ML ⟪
cpR_simpset := (!*cpR_simpset*) *addsimps* [*thm OCL_cp0_OclLocalValid*];
cpR_fold_simpset := (!*cpR_fold_simpset*) *addsimps*
[*thm OCL_cp0_fold_OclLocalValid*];
⟫

Generic Definedness Rules on Local Validity

lemma *lift1_strictify_implies_LocalValid_defined*:
$f \equiv \mathit{lift}_1\ (\mathit{strictify}\ (\lambda x.\ \llcorner g\ x \lrcorner))$
$\Longrightarrow (\tau \models \partial(f\ X)) = (\tau \models \partial\ X)$
apply(*drule lift1_strictify_is_isdef_fw*)
apply(*rule_tac* $t = \ \partial(f\ X)$ **in** *ssubst*)
apply *simp_all*
done

lemma *lift2_strictify_implies_LocalValid_defined*:
$f \equiv \mathit{lift}_2\ (\mathit{strictify}\ (\lambda x.\ \mathit{strictify}\ (\lambda y.\ \llcorner g\ x\ y \lrcorner)))$
$\Longrightarrow (\tau \models \partial(f\ X\ Y)) = ((\tau \models \partial\ X) \land\ (\tau \models \partial\ Y))$
apply(*drule lift2_strictify_is_isdef_fw*)
apply(*rule_tac* $t = \ \partial(f\ X\ Y)$ **in** *ssubst,assumption*)
apply(*simp add*: *OclAnd_def OclTrue_def o_def*
OclIsDefined_def lift0_def lift1_def lift2_def
OclLocalValid_def)
done

Semantic Representations of Validity

lemma *localValid2sem*:
$(\tau \models P) = (P\ \tau = \llcorner \mathit{True} \lrcorner)$
by (*simp add*: *OclLocalValid_def OclTrue_def lift0_def*)

lemma *localValidNot2sem*:
$(\tau \models \neg\ P) = (P\ \tau = \llcorner \mathit{False} \lrcorner)$
by (*simp add*: *OclLocalValid_def OclNot_def OclTrue_def*
strictify_def lift0_def lift1_def o_def
not_down_exists_lift, *auto*)

lemma *localValidDefined2sem*:
$(\tau \models \partial\ S) = \mathit{DEF}(S\ \tau)$
by (*simp add*: *OclLocalValid_def OclIsDefined_def strictify_def*
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)

lemma *localValidUnDefined2sem*:
$(\tau \models \not\partial\ S) = (S\ \tau = \bot)$

by (*simp add: OclLocalValid_def OclIsDefined_def strictify_def*
o_def lift0_def lift1_def lift2_def OclNot_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)

lemma *localValidToNotFalse2sem*:
$(\tau \vDash P\ (x{::}'b{::}type) \vee \partial(P\ x)) = (P\ x\ \tau \neq \llcorner False \lrcorner)$
by(*simp add: OclLocalValid_def OclIsDefined_def strictify_def*
OclNot_def OclOr_def OclAnd_def
o_def lift0_def lift1_def lift2_def
OclTrue_def DEF_lift not_down_exists_lift, auto)

lemma *localValidToNotTrue2sem*:
$(\tau \vDash \neg\ P\ (x{::}'b{::}type) \vee \partial(P\ x)) = (P\ x\ \tau \neq \llcorner True \lrcorner)$
by(*simp add: OclLocalValid_def OclIsDefined_def strictify_def*
OclNot_def OclOr_def OclAnd_def
o_def lift0_def lift1_def lift2_def
OclTrue_def DEF_lift not_down_exists_lift, auto)

lemma *isUndefined_charn*:
$(X = \bot) = (\vDash \partial\ X)$
apply (*auto simp: OclValid_def*)
apply (*simp add: OclUndefined_def OclTrue_def*
lift0_def lift1_def o_def DEF_def OclNot_def
OclIsDefined_def)
apply (*rule ext*)
apply (*drule_tac x = s* **in** *fun_cong*)
apply *auto*
done

lemma *is_FALSE_charn*:
$(X = \mathsf{F}) = (\vDash \neg\ X)$
apply (*auto simp: OclValid_def*)
apply (*simp add: OclUndefined_def OclFalse_def OclTrue_def*
lift0_def lift1_def o_def OclIsDefined_def
DEF_def OclNot_def strictify_def)
apply (*rule ext*)
apply (*drule_tac x = s* **in** *fun_cong*)
apply (*case_tac X s = down*)
apply *auto*
apply (*simp add: not_down_exists_lift*)
apply *auto*
done

lemma *is_TRUE_charn*:
$(X = \mathsf{T}) = (\vDash X)$
by (*auto simp: OclValid_def*)

Local and global validity statements are in fact complete in the sense that they can inherently express all three basic values of booleans. This foundational fact gives rise to the idea to use these judgements both for tableaux and congruence-rewrite calculi.

Instead of representations in terms of semantic primitives, the local judgements can also be expressed on a more abstract level, i.e., in terms of global equivalences.

The following property reveals that strong equality is just mirrored by the HOL equality. Note that the strong equalty is implicitly use also in the construction of sets—its choice is therefore highly critical. Depending on the layout of objects, i.e., the concrete object universe, this results in different properties. For the case of the referential universe and states only containing objects with reference to themselves in the state, logical equality coincides with referential equalty.

lemma *isUndefined_charn_local*:
$(X\ \tau = \bot\ \tau) = (\tau \models \emptyset\ X)$
by (*simp add*: *OclLocalValid_def*
OclUndefined_def OclTrue_def lift0_def lift1_def
o_def OclIsDefined_def DEF_def OclNot_def)

lemma *is_FALSE_charn_local*:
$(X\ \tau = \mathsf{F}\ \tau) = (\tau \models \neg\ X)$
apply (*simp add*: *OclLocalValid_def*
OclUndefined_def OclTrue_def OclFalse_def
lift0_def lift1_def o_def OclIsDefined_def DEF_def
strictify_def OclNot_def)
apply *auto*
apply (*case_tac X St = down*)
apply (*simp_all add*: *not_down_exists_lift*)
apply *auto*
done

lemma *is_TRUE_charn_local*:
$(X\ \tau = \mathsf{T}\ \tau) = (\tau \models X)$
by (*simp add*: *OclLocalValid_def*)

lemma *strongEq_charn*: $(\tau \models x \triangleq y) = (x\ \tau = y\ \tau)$
apply (*simp add*: *OclStrongEq_def OclLocalValid_def lift2_def*
OclTrue_def lift0_def)
done

The Standardization Rules of (Global) Validity Statements.
lemma *OclTrue_is_Valid*[*simp*]: $\models \mathsf{T}$
by (*simp add*: *OclValid_def*)

lemma *OclFalse_is_Invalid*[*simp*]: $\neg(\models \mathsf{F})$
apply (*simp add*: *OclValid_def OclTrue_def OclFalse_def lift0_def*)
by (*unfold not_def*, *rule impI*, *drule fun_cong*, *simp*)

lemma *OclUndefined_is_Invalid*[*simp*]: $\neg\ (\models \bot)$

```
  apply (simp add: OclValid_def OclTrue_def OclUndefined_def lift0_def)
  by    (unfold not_def, rule impI, drule fun_cong, simp)
```

The Standardization Rules of Local Validity Statements.

```
lemma OclTrue_is_LocalValid[simp]: τ ⊨ T
  by (simp add: OclLocalValid_def)
lemma OclFalse_is_LocalInvalid[simp]: ¬(τ ⊨ F)
  by (simp add: OclLocalValid_def OclTrue_def OclFalse_def lift0_def)
lemma OclUndefined_is_LocalInvalid[simp]: ¬ (τ ⊨ ⊥)
  by (simp add: OclLocalValid_def OclTrue_def OclUndefined_def lift0_def)
```

On the basis of the standarization rules, the simplifier can already rule out a number of absurd validity statements:

```
lemma isdef_congr[simp]:   (τ ⊨ ∂(¬ A)) = (τ ⊨ ∂(A)) by simp

lemma (τ ⊨ ∅(¬ A)) = (τ ⊨ ∅(A)) by simp

lemma τ ⊨ ∂(∂ A)  by simp

lemma τ ⊨ ∂(T)    by simp

lemma τ ⊨ ∂(F)    by simp

lemma ¬(τ ⊨ ∂(⊥)) by simp

lemma ¬(τ ⊨ ∅(T)) by simp

lemma ¬(τ ⊨ ∅(F)) by simp

lemma τ ⊨ ∅(⊥)    by simp

lemma τ ⊨ ∂(T)    by simp
```

Another Foundational Fact: *non_quatrium_datur*

```
lemma non_quatrium_datur: (τ ⊨ A) ∨ (τ ⊨ ¬ A) ∨ (τ ⊨ ∅ A)
  apply (simp add: isUndefined_charn_local [symmetric])
  apply (simp add: is_FALSE_charn_local    [symmetric])
  apply (cut_tac base_distinct)
  apply (auto simp: OclLocalValid_def)
  done


ML_setup ⟪
fun prover4 (goal,name) =
    let val thm = prove_goalw (the_context())
                             [OclFalse_def,OclTrue_def,OclUndefined_def,lift0_def]
                 goal (fn _ => [auto_tac(claset(),simpset()),
                               TRY(dtac fun_cong 1),
```

```
                          auto_tac(claset(),simpset())])
    in  bind_thm(name,thm);
        Addsimps[thm]
    end;
⟫

ML_setup ⟪
map prover4
[(F ≠ ⊥, not__FALSE__undef),
 (⊥ ≠ F, not__undef__FALSE),
 (T ≠ ⊥, not__TRUE__undef),
 (⊥ ≠ T, not__undef__TRUE),
 (F ≠ T, not__FALSE__TRUE),
 (T ≠ F, not__TRUE__FALSE),

 (F τ ≠ ⊥ τ,not__FALSE__undef_st),
 (⊥ τ ≠ F τ,not__undef__FALSE_st),
 (T τ ≠ ⊥ τ,not__TRUE__undef_st),
 (⊥ τ ≠ T τ,not__undef__TRUE_st),
 (F τ ≠ T τ,not__FALSE__TRUE_st),
 (T τ ≠ F τ,not__TRUE__FALSE_st)];

⟫

lemma absurd1: ⟦ τ ⊨ A; τ ⊨ ¬ A ⟧ ⟹ R
  by (auto simp add: is_FALSE_charn_local [symmetric],
          simp add: OclLocalValid_def)


lemma absurd1S: τ ⊨ A ⟹ ~(τ ⊨ ¬ A)
  apply (rule_tac Pa = False in swap,simp_all)
  by    (erule absurd1,simp)


lemma absurd2: ⟦ τ ⊨ A; τ ⊨ ∂ A ⟧ ⟹ R
  apply(simp add: isUndefined_charn_local[symmetric])
  apply(simp add: is_TRUE_charn_local[symmetric])
  done

lemma absurd2S: τ ⊨ A ⟹ ~(τ ⊨ ∂ A)
  apply (rule_tac Pa = False in swap,simp_all)
  by    (erule absurd2,simp)


lemma absurd3: ⟦ τ ⊨ ¬ A; τ ⊨ ∂ A ⟧ ⟹ R
  apply(simp add: isUndefined_charn_local[symmetric])
  apply(simp add: is_FALSE_charn_local[symmetric])
  done
```

```
lemma absurd3S: τ ⊨ ¬ A ⟹ ~(τ ⊨ ∅ A)
  apply (rule_tac Pa = False in swap,simp_all)
  by    (erule absurd3,simp)


lemma absurd4: ⟦ τ ⊨ ∅ (∂ A) ⟧ ⟹ R
  by (simp add: OclLocalValid_def)

lemma absurd4S[simp]: ~ (τ ⊨ ∅ (∂ A))
  by (simp add: OclLocalValid_def)

lemma absurd5: ⟦ τ ⊨ ⊥ ⟧ ⟹ R
  by (simp add: OclLocalValid_def)

lemma absurd5S[simp]: ~ (τ ⊨ ⊥)
  by (simp add: OclLocalValid_def)

lemma absurd6: ⟦ τ ⊨ F ⟧ ⟹ R
  by (simp add: OclLocalValid_def)

lemma absurd6S[simp]: ~ (τ ⊨ F)
  by (simp add: OclLocalValid_def)


lemmas absurdE = absurd1 absurd2 absurd3 absurd4 absurd5 absurd6
lemmas absurdS = absurd1S absurd2S absurd3S absurd4S absurd6S absurd6S


declare absurdE [elim]



ML_setup ⟪
val OCL_absurd_tac = best_tac (HOL_cs addEs
                     [(thm absurd1),(thm absurd2),(thm absurd3)])
⟫


lemma not_valid:   (~(τ ⊨ A)) = ((τ ⊨ ¬ A) ∨ (τ ⊨ ∅ A))
  apply (cut_tac τ = τ and A = A in non_quatrium_datur)
  apply auto
  done


lemma not_invalid: (~(τ ⊨ ¬ A)) = ((τ ⊨ A) ∨ (τ ⊨ ∅ A))
  by (cut_tac τ = τ and A = A in non_quatrium_datur, auto)

lemma not_isUndefined: (~(τ ⊨ ∅ A)) = ( (τ ⊨ A) ∨ (τ ⊨ ¬ A))
  by (cut_tac τ = τ and A = A in non_quatrium_datur, auto)
```

```
lemma isUndefined_eq_not_isDefined[simp]:
  (~(τ ⊨ ∂ A)) = (τ ⊨ ∅ A)
  apply auto
  apply (drule not_valid [THEN iffD1], auto simp: OclLocalValid_def)
  done


lemma not_isUndefined_eq_isDefined[simp]:
  (~(τ ⊨ ∅ A)) = (τ ⊨ ∂ A)
  apply (subst isUndefined_eq_not_isDefined[symmetric])
  apply (rule HOL.not_not)
  done

lemma not_isUndefined2: (τ ⊨ ∂ A) = ( (τ ⊨ A) ∨ (τ ⊨ ¬ A))
  by (simp add: not_isUndefined[symmetric])


lemmas  norm_validS =
  not_valid
  not_invalid
  not_isUndefined
  isUndefined_eq_not_isDefined
  not_isUndefined_eq_isDefined
```

Global (Universal) vs. Local Validity

```
lemma global_vs_local_validity: (∀ τ. (τ ⊨ A)) =  (⊨ A)
  by (auto simp:OclLocalValid_def OclValid_def,
      rule ext,erule_tac x = x in allE, assumption)

lemma valid_intro: ⟦ ⋀τ. τ  ⊨ A ⟧ ⟹  ⊨ A
  by (simp only: OclLocalValid_def OclValid_def,rule ext, auto)

lemma valid_elim:  ⟦ ⊨ A ⟧ ⟹        τ ⊨ A
  by (simp only: OclLocalValid_def OclValid_def)
```

Universal Congruence vs. Judgement Equivalence

```
lemma equiv_I:
  assumes a1: ⋀ τ. ((τ ⊨ A) = (τ ⊨ B))
  and a2:     ⋀ τ. ((τ ⊨ ∅ A) = (τ ⊨ ∅ B))
  shows       A = B
  apply (rule ext)
  apply (cut_tac τ = x and A = A in non_quatrium_datur)
  apply (erule disjE)
  apply (erule_tac [2] disjE)
  apply (cut_tac τ1 = x in a1 [THEN iffD1])
  apply (cut_tac [4] τ1 = x in a2 [THEN iffD1])
```

```
apply (auto)
prefer 2
apply (simp_all add: isUndefined_charn_local [symmetric])
apply (cut_tac τ = x and A = B in non_quatrium_datur)
apply (erule disjE)
apply (erule_tac [2] disjE)
prefer 2
apply (simp only: is_FALSE_charn_local [symmetric])

apply (drule a1 [THEN iffD2])

apply (drule_tac [2] a2 [THEN iffD2])
apply (auto)
apply (simp_all add: OclLocalValid_def)
done
```

Absolute equivalence of OCL formula can be expressed in terms of absolute validity as follows:

```
lemma equiv_D:
  (A = B) = ((∀ τ. ((τ ⊨ A) = (τ ⊨ B))) ∧ (∀ τ. ((τ ⊨ ∂ A) = (τ ⊨ ∂ B))))
  by (auto intro!:equiv_I)
```

Local equivalence of OCL formula can be expressed in terms of local validity as follows:

```
lemma equiv_D_local:
  (A τ = B τ) = (((τ ⊨ A) = (τ ⊨ B)) ∧ (((τ ⊨ ∂ A) = (τ ⊨ ∂ B))))
  apply (auto)
  apply (simp_all add: isUndefined_charn_local[symmetric])
  apply (simp_all only: not_valid)
  apply auto
  apply (simp_all add: is_FALSE_charn_local[symmetric])
  apply (simp_all add: OclLocalValid_def)
  done

lemma equiv_I':
  assumes A: ⋀ τ. ((τ ⊨ A) = (τ ⊨ B))
  and B:     ⋀ τ. ((τ ⊨ ¬ A) = (τ ⊨ ¬ B))
  shows      A = B
  apply (rule ext)
  apply (cut_tac τ = x and A = A in non_quatrium_datur,safe)
  apply (cut_tac τ1 = x in A [THEN iffD1])
  apply (cut_tac [3] τ1 = x in B [THEN iffD1])
  apply auto
  apply (simp only: OclLocalValid_def)
  apply (simp only: is_FALSE_charn_local [symmetric])
  apply (cut_tac τ = x and A = B in non_quatrium_datur)
  apply (safe)
  apply (drule A[THEN iffD2])
```

```
  apply (drule_tac [2] B[THEN iffD2])
  apply auto
  apply (simp only: isUndefined_charn_local [symmetric])
  done

lemma equiv_D':
  (A = B) = ((∀ τ. ((τ ⊨ A) = (τ ⊨ B))) ∧ (∀ τ. ((τ ⊨ ¬ A) = (τ ⊨ ¬ B))))
  by(auto intro!:equiv_I')


lemma equiv_D'_local:
  (A τ = B τ) = (((τ ⊨ A) = (τ ⊨ B)) ∧ (((τ ⊨ ¬ A) = (τ ⊨ ¬ B))))
  apply (auto)
  apply (simp_all add:  is_FALSE_charn_local[symmetric])
  apply (simp_all only: not_valid not_invalid is_FALSE_charn_local)
  apply auto
  apply (simp_all only: isUndefined_charn_local[symmetric])
  apply (simp_all only: OclLocalValid_def)
  done
```

The Theory of Strong and Strict Equality

The Setup: cp, strictness, definedness.

```
lemmas cp_strongEq = OclStrongEq_def  [THEN cp_lift2_fw, standard]
lemmas cp_strictEq = OclStrictEq_def  [THEN cp_lift2_fw, standard]

declare cp_strongEq [intro!] cp_strictEq [intro!]

lemma isDefined_strongEq[iff,simp]:⊨ ∂ X ≜ Y
  by(auto simp: OclIsDefined_def OclStrongEq_def
              lift2_def lift1_def lift0_def
              OclUndefined_def o_def OclTrue_def
              DEF_def OclValid_def)

lemmas  isDefined_strictEq = OclStrictEq_def
                         [THEN lift2_strictify_is_isdef_fw,
                          standard]
lemmas  OclUndefined__strictEq = OclStrictEq_def
                         [THEN lift2_undef1a_fw,
                          standard]
lemmas  strictEq__undef = OclStrictEq_def
                         [THEN lift2_undef2a_fw,
                          standard]
declare isDefined_strictEq [simp]
        OclUndefined__strictEq [simp]
        strictEq__undef [simp]
```

Properties of Strong Equality

```
lemma StrongEq_notvalid1[simp]:
```

$[\![\tau \models \partial\ x; \tau \models \not\partial\ y]\!] \Longrightarrow (x \triangleq y)\ \tau = \mathsf{F}\ \tau$
by(*simp add: localValidDefined2sem localValidUnDefined2sem DEF_def*
OclStrongEq_def OclFalse_def lift0_def lift2_def)

lemma *StrongEq_notvalid2*[*simp*]:
$[\![\tau \models \not\partial\ x; \tau \models \partial\ y]\!] \Longrightarrow (x \triangleq y)\ \tau = \mathsf{F}\ \tau$
by(*auto simp: localValidDefined2sem localValidUnDefined2sem DEF_def*
OclStrongEq_def OclFalse_def lift0_def lift2_def)

lemma *StrongEq_valid_undef* :
$[\![\tau \models \not\partial\ x; \tau \models \not\partial\ y]\!] \Longrightarrow (x \triangleq y)\ \tau = \mathsf{T}\ \tau$

will be proven in |OCL_Logic| by: by(ocl_hypsubst,simp)

oops

lemma *strictEq_is_strongEq_LE*:
$[\![\tau \models \partial\ a; \tau \models \partial\ b]\!] \Longrightarrow (a \doteq b)\tau\ = (a \triangleq b)\tau$
by(*simp add: OclStrongEq_def OclStrictEq_def OclTrue_def o_def*
OclIsDefined_def lift0_def lift1_def lift2_def
OclLocalValid_def)

In the theory files, this section contains the SML code that implements OCL_normal_tac.

lemma *strictEq_is_strongEq*[*simp*]:
$[\![\models \partial\ a; \models \partial\ b]\!] \Longrightarrow a \doteq b = a \triangleq b$
by(*tactic OCL_normal_tac (get_local_clasimpset ctxt)*
(*thmstrictEq_is_strongEq_LE*) *1*)

lemma *isDefined_if_valid*:
$\tau \models A \Longrightarrow \tau \models \partial\ A$
by (*rule classical,*
auto simp: OclTrue_def
OclIsDefined_def lift0_def lift1_def lift2_def
o_def OclLocalValid_def)

lemma *localvalid_strictEq_implies_localvalid_strongEq*:
$[\![\tau \models a \doteq b\ ]\!] \Longrightarrow \tau \models a \triangleq b$
apply(*frule isDefined_if_valid,*
simp only: OclStrictEq_def
[*THEN lift2_strictify_implies_LocalValid_defined*])
apply(*auto simp: OclTrue_def OclStrictEq_def OclStrongEq_def*
OclIsDefined_def lift0_def lift1_def lift2_def
o_def OclLocalValid_def)
done

```
lemma strongEq_subst:
  ⟦a ≜ b = T; P a = T; cp P⟧ ⟹ P b = T
  apply(unfold OclStrongEq_def OclTrue_def lift0_def lift1_def lift2_def)
  apply(rule ext)
  apply(drule_tac x = s in fun_cong)
  apply(drule_tac x = s in fun_cong)
  apply(simp add: cp_def)
  apply(erule exE)
  apply(rotate_tac -1)
  apply simp
  done


lemma strongEq_refl_UC[simp] : (a ≜ a) =  T
  apply(rule ext)
  by   (simp add: OclStrongEq_def OclTrue_def
                lift0_def lift2_def)

lemma strongEq_refl: ⊨ a ≜ a by simp

lemma strictEq_refl_UC[simp] :
  ⟦⊨ ∂ a⟧ ⟹ a ≐ a =  T
  by simp

lemma strongEq_sym:
  ⊨ a ≜ b ⟹ ⊨  b ≜ a
  apply (unfold OclStrongEq_def OclTrue_def lift0_def lift2_def OclValid_def)
  apply (rule ext)
  apply (drule_tac x = τ in fun_cong)
  apply simp
  done

lemma strongEq_trans:
  ⟦⊨ a ≜ b; ⊨ b ≜ c⟧ ⟹ ⊨ a ≜ c
  apply(unfold OclStrongEq_def OclTrue_def lift0_def lift2_def OclValid_def)
  apply (rule ext)
  apply (drule_tac x = τ in fun_cong)
  apply (drule_tac x = τ in fun_cong)
  apply simp
  done


lemma strongEq_refl_local:  τ ⊨ a ≜ a
  by (simp add: OclStrongEq_def OclLocalValid_def
               OclTrue_def lift0_def lift2_def)

lemma strongEq_sym_local:  τ ⊨ a ≜ b ⟹ τ ⊨ b ≜ a
```

by (*simp add: OclLocalValid_def OclStrongEq_def*
OclTrue_def lift0_def lift2_def)

lemma *strongEq_trans_local*:
$[\![\tau \vDash a \triangleq b; \tau \vDash b \triangleq c]\!] \Longrightarrow \tau \vDash a \triangleq c$
by (*simp add: OclLocalValid_def OclStrongEq_def*
OclTrue_def lift0_def lift2_def)

Extensionality must swap parameters and therefore looks quite inelegant ...

lemma *strongEq_ext_local*:
$(\bigwedge x.\ \tau \vDash f\,x \triangleq g\,x) \Longrightarrow \tau \vDash (\lambda\ X\ \tau.\ f\ \tau\ X) \triangleq (\lambda\ X\ \tau.\ g\ \tau\ X)$
by (*simp add: OclLocalValid_def OclStrongEq_def*
OclTrue_def lift0_def lift2_def)

thm *strongEq_ext_local*

constdefs
$SW :: \ ('b \Rightarrow 'a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$
$SW\ f \equiv (\lambda\ X\ \tau.\ f\ \tau\ X)$

lemma *SW_idem[simp]*: $SW(SW\ f) = f$
by(*simp add: SW_def*)

lemma *strongEq_ext_local2*:
$(\bigwedge x.\ \tau \vDash f\,x \triangleq g\,x) \Longrightarrow \tau \vDash (SW\ f) \triangleq (SW\ g)$
by (*simp add: OclLocalValid_def OclStrongEq_def*
OclTrue_def lift0_def lift2_def SW_def)

lemma *strongEq_subst_local*:
$[\![\ \tau \vDash a \triangleq b; \tau \vDash P\ a; cp\ P]\!] \Longrightarrow \tau \vDash P\ b$
apply (*simp add: OclStrongEq_def OclLocalValid_def cp_def*
OclTrue_def lift0_def lift1_def lift2_def)
apply (*erule exE*)
apply (*rotate_tac −1*)
apply *simp*
done

lemmas *strongEq_subst_local_sym = strongEq_sym_local*
[THEN strongEq_subst_local]

Technical Substitution Lemmas (for tactics)

lemma *strongEq1*:
$[\![\tau \vDash X \triangleq Y; \tau\ \vDash P\ Y; cp\ P\,]\!] \Longrightarrow \tau\ \vDash P\ X$
by(*drule strongEq_sym_local*,

erule strongEq_subst_local, simp)

lemmas *strongEq1_rev* = *strongEq1*[*OF strongEq_sym_local*]

lemma *strongEq2*:
$[\![\tau \models X \triangleq Y;\ (\tau \models P\ Y) = (\tau \models P'\ Y);\ cp\ P;\ cp\ P']\!]$
$\Longrightarrow (\tau \models P\ X) = (\tau \models P'\ X)$
apply(*rule iffI*)
apply(*rule_tac X=X* **in** *strongEq1*,
simp_all,drule sym, simp)
apply(*erule strongEq1_rev, simp_all*)
apply(*rule_tac X=X* **in** *strongEq1, simp_all*)
apply(*simp, erule strongEq1_rev, simp_all*)
done

lemmas *strongEq2_rev* = *strongEq2*[*OF strongEq_sym_local*]

lemma *strongEq3*:
$[\![\tau \models X \triangleq Y;\ P\ Y\ \tau = P'\ Y\ \tau;\ cp\ P;\ cp\ P'\ ]\!] \Longrightarrow P\ X\ \tau = P'\ X\ \tau$
apply(*rule_tac P=P* **and** *P' = P'* **in** *ocl_cp_subst3, simp_all*)
apply(*simp add: OclLocalValid_def OclStrongEq_def strictify_def*
o_def lift0_def lift1_def lift2_def OclNot_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemmas *strongEq3_rev* = *strongEq3*[*OF strongEq_sym_local*]

lemma *strongEq1_fw*:
$[\![\tau \models P\ X;\ \tau \models Y \triangleq X;\ cp\ P]\!] \Longrightarrow \tau \models P\ Y$
by(*erule strongEq1,simp*)

lemma *strongEq2_fw*:
$[\![(\tau \models P\ X) = (\tau \models P'\ X);\ \tau \models Y \triangleq X;\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ X) = (\tau \models P'\ X)$
by(*erule strongEq2,simp_all*)

lemma *strongEq3_fw*:
$[\![P\ X\ \tau = P'\ X\ \tau;\ \tau \models Y \triangleq X;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ Y\ \tau = P'\ Y\ \tau$
by(*erule strongEq3,simp_all*)

lemma *strongEq1_fw2*:
$[\![\tau \models P\ X;\ \tau \models X \triangleq Y;\ cp\ P]\!] \Longrightarrow \tau \models P\ Y$
by(*erule strongEq1_rev,simp*)

lemma *strongEq2_fw2*:
$[\![(\tau \models P\ X) = (\tau \models P'\ X);\ \tau \models X \triangleq Y;\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ X) = (\tau \models P'\ X)$
by(*erule strongEq2_rev,simp_all*)

lemma *strongEq3_fw2*:
$[\![P\ X\ \tau = P'\ X\ \tau;\ \tau \models X \triangleq Y;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ Y\ \tau = P'\ Y\ \tau$
by(*erule strongEq3_rev,simp_all*)

Automated Canonization and Rewriting in OCL

OCL offers a number of equivalences: From strict and strong equality, we have three types of congruences/equivalences, such that reasoning about them is tedious and must be tactically supported. The ultimate goal of this section is to work out two tactic procedures for this purpose that pursue this goal. As basis for these tactic procedures, a number of rules and rule variants must be derived, whose existence is merely justified by deduction technical reasons.

validity propagation rules
lemmas *strictEq1 = strongEq1[OF localvalid_strictEq_implies_localvalid_strongEq]*
lemmas *strictEq2 = strongEq2[OF localvalid_strictEq_implies_localvalid_strongEq]*
lemmas *strictEq3 = strongEq3[OF localvalid_strictEq_implies_localvalid_strongEq]*

lemmas *strictEq1_rev = strongEq1_rev*
[OF localvalid_strictEq_implies_localvalid_strongEq]
lemmas *strictEq2_rev = strongEq2_rev*
[OF localvalid_strictEq_implies_localvalid_strongEq]
lemmas *strictEq3_rev = strongEq3_rev*
[OF localvalid_strictEq_implies_localvalid_strongEq]

lemma *subst_LJ_TRUE*:
$[\![\tau \models X;\ \tau \models P\ \mathsf{T};\ cp\ P\ ]\!] \Longrightarrow \tau\ \models P\ X$
apply (*simp add: OclLocalValid_def,drule sym*)
apply (*erule_tac P = P* **in** *ocl_cp_subst,auto*)
done

lemma *subst_LJ_TRUE_fw*:
$[\![\tau \models X;\ \tau \models P\ X;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ \mathsf{T}$
apply (*simp add: OclLocalValid_def*)
apply (*erule_tac P = P* **in** *ocl_cp_subst*)
apply *auto*
done

lemma *subst_LJ_TRUE_fw_rev*:
$[\![\tau \models P\ X;\ \tau \models X;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ \mathsf{T}$
apply (*simp add: OclLocalValid_def*)
apply (*erule_tac P = P* **in** *ocl_cp_subst*) **back back**

apply *auto*
done

lemma *subst_LE_TRUE*:
$[\![\tau \models X;\ P\ \top\ \tau = P'\ \top\ \tau;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ X\ \tau = P'\ X\ \tau$
apply(*rule local_validity_propagation2* [*simplified is_TRUE_charn_local*]) **back**
by *auto*

lemma *subst_LE_TRUE_fw*:
$[\![\tau \models X;\ P\ X\ \tau = P'\ X\ \tau\ ;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ \top\ \tau = P'\ \top\ \tau$
apply(*simp add*: *OclLocalValid_def*)
apply(*rule trans*, *rule sym*)
apply(*erule_tac* τ=τ **and** P=P **in** *cp_charn*, *simp*)
apply(*rule trans*, *assumption*)
apply(*erule_tac* τ=τ **and** P=P' **in** *cp_charn*, *simp*)
done

lemma *subst_LE_TRUE_fw_rev*:
$[\![P\ (X::'a \Rightarrow Boolean\_0)\ \tau = P'\ X\ \tau\ ;\ \tau \models X;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ \top\ \tau = P'\ \top\ \tau$
by(*erule subst_LE_TRUE_fw*, *auto*)

lemma *subst_LJE_TRUE*:
$[\![\tau \models X;\ (\tau \models P\ \top) = (\tau \models P'\ \top);\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ X) = (\tau \models P'\ X)$
by(*erule subst_LE_TRUE*, *auto*)

lemma *subst_LJE_TRUE_fw*:
$[\![\tau \models X;\ (\tau \models P\ X) = (\tau \models P'\ X);\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ \top) = (\tau \models P'\ \top)$
by(*erule subst_LE_TRUE_fw*, *auto*)

lemma *subst_LJE_TRUE_fw_rev*:
$[\![(\tau \models P\ X) = (\tau \models P'\ X);\ \tau \models X;\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ \top) = (\tau \models P'\ \top)$
by(*erule subst_LJE_TRUE_fw*, *auto*)

lemma *subst_TRUE_LJE*:
$[\![\tau \models X;\ cp\ P\ ]\!] \Longrightarrow (\tau \models P\ X) = (\tau \models P\ \top)$
apply(*auto*)
apply(*erule subst_LJ_TRUE_fw*, *auto*)
apply(*erule subst_LJ_TRUE*,*auto*)
done

lemma *subst_LJ_FALSE*:
$[\![\tau \models \neg\ X;\ \tau \models P\ \mathsf{F};\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ X$
apply (*drule is_FALSE_charn_local* [*THEN iffD2*])
apply (*rotate_tac* −*1*)
apply (*drule sym*)
apply (*simp add*: *OclLocalValid_def*)
apply (*erule_tac* $P = P$ **in** *ocl_cp_subst*)
apply *auto*
done

lemma *subst_LJ_FALSE_fw*:
$[\![\ \tau \models \neg\ X;\ \tau \models P\ X;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ \mathsf{F}$
apply (*drule is_FALSE_charn_local* [*THEN iffD2*])
apply (*rotate_tac* −*1*)
apply (*simp add*: *OclLocalValid_def*)
apply (*erule_tac* $P = P$ **in** *ocl_cp_subst*)
apply *auto*
done

lemma *subst_LJ_FALSE_fw_rev*:
$[\![\ \tau \models P\ X;\ \tau \models \neg\ X;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ \mathsf{F}$
apply (*erule subst_LJ_FALSE_fw*)
apply *auto*
done

lemma *subst_LE_FALSE*:
$[\![\tau \models \neg\ X;\ P\ \mathsf{F}\ \tau = P'\ \mathsf{F}\ \tau;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ X\ \tau = P'\ X\ \tau$
apply(*rule local_validity_propagation3* [*simplified is_FALSE_charn_local*]) **back**
by *auto*

lemma *subst_LE_FALSE_fw*:
$[\![\tau \models \neg\ X;\ P\ X\ \tau = P'\ X\ \tau\ ;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ \mathsf{F}\ \tau = P'\ \mathsf{F}\ \tau$
apply(*simp only*: *is_FALSE_charn_local*[*symmetric*])
apply(*rule trans*, *rule sym*)
apply(*erule_tac* $\tau{=}\tau$ **and** $P{=}P$ **in** *cp_charn*, *simp*)
apply(*rule trans*, *assumption*)
apply(*erule_tac* $\tau{=}\tau$ **and** $P{=}P'$ **in** *cp_charn*, *simp*)
done

lemma *subst_LE_FALSE_fw_rev*:
$[\![P\ X\ (\tau{::}'a) = P'\ X\ \tau;\ \tau \models \neg\ X;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ \mathsf{F}\ \tau = P'\ \mathsf{F}\ \tau$
by(*erule subst_LE_FALSE_fw*, *auto*)

lemma *subst_LJE_FALSE*:
$[\![\tau \models \neg\ X;\ (\tau \models P\ \mathsf{F}) = (\tau \models P'\ \mathsf{F});\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ X){=}(\tau \models P'\ X)$
by(*erule subst_LE_FALSE*, *auto*)

lemma *subst_LJE_FALSE_fw*:
$[\![\tau \models \neg\ X;\ (\tau \models P\ X)=(\tau \models P'\ X);\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ \mathsf{F}) = (\tau \models P'\ \mathsf{F})$
by(*erule subst_LE_FALSE_fw, auto*)

lemma *subst_LJE_FALSE_fw_rev*:
$[\![(\tau \models P\ X)=(\tau \models P'\ X);\ \tau \models \neg\ X;\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ \mathsf{F}) = (\tau \models P'\ \mathsf{F})$
by(*erule subst_LJE_FALSE_fw, auto*)

lemma *subst_FALSE_LJE*:
$[\![\tau \models \neg\ X;\ cp\ P\ ]\!] \Longrightarrow (\tau \models P\ X)=(\tau \models P\ \mathsf{F})$
apply(*auto*)
apply(*erule subst_LJ_FALSE_fw, auto*)
apply(*erule subst_LJ_FALSE,auto*)
done

lemma *subst_LJ_undef*:
$[\![\ \tau \models \emptyset\ X;\ \tau \models P\ \bot;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ X$
apply (*drule isUndefined_charn_local [THEN iffD2]*)
apply (*rotate_tac −1*)
apply (*drule sym*)
apply (*simp add: OclLocalValid_def*)
apply (*erule_tac P = P* **in** *ocl_cp_subst*)
apply *auto*
done

lemma *subst_LJ_undef_fw*:
$[\![\ \tau \models \emptyset\ X;\ \tau \models P\ X;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ \bot$
apply (*drule isUndefined_charn_local [THEN iffD2]*)
apply (*rotate_tac −1*)
apply (*simp add: OclLocalValid_def*)
apply (*erule_tac P = P* **in** *ocl_cp_subst*)
apply *auto*
done

lemma *subst_LJ_undef_fw_rev*:
$[\![\ \tau \models P\ X;\ \tau \models \emptyset\ X;\ cp\ P\ ]\!] \Longrightarrow \tau \models P\ \bot$
by (*erule subst_LJ_undef_fw, auto*)

lemma *subst_LE_undef*:
$[\![\tau \models \emptyset\ X;\ P\ \bot\ \tau = P'\ \bot\ \tau;\ cp\ P;\ cp\ P']\!] \Longrightarrow P\ X\ \tau = P'\ X\ \tau$
apply (*rule_tac P=P* **and** *τ=τ* **in** *ocl_cp_subst3*)

apply (*simp_all*)
apply (*simp add: OclLocalValid_def OclIsDefined_def strictify_def OclNot_def o_def lift0_def lift1_def lift2_def OclTrue_def DEF_def OclUndefined_def*)
done

lemma *subst_LE_undef_fw*:
$[\![\tau \models \emptyset\ X; P\ X\ \tau = P'\ X\ \tau\ ;\ cp\ P;\ cp\ P']\!] \Longrightarrow\ P\ \bot\ \tau = P'\ \bot\ \tau$
apply (*rule_tac P=P* **and** τ=τ **in** *ocl_cp_subst3*)
apply (*simp_all*)
apply (*simp add: OclLocalValid_def OclIsDefined_def strictify_def OclNot_def o_def lift0_def lift1_def lift2_def OclTrue_def DEF_def OclUndefined_def*)
done

lemma *subst_LE_undef_fw_rev*:
$[\![P\ (X{::}'a{\Rightarrow}'b{::}bot)\ \tau = P'\ X\ \tau;\ \tau \models \emptyset\ X;\ cp\ P;\ cp\ P']\!] \Longrightarrow\ P\ \bot\ \tau = P'\ \bot\ \tau$
by(*erule subst_LE_undef_fw, auto*)

lemma *subst_LJE_undef*:
$[\![\tau \models \emptyset\ X;\ (\tau \models P\ \bot) = (\tau \models P'\ \bot);\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ X){=}(\tau \models P'\ X)$
by(*erule subst_LE_undef, auto*)

lemma *subst_LJE_undef_fw*:
$[\![\tau \models \emptyset\ X;\ (\tau \models P\ X){=}(\tau \models P'\ X);\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ \bot) = (\tau \models P'\ \bot)$
by(*erule subst_LE_undef_fw, auto*)

lemma *subst_LJE_undef_fw_rev*:
$[\![(\tau \models P\ X){=}(\tau \models P'\ X);\ \tau \models \emptyset\ X;\ cp\ P;\ cp\ P']\!] \Longrightarrow (\tau \models P\ \bot) = (\tau \models P'\ \bot)$
by(*erule subst_LJE_undef_fw, auto*)

lemma *subst_undef_LJE*:
$[\![\tau \models \emptyset\ X;\ cp\ P\ ]\!] \Longrightarrow (\tau \models P\ X){=}(\tau \models P\ \bot)$
apply(*auto*)
apply(*erule subst_LJ_undef_fw, auto*)
apply(*erule subst_LJ_undef,auto*)
done

lemmas *strictEq1_fw2 = strongEq1_fw2 [OF _ localvalid_strictEq_implies_localvalid_strongEq]*
lemmas *strictEq2_fw2 = strongEq2_fw2 [OF _ localvalid_strictEq_implies_localvalid_strongEq]*
lemmas *strictEq3_fw2 = strongEq3_fw2 [OF _ localvalid_strictEq_implies_localvalid_strongEq]*

lemmas *strictEq1_fw = strongEq1_fw [OF _ localvalid_strictEq_implies_localvalid_strongEq]*

lemmas *strictEq2_fw* = *strongEq2_fw*
[*OF* _ *localvalid_strictEq_implies_localvalid_strongEq*]
lemmas *strictEq3_fw* = *strongEq3_fw*
[*OF* _ *localvalid_strictEq_implies_localvalid_strongEq*]

Implementation of OCL_hyp_subst_tac

These rules give some ideas for automation:

1. OCL_norm_tac computes local norm of validities: $\models A \Longrightarrow \models B$ is converted $\tau \models A \Longrightarrow \tau \models B$ .... This is done forward and backward style.

2. OCL_absurd_tac should decide if a subgoal is OCL absurd.

3. OCL_hyp_subst_tac for variables or parameters A, which uses:

$$A = t, t = A \qquad \text{(standard hyp\_subst\_tac)} \tag{B.6}$$
$$A\tau = t\tau, t\tau = A\tau \qquad \text{(local congruences)} \tag{B.7}$$
$$\models \emptyset A, \tau \models \emptyset A \qquad \text{(undefinednesses local and global)} \tag{B.8}$$
$$\models \neg A, \tau \models \neg A \qquad \text{(falsities local and global)} \tag{B.9}$$
$$\models A \triangleq t, \models t \triangleq A \qquad \text{(strong equalities local and global)} \tag{B.10}$$
$$\tau \models A \triangleq t, \tau \models A \triangleq t \tag{B.11}$$

4. OCL_def_hyp_split_tac:

   - atomizes definedness hypothesises , e.g., $\partial(f\ A\ B) = \partial A \wedge \partial B \ldots$
   - case-splits undefinedness hypothesis $\emptyset(f\ A\ B) = \emptyset(A) \vee \emptyset(B) \ldots$
   - and performs safe_tac and OCL_hyp_subst_tac. (but no global simplification afterwards in order to do not too much in a black box.

5. *OCL_simp* and *OCL_asm_simp*. New rewriter that can cope with _ $\triangleq$ _ and _ $\doteq$ _ and its specific format of the subst rule. Future: use standard rewriter with a fake subst rule on OCL, transform the proof objects and refeed this into the kernel?

6. integrate into simp-procedure as looper or subgoaler.

lemmas *DEF_rules* = *subst_LE_TRUE* [*of* _ ∂ *X*,*standard*]
subst_LJE_TRUE[*of* _ ∂ *X*,*standard*]
subst_LJ_TRUE [*of* _ ∂ *X*,*standard*]

lemmas *DEF_fw_rev_rules* =
subst_LJ_TRUE_fw_rev [*of* _ _ ∂ *X*,*standard*]
subst_LJE_TRUE_fw_rev[*of* _ _ ∂ *X*,*standard*]
subst_LE_TRUE_fw_rev [*of* _ ∂ *X*,*standard*]

Testing OCL_hyp_subst_tac

lemma $\llbracket \tau \models A; \tau \models \neg B; \tau \models A \wedge B ; \tau \models \not\partial\ B\rrbracket \Longrightarrow \tau \models A \wedge B \wedge C$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*, *simp*)

lemma $\llbracket \tau \models A; \tau \models A \wedge B ; \tau \models \not\partial\ B; (\tau \models A) = (\tau \models C)\rrbracket \Longrightarrow A\ \tau = B\ \tau$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*, *simp*)

lemma $!!\tau\ A\ .\ \llbracket \tau \models A; \tau \models A \wedge B ; \tau \models \not\partial\ B; (\tau \models A) = (\tau \models C)\rrbracket \Longrightarrow A\ \tau = B\ \tau$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*, *simp*)

lemma $\llbracket\ \ \tau \models A; \tau \models D \triangleq E\ ;\ \ \tau \models D \triangleq 1\rrbracket \Longrightarrow \tau \models A \wedge E \triangleq 1$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*, *simp*)

lemma $\llbracket\ \ \tau \models A; \tau \models D \triangleq E\ ;\ \ \tau \models 1 \triangleq D\rrbracket\ \ \Longrightarrow \tau \models A \wedge E \triangleq 1$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*, *simp*)

lemma $\llbracket\ \ \tau \models A; \tau \models E \triangleq D\ ; \tau \models 1 \triangleq E\ ;\ \ \tau \models 1 \triangleq D\rrbracket\ \ \Longrightarrow \tau \models A \wedge E \triangleq 1$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*, *simp*)

lemma $\llbracket \tau \models A; \tau \models \neg B; \tau \models 1 \triangleq E;\ \ A\ \tau = B\ \tau;\ P\ \tau = \mathsf{T}\ \tau;$
$\quad\quad \tau \models D \doteq 1; \tau \models D \doteq E\rrbracket$
$\quad \Longrightarrow \tau \models A \wedge B$
apply(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*)
apply(*simp*)
done

lemma $\llbracket \tau \models A; \tau \models \neg B; \tau \models E \triangleq 1; \tau \models 1 \triangleq E;$
$\quad\quad \tau \models E \doteq 1;\tau \models 1 \doteq E;\tau \models \not\partial\ B\rrbracket \Longrightarrow \tau \models A \wedge B$
by(*tacticOCL_hyp_subst_tac* (*get_local_clasimpset ctxt*) *1*,*simp*)

Testing OCL_subst_tac

lemma *isDefined_cong5* : $(\partial\ X \vee \not\partial\ X) = \mathsf{T}$
apply (*rule ext*, *case_tac* $x \models\ \ \partial(X)$)

apply (*tacticOCL_subst_tac*(*false*, *false*, *false*)
$\quad\quad\quad\quad\quad\quad\quad$ [] (*get_local_clasimpset ctxt*) *1*, *simp_all*)
apply (*tacticOCL_subst_tac*(*false*, *false*, *false*)
$\quad\quad\quad\quad\quad\quad\quad$ [] (*get_local_clasimpset ctxt*) *1*, *simp_all*)
done

Some more Absurdities

lemma *not_isDefined_if_isUndefined*:
$\ \ \tau \models \partial\ A \Longrightarrow (\sim(\tau \models \not\partial\ A))$
  by *auto*

lemma *non_tertium_datur_if_isDefined*:
$[\![\ \tau \models \partial A;\ \tau \models A \Longrightarrow P;\ \tau \models\ \neg\ A \Longrightarrow P\ ]\!] \Longrightarrow\ \ P$
by (*cut_tac* $\tau = \tau$ **and** $A = A$ **in** *non_quatrium_datur*, *auto*)

lemma *isDefined_if_valid'*:
$\tau \models A \Longrightarrow \tau \models \partial\ A$
by (*rule classical*, *auto*)

lemma *isDefined_if_invalid*:
$\tau \models \neg\ A \Longrightarrow \tau \models \partial\ A$
by (*rule classical*, *auto*)

Trichotomy Specialized for if

lemma *if_trichotomy*:
$[\![\ \tau \models P\ \bot;\ \tau \models P\ Y;\ \tau \models P\ Z;\ cp\ P\ ]\!] \Longrightarrow$
$\tau \models P\ (\mathit{if}\ X\ \mathit{then}\ Y\ \mathit{else}\ Z\ \mathit{endif})$
apply (*simp add*: *OclLocalValid_def*)
apply (*rule_tac* $P = \lambda x.\ P\ (\mathit{if}\ x\ \mathit{then}\ Y\ \mathit{else}\ Z\ \mathit{endif})$ **in** *cp_distinct_core*)
apply *auto*
apply (*rule_tac* $P = P$ **in** *cp_compose2*)
apply *auto*
done

lemma *if_trichotomyE*:
$[\![\ \tau \models P\ (\mathit{if}\ X\ \mathit{then}\ Y\ \mathit{else}\ Z\ \mathit{endif});$
$\quad \tau \models P\ \bot\ \ \Longrightarrow R;$
$\quad \tau \models P\ Y\ \ \Longrightarrow R;$
$\quad \tau \models P\ Z\ \ \Longrightarrow R;$
$\quad cp\ P\ ]\!]$
$\Longrightarrow R$
apply (*cut_tac* $\tau = \tau$ **and** $A = X$ **in** *non_quatrium_datur*)
apply *safe*

apply (*drule_tac* [*3*] *subst_LJ_undef_fw*)
prefer *3*
apply (*assumption*)
apply (*rule_tac* [*3*] $P = P$ **in** *cp_compose2*, *simp_all*)

apply (*drule_tac* [*2*] *subst_LJ_FALSE_fw*)
prefer *2*
apply (*assumption*)
apply (*drule subst_LJ_TRUE_fw*)
back
apply (*assumption*)
apply *auto*
apply (*rule_tac* [*2*] $P = P$ **in** *cp_compose2*)

```
apply (rule_tac P = P in cp_compose2)
apply auto
done
```

```
lemmas OclIf_LJE[simp] =
    subst_TRUE_LJE [of _ _ λX. if X
                                   then Y
                                   else Z endif,
                 simplified]
    subst_FALSE_LJE[of _ _ λX. if X
                                   then Y
                                   else Z endif,
                 simplified]
    subst_undef_LJE[of _ _ λX. if X
                                   then Y
                                   else Z endif,
                 simplified]
```

A Weak (Propositional) LJE Calculus for Propositional OCL

The weak propositional LJE gives rise to a fairly promising decision procedure for a fragment of OCL:

- blow away quantifiers
- infer from inclusions of variables the invariants
- infer from the invariants the definedness of variables
- all remaining variables occuring in formula: case-splitting over definedness, congruence closure over undefinednesses,
- simp_all: universal congruences
- simp_all: weak propositional LJE
- blast_tac.

Weak LJE Calculus means here that everything is restricted to defined expressions.

```
lemma localValidNot2not :
  ⟦ τ ⊨ ∂ A ⟧ ⟹ (τ ⊨ ¬ A) = (¬ (τ ⊨ A))
  by(auto elim!: non_tertium_datur_if_isDefined)
```

```
lemma localValidAnd2conj :
  ⟦ τ ⊨ ∂ A; τ ⊨ ∂ B ⟧ ⟹ (τ ⊨ A ∧ B) = ((τ ⊨ A) ∧ (τ ⊨ B))
  apply(auto elim!: non_tertium_datur_if_isDefined)
  by(tacticALLGOALS(OCL_hyp_subst_tac (get_local_clasimpset ctxt)),simp_all)
```

lemma *localValidOr2disj* :
$[\![\ \tau \models \partial\ A;\ \tau \models\ \partial\ B\ ]\!] \Longrightarrow (\tau \models A \vee B) = ((\tau \models A) \vee (\tau \models B))$
apply(*auto elim*!: *non_tertium_datur_if_isDefined*)
by(*tacticALLGOALS*(*OCL_hyp_subst_tac* (*get_local_clasimpset ctxt*)),*simp_all*)

lemma *localValidImplies2impl* :
$[\![\ \tau \models \partial\ A;\ \tau \models\ \partial\ B\ ]\!] \Longrightarrow (\tau \models A \longrightarrow B) = ((\tau \models A) \longrightarrow (\tau \models B))$
by(*simp add*: *OclImplies_def localValidNot2not*
localValidOr2disj localValidAnd2conj)

lemma *isDefined_notI*: $[\![\tau \models \partial\ A]\!] \Longrightarrow \tau \models \partial\ (\neg\ A)$ **by** *simp*

lemma *isDefined_andI*: $[\![\tau \models \partial\ A;\ \tau \models \partial\ B]\!] \Longrightarrow \tau \models \partial\ (A \wedge B)$
apply(*auto elim*!: *non_tertium_datur_if_isDefined*)
apply(*tacticALLGOALS*(*OCL_hyp_subst_tac* (*get_local_clasimpset ctxt*)))
apply(*simp_all*)
done

lemma *isDefined_orI*:$[\![\tau \models \partial\ A;\ \tau \models \partial\ B]\!] \Longrightarrow \tau \models \partial\ (A \vee B)$
apply(*auto elim*!: *non_tertium_datur_if_isDefined*)
apply(*tacticALLGOALS*(*OCL_hyp_subst_tac* (*get_local_clasimpset ctxt*)))
apply(*simp_all*)
done

lemma *isDefined_xorI*:$[\![\tau \models \partial\ A;\ \tau \models \partial\ B]\!] \Longrightarrow \tau \models \partial\ (A \oplus B)$
apply(*auto elim*!: *non_tertium_datur_if_isDefined*)
apply(*tacticALLGOALS*(*OCL_hyp_subst_tac* (*get_local_clasimpset ctxt*)))
apply(*simp_all*)
done

lemma *isDefined_impliesI*:$[\![\tau \models \partial\ A;\ \tau \models \partial\ B]\!] \Longrightarrow \tau \models \partial\ (A \longrightarrow B)$
apply(*auto elim*!: *non_tertium_datur_if_isDefined*)
apply(*tacticALLGOALS*(*OCL_hyp_subst_tac* (*get_local_clasimpset ctxt*)))
apply(*simp_all*)
done

lemma *isDefined_implies2I*:$[\![\tau \models \partial\ A;\ \tau \models \partial\ B]\!] \Longrightarrow \tau \models \partial\ (A \xrightarrow{2} B)$
apply(*auto elim*!: *non_tertium_datur_if_isDefined*)
apply(*tacticALLGOALS*(*OCL_hyp_subst_tac* (*get_local_clasimpset ctxt*)))
apply(*simp_all*)
done

lemma *localValidImplies22impl* :
$[\![\ \tau \models \partial\ A;\ \tau \models\ \partial\ B\ ]\!] \Longrightarrow (\tau \models A\ \xrightarrow{2} B) = ((\tau \models A) \longrightarrow (\tau \models B))$
by(*simp add*: *OclImplies2_def localValidNot2not isDefined_andI*
localValidOr2disj localValidAnd2conj)

Isar Interface Setup

This part of the so-called thymorpher generates a number of rules and a specific setup for all operators given to him.

This includes (all examples are for the case of the weak (strict) equality here.

1. context passingness rules (cp) for all unary and binary operators, e.g.:

$$[\![cpP; cpP']\!] \Longrightarrow cp(\lambda X.PX \doteq P'X)$$

   These rules were used both as simplifier rules as well as safe introduction rules.

2. all undefinedness reduction rules (unary and binary case), e.g.:

$$\bot \doteq X = \bot\text{"}and\text{"}X \doteq \bot = \bot$$

   These rules were used as global simplifier rules.

3. universal congruences for definedness, e.g.:

$$\partial X \doteq Y = \partial X \wedge \partial Y$$

4. the local judgement equivalence for definedness, e.g.:

$$\tau \vDash \partial(X \doteq Y) = (\tau \vDash \partial X \wedge \tau \vDash \partial Y)$$

   This equality is used for optimized reasoning over definedness inside some OCL-tactics.

Definition of ocl_subst

In the theory files, this section contains the Isar setup code for ocl_subst.

Definition of ocl_auto

In the theory files, this section contains the Isar setup code for ocl_auto.

end

B.4.4. OCL Logic

theory *OCL_Logic*
  imports $*HOLOCL_HOME/src/OCL_Logic_core*
begin

Testing and Demonstrations of Main Tactics

Automatic Generation of UC Variants of Global Equivalence Theorems

ML⟪ *bind_thm(strictEq_is_strongEq_UC,*
OCL_normal (the_context())(thmstrictEq_is_strongEq_LE)) ⟫

thm *OCL_cp_OclStrictEq*
OCL_undef_1_OclStrictEq
OCL_undef_2_OclStrictEq
OCL_is_def_OclStrictEq
OCL_is_defopt_OclStrictEq

thm *OclIsDefined_def*

ocl_setup_op[*OclIsDefined*]
ocl_setup_op[*OclStrongEq*]
ocl_setup_op[*OclAnd, OclNot*]
ocl_setup_op[*OclOr_alt, OclXor_alt*]
ocl_setup_op[*OclImplies_alt, OclImplies1_alt, OclImplies2_alt*]
ocl_setup_op [*OclSand, OclSor_alt, OclSxor_alt, OclSimplies_alt*]
ocl_setup_op [*OclIf*]

Test of ocl_hypsubst

lemma $[\![\ \tau \vDash A;\ \tau \vDash \neg B;\ \tau \vDash A \wedge B\ ;\ \tau \vDash \not\partial\ B]\!] \Longrightarrow \tau \vDash A \wedge B$
by(*ocl_hypsubst* [*1*], *simp*)

lemma $[\![\ \tau \vDash A;\ \tau \vDash A \wedge B\ ;\ \tau \vDash \not\partial\ B;\ (\tau \vDash A) = (\tau \vDash C)]\!] \Longrightarrow A\ \tau = B\ \tau$
by(*ocl_hypsubst, simp*)

lemma $!!\tau\ A.\ [\![\ \tau \vDash A;\ \tau \vDash A \wedge B\ ;\ \tau \vDash \not\partial\ B;\ (\tau \vDash A) = (\tau \vDash C)]\!] \Longrightarrow A\ \tau = B\ \tau$
by(*ocl_hypsubst, simp*)

lemma $[\![\ \ \tau \vDash A;\ \tau \vDash D \triangleq E\ ;\ \ \tau \vDash D \triangleq \mathit{1}]\!] \Longrightarrow \tau \vDash A \wedge E \triangleq \mathit{1}$
by(*ocl_hypsubst, simp*)

lemma $[\![\ \ \tau \vDash A;\ \tau \vDash D \triangleq E\ ;\ \ \tau \vDash \mathit{1} \triangleq D]\!]\ \ \Longrightarrow \tau \vDash A \wedge E \triangleq \mathit{1}$
by(*ocl_hypsubst, simp*)

lemma $[\![\ \ \tau \vDash A;\ \tau \vDash E \triangleq D\ ;\ \ \tau \vDash \mathit{1} \triangleq D]\!]\ \ \Longrightarrow \tau \vDash A \wedge E \triangleq \mathit{1}$
by(*ocl_hypsubst, simp*)

lemma $[\![\ \tau \vDash A;\ \tau \vDash \neg B;\ \tau \vDash E{\triangleq}\mathit{1};\ \tau \vDash \mathit{1}{\triangleq}E;$
$\tau \vDash E{\doteq}\mathit{1};\tau \vDash \mathit{1}{\doteq}E;\tau \vDash \not\partial\ B]\!] \Longrightarrow \tau \vDash A \wedge B$
by(*ocl_hypsubst,simp*)

Test of ocl_subst

lemma $(\partial\ X \vee \not\partial\ X) = \mathsf{T}$
apply (*rule ext,case_tac x* $\vDash\ \ \partial(X)$)
apply (*ocl_subst* (*no_asm_simp*),*simp_all*)

apply $(ocl\_subst\ \ (no\_asm\_use)(no\_asm\_simp)(no\_concl\_simp) \mid ocl\_subst)$
apply $(simp)$
done

lemma $StrongEq\_valid\_undef$:
$[\![\tau \vDash \not\partial\ x; \tau \vDash \not\partial\ y]\!] \Longrightarrow (x \triangleq y)\ \tau = \mathsf{T}\ \tau$
by$(ocl\_hypsubst,simp)$

The following lemma explains strong equality entirely in terms of OCL in the sense of the standard, i.e., by using definedness and strict equality, which is the default. Due to *strongEq_charn* $((?\tau \vDash\ ?x \triangleq\ ?y) = (?x\ ?\tau = \ ?y\ ?\tau))$, the outstanding importance of strong equality becomes obvious. The following lemma explains, why its reformulation in standard operators is possible, but tedious, and represents a post-hoc justification for the introduction strong equality.

lemma $strongEq\_charn2$:
$(x \triangleq y) = (\ if\ \partial\ x \wedge \partial\ y\ then\ x \doteq y$
$\qquad else\ if\ \not\partial\ x \wedge \not\partial\ y\ then\ \mathsf{T}\ else\ \mathsf{F}\ endif$
$\qquad endif)$
apply $(rule\ ext,case\_tac\ xa \vDash\ \partial(x),simp\_all)$
apply $(ocl\_subst,simp\_all)$
apply $(case\_tac\ xa \vDash\ \partial(y),simp\_all)$
apply $(ocl\_subst,simp\_all,\ rule\ sym)$
apply $(auto\ intro\colon strictEq\_is\_strongEq\_LE)$
prefer 2
apply $(case\_tac\ xa \vDash\ \partial(y),simp\_all)$
apply $(ocl\_hypsubst\ [2],\ simp\_all)$
apply $(rotate\_tac\ -1)$
apply $(ocl\_subst,simp\_all)$
apply $(ocl\_hypsubst\ [1],ocl\_hypsubst\ [2],\ simp\_all)$
done

Strong and Weak Definedness Calculi

lemma $isDefined\_cong5 : (\partial\ X \vee \not\partial\ X) = \mathsf{T}$
apply $(rule\ ext,case\_tac\ x \vDash\ \partial(X))$
by $\ (ocl\_subst,\ simp\_all,\ ocl\_subst,\ simp\_all)$

lemma $isDefined\_cong6 : (\not\partial\ X \vee \partial\ X\ ) = \mathsf{T}$
apply $(rule\ ext,case\_tac\ x \vDash\ \partial(X))$
by $\ (ocl\_subst,\ simp\_all,\ ocl\_subst,\ simp)$

lemma $isDefined\_cong7 : (\partial\ X \vee \not\partial\ X \vee Z) = \mathsf{T}$
apply $(rule\ ext,case\_tac\ x \vDash\ \partial(X))$
by $\ (ocl\_subst,\ simp\_all,\ ocl\_subst,\ simp)$

lemma $isDefined\_cong8 : (\not\partial\ X \vee \partial\ X \vee Z) = \mathsf{T}$
apply $(rule\ ext,case\_tac\ x \vDash\ \partial(X))$

by $(ocl\_subst, simp\_all, ocl\_subst, simp)$

lemma $isDefined\_cong9 : (\partial\ X \wedge \not\partial\ X) = \mathsf{F}$
apply $(rule\ ext, case\_tac\ x \models\ \partial(X))$
by $(ocl\_subst, simp\_all, ocl\_subst, simp)$

lemma $isDefined\_cong10 : (\not\partial\ X \wedge \partial\ X) = \mathsf{F}$
apply $(rule\ ext, case\_tac\ x \models\ \partial(X))$
by $(ocl\_subst, simp\_all, ocl\_subst, simp)$

lemma $isDefined\_cong11 : (\partial\ X \wedge \not\partial\ X \wedge Z) = \mathsf{F}$
apply $(rule\ ext, case\_tac\ x \models\ \partial(X))$
by $(ocl\_subst, simp\_all, ocl\_subst, simp)$

lemma $isDefined\_cong12 : (\not\partial\ X \wedge \partial\ X \wedge Z) = \mathsf{F}$
apply $(rule\ ext, case\_tac\ x \models\ \partial(X))$
by $(ocl\_subst, simp\_all, ocl\_subst, simp)$

lemmas *core_ definedness =*
OclIsDefined_ _False OclIsDefined_ _True OclIsDefined_ _undef
isDefined_ idem isDefined_ notD0
isDefined_ cong4 isDefined_ cong3 isDefined_ cong2 isDefined_ cong1
isDefined_ cong5 isDefined_ cong6 isDefined_ cong7 isDefined_ cong8
isDefined_ cong9 isDefined_ cong10 isDefined_ cong11 isDefined_ cong12
isDefined_ strongEq [THEN is_ TRUE_ charn[THEN iffD2]]
isDefined_ strictEq

lemmas *strong_ definedness = core_ definedness*
isDefined_ ifD0 isDefined_ andD0 isDefined_ impliesD0 isDefined_ orD0

lemmas *weak_ definedness = core_ definedness*

isDefined_ if isDefined_ implies
isDefined_ or isDefined_ and

thm *weak_ definedness*
thm *strong_ definedness*
thm *strictEq_ is_ strongEq_ LE*

lemmas *weak_ prop_ LJE =*
OclTrue_ is_ Valid OclFalse_ is_ Invalid OclUndefined_ is_ Invalid
OclTrue_ is_ LocalValid absurd6S absurd5S

localValidNot2not localValidAnd2conj
localValidOr2disj localValidImplies2impl

thm *isDefined_if* [*simplified is_TRUE_charn global_vs_local_validity*[*symmetric*]]

thm *weak_prop_LJE*

lemma *globalValidAnd2conj*:
$[\![\models \partial\ A; \models \partial\ B]\!] \Longrightarrow (\models A \wedge B) = ((\models A) \wedge (\models B))$
by (*auto simp*: *global_vs_local_validity*[*symmetric*] *localValidAnd2conj*)

lemma *globalValidNot2localValidNot*:
$\models \partial\ A \Longrightarrow (\models \neg\ A) = (\forall\ \tau.\ \neg\ (\tau \models A))$
by (*auto simp*: *global_vs_local_validity*[*symmetric*]
localValidNot2not[*symmetric*])

lemma *globalValidOr2localValidDisj*:
$[\![\models \partial\ A; \models \partial\ B]\!] \Longrightarrow (\models A \vee B) = (\forall\ \tau.\ (\tau \models A) \vee (\tau \models B))$
by (*auto simp*: *global_vs_local_validity*[*symmetric*]
localValidOr2disj [*symmetric*])

lemma *globalValidImplies2localValidimpl*:
$[\![\models \partial\ A; \models \partial\ B]\!] \Longrightarrow (\models A \longrightarrow B) = (\forall\ \tau.\ (\tau \models A) \longrightarrow (\tau \models B))$
by (*auto simp*: *global_vs_local_validity*[*symmetric*]
localValidImplies2impl [*symmetric*])

These previous equvalences demonstrate, why a “global judgement equivalence” calculus does not make sense. Desirable versions of these rules would be:

$$\models \partial A \Longrightarrow (\models \neg A) = (\neg(\models A)) \tag{B.12}$$
$$[\![\models \partial A; \models \partial B]\!] \Longrightarrow (\models A \vee B) = ((\models A) \vee (\models B)) \tag{B.13}$$
$$[\![\models \partial A; \models \partial B]\!] \Longrightarrow (\models A \longrightarrow B) = ((\models A) \longrightarrow (\models> B)) \tag{B.14}$$

This is not achievable since the logical disjunction not distribute over the universal quantifier hidden in the global validity statement.

Thus, a “global judgement equivalence” is not closed under global validity, such that any complete reasoning must be broken down on local judgements sooner or later.

end

B.4.5. Numerals

theory *OCL_Numerals*
imports
$*HOLOCL_HOME/src/OCL_Logic*

begin instance *up* :: (*number*)*number* **..**
instance *fun*:: (*type*,*number*)*number* **..**

OCL Ordering Interface

This is ony a syntactic interface for the ordering operators in OCL. Concrete definitions are provided in the structures Integer and Real.

```
consts
  OclLe          :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
  OclLess        :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
  OclGe          :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
  OclGreater     :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean


syntax
  _OclLe_std        :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/<= _) [63,64]63)
  _OclLess_std      :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/< _) [63,64]63)
  _OclGe_std        :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/>= _) [63,64]63)
  _OclGreater_std   :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/> _) [63,64]63)


syntax
  _OclLe_ascii      :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/'<=' _) [63,64]63)
  _OclLess_ascii    :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/'<' _) [63,64]63)
  _OclGe_ascii      :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/'>=' _) [63,64]63)
  _OclGreater_ascii:: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/'>' _) [63,64]63)

syntax (xsymbols)
  _OclLe_math       :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/≤ _) [63,64]63)
  _OclLess_math     :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/< _) [63,64]63)
  _OclGe_math       :: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/≥ _) [63,64]63)
  _OclGreater_math:: [('a,'b) VAL, ('a,'b) VAL] => 'a Boolean
                       ((1_/> _) [63,64]63)end
```

B.4.6. OCL Integer

```
theory OCL_Integer
imports
  $HOLOCL_HOME/src/library/basic/OCL_Numerals
begin constdefs
  is_Integer_0    :: 'a U ⇒ bool
  is_Integer_0    ≡ sum_case (λx. False)
                        (sum_case (λx. False)
                        (sum_case (λx. True)(λx. False)))
  get_Integer_0   :: 'a U ⇒ Integer_0
  get_Integer_0  ≡ sum_case (λx. ϵ x. True)
                         (sum_case  (λx. ϵ   x. True)
                         (sum_case (λx. x)(λx. ϵ x. True)))
  mk_Integer_0    :: Integer_0 ⇒ 'a U
  mk_Integer_0    ≡ Inr ∘ Inr ∘ Inl

lemma get_mk_Integer_id_0: get_Integer_0(mk_Integer_0 x) = x
  apply (simp add: get_Integer_0_def mk_Integer_0_def)
  done
lemma is_mk_Integer_0: is_Integer_0(mk_Integer_0 x) = True
  apply (simp add: is_Integer_0_def mk_Integer_0_def)
  done
lemma mk_get_Integer_id_0: is_Integer_0 x ⟹ mk_Integer_0(get_Integer_0 x) = x
  apply (simp add: get_Integer_0_def mk_Integer_0_def is_Integer_0_def )
  apply (case_tac x, simp, simp)
  apply (case_tac b, simp,simp)
  apply (case_tac ba, simp,simp)
  done


constdefs
   Integer_0 :: ('st, Integer_0 Set_0) VAL
  Integer_0 ≡ lift0(Abs_Set_0 ⌊lift ‘ UNIV⌋)

defs
  Zero_ocl_int_def: 0 ≡ lift₀( ⌊0::int⌋ )
  One_ocl_int_def:  1 ≡ lift₀( ⌊1::int⌋ )


lemma OCL_is_def_Zero_ocl_int [simp]:
  ∂ (0::'a Integer) = T
  apply(rule ext)
  apply(simp add: OclIsDefined_def Zero_ocl_int_def OclTrue_def ss_lifting')
  done

lemma OCL_is_def_One_ocl_int [simp]:
  ∂ (1::'a Integer) = T
  apply(rule ext)
```

apply(*simp add: OclIsDefined_def One_ocl_int_def OclTrue_def ss_lifting'*)
done

defs (**overloaded**)
number_of_def : $(number\_of :: bin \Rightarrow {'a}\ Integer)$
$\equiv \lambda\ b.\ lift_0(\ \llcorner(number\_of::bin \Rightarrow int)\ b\lrcorner\ )$

defs
plus_def: $op + \equiv lift_2(strictify(\lambda\ x::Integer\_0.\ strictify(\lambda\ y.$
$\llcorner(\ \ulcorner x\urcorner\ )+(\ \ulcorner y\urcorner\ )\lrcorner\ )))$

minus_def: $op - \equiv lift_2(strictify(\lambda\ x::Integer\_0.\ strictify(\lambda\ y.$
$\llcorner(\ \ulcorner x\urcorner\ )-(\ \ulcorner y\urcorner\ )\lrcorner\ )))$

times_def: $op * \equiv lift_2(strictify(\lambda\ x::Integer\_0.\ strictify(\lambda\ y.$
$\llcorner(\ \ulcorner x\urcorner\ )*(\ \ulcorner y\urcorner\ )\lrcorner\ )))$

ocl_setup_op [*plus*, *minus*, *times*]

lemma *plus_zero*: $(X::{'a}\ Integer) + 0 = X$
by(*rule ext*, *simp add*: *plus_def Zero_ocl_int_def ss_lifting'*)

lemma *plus_commute*: $(X::{'a}\ Integer) + Y = Y + X$
by(*rule ext*, *simp add*: *plus_def ss_lifting'*)

defs
OclLe_def: $OclLe \equiv lift_2(strictify(\lambda x::Integer\_0.$
$strictify(\lambda y.\ \llcorner(\ \ulcorner x\urcorner\ ) <= (\ \ulcorner y\urcorner\ )\lrcorner\ )))$
OclLess_def: $OclLess \equiv lift_2(strictify(\lambda\ x::Integer\_0.$
$strictify(\lambda\ y.\ \llcorner(\ \ulcorner x\urcorner\ ) < (\ \ulcorner y\urcorner\ )\lrcorner\ )))$
OclGe_def: $OclGe \equiv lift_2(strictify(\lambda\ x::Integer\_0.$
$strictify(\lambda\ y.\ \llcorner\neg(\ \ulcorner x\urcorner\ ) < (\ \ulcorner y\urcorner\ )\lrcorner\ )))$
OclGreater_def: $OclGreater \equiv lift_2(strictify(\lambda\ x::Integer\_0.$
$strictify(\lambda\ y.\ \llcorner\neg(\ \ulcorner x\urcorner\ )<=(\ \ulcorner y\urcorner\ )\lrcorner\ )))$

ocl_setup_op [*OclLe*, *OclLess*, *OclGe*, *OclGreater*]

lemma *OclGe_elim*[*simp*] : $OclGe\ (X::{'a}\ Integer)\ Y = OclLe\ Y\ X$
apply(*rule ext*)
apply(*case_tac x* $\vDash\ \partial(X)$,*simp_all*, *ocl_hypsubst* [*2*], *simp_all*)
apply(*case_tac x* $\vDash\ \partial(Y)$,*simp_all*, *ocl_hypsubst* [*2*], *simp_all*)
apply(*simp add*: *OclGe_def OclLe_def*
OclIsDefined_def strictify_def OclLocalValid_def

```
              o_def lift0_def lift1_def lift2_def
              OclTrue_def OclUndefined_def DEF_def)
apply(simp add: Orderings.linorder_not_less)
done


lemma OclGreater_elim[simp] : OclGreater (X::'a Integer) Y =  OclLess Y X
apply(rule ext)
apply(case_tac x ⊨ ∂(X),simp_all, ocl_hypsubst [2], simp_all)
apply(case_tac x ⊨ ∂(Y),simp_all, ocl_hypsubst [2], simp_all)
apply(simp add:  OclGreater_def OclLess_def
              OclIsDefined_def strictify_def OclLocalValid_def
              o_def lift0_def lift1_def lift2_def
              OclTrue_def OclUndefined_def DEF_def)
apply(simp add: Orderings.linorder_not_le)
done


lemma OclLe_compute [simp]:
     OclLe((number_of:: bin ⇒ 'a Integer) a)((number_of:: bin ⇒ 'a Integer) b) =
      (if ¬ neg ((number_of:: bin ⇒ int) (bin_add b (bin_minus a)))then T else F)
apply (rule ext)
apply (simp add: number_of_def OclLe_def
              OclIsDefined_def strictify_def
              o_def lift0_def lift1_def lift2_def
              OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done


lemma OclLess_compute [simp]:
     OclLess((number_of:: bin ⇒ 'a Integer) a)((number_of:: bin ⇒ 'a Integer) b) =
      (if neg ((number_of:: bin ⇒ int) (bin_add a (bin_minus b)))then T else F)
apply (rule ext)
apply (simp add: number_of_def OclLess_def
              OclIsDefined_def strictify_def
              o_def lift0_def lift1_def lift2_def
              OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done


consts

  OclMod   :: ['a Integer, 'a Integer] ⇒ 'a Integer
            (_ ->mod'( _ ') [66,65]65)

  OclDiv   :: ['a Integer, 'a Integer] ⇒ 'a Integer
            (_ ->div'( _ ') [66,65]65)

  OclRound :: 'a Integer ⇒ 'a Integer
```

$(\_ \; {-}{>}round'(\; ') \; [66])$

$OclFloor \;::\; 'a \; Integer \Rightarrow 'a \; Integer$
$(\_ \; {-}{>}floor'(\; ') \; [66])$

$OclMin \;::\; ['a \; Integer, 'a \; Integer] \Rightarrow 'a \; Integer$
$(\_ \; {-}{>}min'(\; \_ \; ') \; [66,65]65)$

$OclMax \;::\; ['a \; Integer, 'a \; Integer] \Rightarrow 'a \; Integer$
$(\_ \; {-}{>}max'(\; \_ \; ') \; [66,65]65)$

$OclAbs \;::\; 'a \; Integer \Rightarrow 'a \; Integer$
$(\_ \; {-}{>}abs'(\; ') \; [66])$

$OclNegative \;::\; 'a \; Integer \Rightarrow 'a \; Integer$
$(- \; \_ \; [66])$

defs
$OclRound\_def: SELF \; {-}{>}round() \equiv SELF$
$OclFloor\_def: SELF \; {-}{>}floor() \equiv SELF$

defs
$OclMod\_def: OclMod \equiv lift_2(strictify(\lambda \; x{::}Integer\_0. \; strictify(\lambda \; y.$
$if \ulcorner y \urcorner = 0 \; then \; down \; else$
$\llcorner (\ulcorner x \urcorner) \; mod \; (\ulcorner y \urcorner) \lrcorner)))$

$OclDiv\_def: OclDiv \equiv lift_2(strictify(\lambda \; x{::}Integer\_0. \; strictify(\lambda \; y.$
$if \ulcorner y \urcorner = 0 \; then \; down \; else$
$\llcorner (\ulcorner x \urcorner) \; div \; (\ulcorner y \urcorner) \lrcorner)))$

$OclMin\_def: SELF \; {-}{>}min(I) \equiv if \; (SELF \leq I) \; then \; SELF \; else \; I \; endif$

$OclMax\_def: SELF \; {-}{>}max(I) \equiv if \; (SELF \geq I) \; then \; SELF \; else \; I \; endif$

$OclAbs\_def: OclAbs \equiv lift_1(strictify(\lambda \; x{::}Integer\_0.$
$\llcorner abs \; (\ulcorner x \urcorner) \lrcorner))$

$OclNegative\_def: OclNegative \equiv lift_1(strictify(\lambda \; x{::}Integer\_0.$
$\llcorner (- (\ulcorner x \urcorner)) \lrcorner))$

ocl_setup_op $[OclMod, OclDiv, OclAbs, OclNegative]$

Computational Rules

Foundational rule over Integer values in OCL:

lemma *Integer_values_defined*[*simp*]:
  ∂((*number_of*:: *bin* ⇒ ′*a Integer*) *a*) = ⊤
apply (*rule ext*)
by (*simp add*: *number_of_def*
    OclIsDefined_def strictify_def
    o_def lift0_def lift1_def lift2_def
    OclTrue_def OclUndefined_def DEF_def)

These two elementary cases are required by the standard:

lemma *div_zero*: *x* −>*div*(*0*) = ⊥
apply (*rule ext*)
apply (*simp add*: *OclDiv_def Zero_ocl_int_def*
    OclIsDefined_def strictify_def
    o_def lift0_def lift1_def lift2_def
    OclTrue_def OclUndefined_def DEF_def)
done

lemma *mod_zero*: *x* −>*mod*(*0*) = ⊥
apply (*rule ext*)
apply (*simp add*: *OclMod_def Zero_ocl_int_def*
    OclIsDefined_def strictify_def
    o_def lift0_def lift1_def lift2_def
    OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma *plus_compute* [*simp*]:
  ((*number_of*:: *bin* ⇒ ′*a Integer*) *a*) + ((*number_of*:: *bin* ⇒ ′*a Integer*) *b*) =
  number_of (*bin_add a b*)
apply (*rule ext*)
apply (*simp add*: *number_of_def plus_def*
    OclIsDefined_def strictify_def
    o_def lift0_def lift1_def lift2_def
    OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma *3* + *4* = (*7*::′*a Integer*) **by** *simp*

lemma *times_compute* [*simp*]:
  ((*number_of*:: *bin* ⇒ ′*a Integer*) *a*) ∗ ((*number_of*:: *bin* ⇒ ′*a Integer*) *b*) =
  number_of (*bin_mult a b*)
apply (*rule ext*)
apply (*simp add*: *number_of_def times_def*
    OclIsDefined_def strictify_def
    o_def lift0_def lift1_def lift2_def
    OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma *minus_compute* [*simp*]:
$((number\_of{::}\ bin \Rightarrow {'a}\ Integer)\ a) - ((number\_of{::}\ bin \Rightarrow {'a}\ Integer)\ b) =$
$(number\_of\ ((bin\_add\ a\ (bin\_minus\ b))))$
apply (*rule ext*)
apply (*simp add*: *number_of_def minus_def*
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma *strong_eq_compute* [*simp*]:
$\neg\ iszero((number\_of{::}bin \Rightarrow int)\ (bin\_add\ a\ (bin\_minus\ b))) \Longrightarrow$
$(((number\_of{::}bin{\Rightarrow}{'a}\ Integer)a) \triangleq ((number\_of{::}bin \Rightarrow{'a}\ Integer)b))$
$= \mathsf{F}$
apply (*rule ext*)
apply (*simp add*: *number_of_def OclStrongEq_def*
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma *strict_eq_compute* [*simp*]:
$\neg\ iszero((number\_of{::}bin \Rightarrow int)\ (bin\_add\ a\ (bin\_minus\ b))) \Longrightarrow$
$(((number\_of{::}bin{\Rightarrow}{'a}\ Integer)a) \doteq ((number\_of{::}bin \Rightarrow{'a}\ Integer)b))$
$= \mathsf{F}$
apply (*rule ext*)
apply (*simp add*: *number_of_def OclStrictEq_def*
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma $[\![\ \tau \models 3 \triangleq C\ ;\ \tau \models C + 2 \doteq (7{::}{'a}\ Integer)]\!] \Longrightarrow \tau \models A \wedge B$
apply *ocl_hypsubst*
apply *simp*
done

end

B.4.7. OCL Real

theory *OCL_Real*

```
imports
  $HOLOCL_HOME/src/library/basic/OCL_Numerals
begin constdefs
  is_Real_0       :: 'a U ⇒ bool
  is_Real_0       ≡  sum_case  (λx. False)
                       (sum_case (λx. True) (λx. False))
  get_Real_0      :: 'a U ⇒ Real_0
  get_Real_0      ≡  sum_case (λx. ϵ  x. True)
                       (sum_case (λx. x) (λx. ϵ x. True))
  mk_Real_0       :: Real_0 ⇒ 'a U
  mk_Real_0       ≡  Inr o Inl

lemma get_mk_Real_id_0 : get_Real_0(mk_Real_0 x) = x
  apply (simp add: get_Real_0_def mk_Real_0_def)
  done
lemma is_mk_Real_0 : is_Real_0(mk_Real_0 x) = True
  apply (simp add: is_Real_0_def mk_Real_0_def)
  done
lemma mk_get_Real_id_0 : is_Real_0 x ⟹ mk_Real_0(get_Real_0 x) = x
  apply (simp add: get_Real_0_def mk_Real_0_def is_Real_0_def )
  apply (case_tac x, simp, simp)
  apply (case_tac b, simp, simp)
  done

constdefs
   Real_0    :: ('st, Real_0 Set_0)     VAL
  Real_0     ≡ lift0(Abs_Set_0 ⌞lift ‘ UNIV⌟)

defs
  OclZero_def: 0 ≡ lift₀( ⌞0::real⌟ )
  OclOne_def : 1 ≡ lift₀( ⌞1::real⌟ )

defs
  plus_def   : op + ≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                         ⌞( ⌜x⌝ )+( ⌜y⌝ )⌟ )))
  minus_def  : op − ≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                         ⌞( ⌜x⌝ )−( ⌜y⌝ )⌟ )))
  times_def  : op ∗ ≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                         ⌞( ⌜x⌝ )∗( ⌜y⌝ )⌟ )))

ocl_setup_op [plus, minus, times]

consts
  OclRound   :: 'a Real => 'a Integer          (_ −>round'( ') [66])
  OclFloor   :: 'a Real => 'a Integer          (_ −>floor'( ') [66])

  OclMin     :: ['a Real, 'a Real] => 'a Real  (_ −>min'( _  ') [66,65]65)
  OclMax     :: ['a Real, 'a Real] => 'a Real  (_ −>max'( _  ') [66,65]65)
```

```
  OclDivide  :: ['a Real, 'a Real] => 'a Real
                                        (_ ->divide'( _ ')[66,65]65)

  OclAbs     :: 'a Real => 'a Real         (_ ->abs'( ') [66])
  OclNegative:: 'a Real => 'a Real         (- _ [66])


defs
  OclDivide_def:
   OclDivide ≡ lift₂ (strictify(λ x::Real_0. strictify(λ y::Real_0.
                                 ( ⌞ (⌜x⌝ / ⌜y⌝) ⌟ ))))

defs
  OclLe_def:
  OclLe      ≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                                 ⌞ ( ⌜x⌝ ) <= ( ⌜y⌝ )⌟ )))
  OclLess_def:
  OclLess    ≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                                 ⌞ ( ⌜x⌝ ) < ( ⌜y⌝ )⌟ )))
  OclGe_def:
  OclGe      ≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                                 ⌞ ¬( ⌜x⌝ ) < ( ⌜y⌝ )⌟ )))
  OclGreater_def:
  OclGreater≡ lift₂(strictify(λ x::Real_0. strictify(λ y.
                                 ⌞ ¬( ⌜x⌝ ) <= ( ⌜y⌝ )⌟ )))

  OclMin_def:
  SELF ->min(I) ≡ if (SELF `<=` I) then SELF else I endif

  OclMax_def:
  SELF ->max(I) ≡ if (SELF `>=` I) then SELF else I endif

  OclAbs_def:
  OclAbs         ≡ lift₁(strictify(λ x::Real_0.
                                 ⌞ abs ( ⌜x⌝ ) ⌟ ))

  OclNegative_def:
  OclNegative    ≡ lift₁(strictify(λ x::Real_0.
                                 ⌞ ( - ( ⌜x⌝ )) ⌟ ))

ocl_setup_op [OclMax, OclMin, OclNegative]
ocl_setup_op [OclLe, OclLess, OclGe, OclGreater]

end
```

B.4.8. OCL String

```
theory OCL_String
```

```
imports
  $HOLOCL_HOME/src/library/basic/OCL_Integer
begin constdefs
  is_String_0     :: 'a U ⇒ bool
  is_String_0     ≡ sum_case (λx. False)
                          (sum_case (λx. False)
                          (sum_case (λx. False)
                          (sum_case (λx. False)(λx. True))))

  get_String_0    :: 'a U ⇒ String_0
  get_String_0  ≡ sum_case (λx. ϵ x. True)
                          (sum_case (λx. ϵ x. True)
                          (sum_case (λx. ϵ x. True)
                          (sum_case (λx. ϵ x. True)(λx. x))))
  mk_String_0     :: String_0 ⇒ 'a U
  mk_String_0     ≡ Inr ∘ Inr ∘ Inr o Inr

lemma get_mk_String_id_0: get_String_0(mk_String_0 x) = x
  apply (simp add: get_String_0_def mk_String_0_def)
  done
lemma is_mk_String_0: is_String_0(mk_String_0 x) = True
  apply (simp add: is_String_0_def mk_String_0_def)
  done
lemma mk_get_String_id_0: is_String_0 x ⟹ mk_String_0(get_String_0 x) = x
  apply (simp add: get_String_0_def mk_String_0_def is_String_0_def )
  apply (case_tac x, simp, simp)
  apply (case_tac b, simp, simp)
  apply (case_tac ba, simp, simp)
  apply (case_tac bb, simp, simp)
 done

constdefs
   String_0  :: ('st, String_0 Set_0)   VAL
  String_0  ≡ lift0(Abs_Set_0 ⌞lift ‘ UNIV⌟)

consts
  OclConcat     :: ['a String, 'a String] ⇒ 'a String   (_ −>concat'( _ ') [66,65]65)
  OclToLower    :: 'a String ⇒ 'a String                 (_ −>toLower'( _ ') [66,65]65)
  OclToUpper    :: 'a String ⇒ 'a String                 (_ −>toUpper'( _ ') [66,65]65)
  stringsize    :: 'a String ⇒ 'a Integer              (_ −>size'( ') [66])
  OclSubstring :: ['a String, 'a Integer, 'a Integer] ⇒ 'a String
                                                        (_ −>subString'(_ _')[66,65,65])

defs
  OclConcat_def:    OclConcat     ≡ lift₂(strictify(λ x::String_0.
                                        strictify(λ y. ⌞⌜x⌝ @ ⌜y⌝⌟)))
  stringsize_def:   stringsize    ≡ lift₁(strictify(λ x::String_0. ⌞ int(size ( ⌜x⌝ )) ⌟ ))

  OclSubstring_def:OclSubstring ≡ lift₃(strictify(λ s::String_0.
```

$$\begin{array}{l}
\qquad\qquad\qquad \mathit{strictify}(\lambda\ l{::}\mathit{Integer_0}. \\
\qquad\qquad\qquad \mathit{strictify}(\lambda\ u{::}\mathit{Integer_0}. \\
\llcorner\ (\mathit{List.take}\ (\mathit{nat}((\ \ulcorner u \urcorner\ -\ \ulcorner l \urcorner\ +\ 1{::}\mathit{int}\ )))\\
\quad (\mathit{List.drop}\ (\mathit{nat}(\ \ulcorner l \urcorner\ -\ 1{::}\mathit{int}))\ \ulcorner s \urcorner\ ))\ \lrcorner\ )\ )))
\end{array}$$

end

B.4.9. OCL Collection

theory *OCL_Collection*
imports
 $HOLOCL_HOME/src/library/basic/OCL_Integer
begin

The purpose of this theory is to provide the syntactic framework for the abstract class `Collection` in the sense of the standard. It is defined by the Isabelle type class "collection". We require that collections must always have a bottom element.

Unfortunately, this implementation by the type class "collection" is an approximation (a type constructor class would be more appropriate, but is not available in the Isabelle type system). Consequently, in lemmas the concrete type instances for operators resulting from the abstract class must be given; the types inferred by Isabelle automatically are too coarse and can lead to dead ends in proofs (definitions can not be unfolded).

The Syntax of Abstract Operations

The core operation *size* (which may be undefined if and only if the collection is infinitely large), *select*, *collect*, *includes* and *any*.

Operations on all Collection Types

consts

$$\begin{array}{lll}
\mathit{OclSize} & :: & ('\tau,'\alpha{::}\mathit{collection})\,\mathit{VAL} \Rightarrow\ '\tau\ \mathit{Integer} \\
\mathit{OclCount} & :: & [('\tau,'\beta{::}\mathit{collection})\,\mathit{VAL},('\tau,'\alpha{::}\mathit{bot})\ \mathit{VAL}] \Rightarrow\ '\tau\ \mathit{Integer} \\
\mathit{OclIncludes} & :: & [('\tau,'\beta{::}\mathit{collection})\,\mathit{VAL},('\tau,'\alpha{::}\mathit{bot})\ \mathit{VAL}] \Rightarrow\ '\tau\ \mathit{Boolean} \\
\mathit{OclExcludes} & :: & [('\tau,'\beta{::}\mathit{collection})\,\mathit{VAL},('\tau,'\alpha)\ \mathit{VAL}] \Rightarrow\ '\tau\ \mathit{Boolean} \\
\mathit{OclIncluding} & :: & [('\tau,'\beta{::}\mathit{collection})\,\mathit{VAL},('\tau,'\alpha)\ \mathit{VAL}] \Rightarrow ('\tau,'\beta)\ \mathit{VAL} \\
\mathit{OclExcluding} & :: & [('\tau,'\beta{::}\mathit{collection})\,\mathit{VAL},('\tau,'\alpha)\ \mathit{VAL}] \Rightarrow ('\tau,'\beta)\ \mathit{VAL} \\
\mathit{OclFlatten} & :: & ('\tau,'\alpha{::}\mathit{collection})\,\mathit{VAL} \Rightarrow ('\tau,'\beta{::}\mathit{bot})\ \mathit{VAL} \\
\mathit{OclSum} & :: & ('\tau,'\alpha{::}\mathit{collection})\,\mathit{VAL} \Rightarrow\ '\tau\ \mathit{Integer} \\
\mathit{OclAsSet} & :: & ('\tau,'\alpha{::}\mathit{collection})\,\mathit{VAL} \Rightarrow ('\tau,'\beta{::}\mathit{collection})\ \mathit{VAL} \\
\mathit{OclAsSequence} & :: & ('\tau,'\alpha{::}\mathit{collection})\,\mathit{VAL} \Rightarrow ('\tau,'\beta{::}\mathit{collection})\ \mathit{VAL}
\end{array}$$

$OclAsBag$ :: $('\tau,'\alpha::collection)\ VAL \Rightarrow ('\tau,'\beta::collection)\ VAL$
$OclAsOrderedSet$:: $('\tau,'\alpha::collection)\ VAL \Rightarrow ('\tau,'\beta::collection)\ VAL$
$OclIncludesAll$ :: $[('\tau,'\alpha::collection)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$OclExcludesAll$ :: $[('\tau,'\alpha::collection)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow '\tau\ Boolean$
$OclIsEmpty$ :: $('\tau,'\alpha::collection)\ VAL \Rightarrow '\tau\ Boolean$
$OclNotEmpty$ :: $('\tau,'\alpha::collection)\ VAL \Rightarrow '\tau\ Boolean$
$OclComplement$ :: $('\tau,'\beta::collection)\ VAL \Rightarrow ('\tau,'\beta)\ VAL$
$OclUnion$ :: $[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\beta)\ VAL] \Rightarrow ('\tau,'\beta)\ VAL$
$OclCollectionRange$ :: $[('\tau,'\alpha)\ VAL,('\tau,'\alpha)\ VAL] \Rightarrow ('\tau,'\beta)\ VAL$

syntax

_ $OclSize$_ std :: $('\tau,\ '\alpha::collection)\ VAL$ => $'\tau\ Integer$
(_ `->size()`'(') [66])

_ $OclCount$_ std :: $[('\tau,'\beta::collection)\ VAL,('\tau,'\alpha::bot)\ VAL]$ => $'\tau\ Integer$
(_ `->count`'(_ ') [66,65]65)

_ $OclIncludes$_ std :: $[\ ('\tau,\ '\beta::collection)\ VAL,('\tau,\ '\alpha)\ VAL]$ => $'\tau\ Boolean$
(_ `->includes`'(_ ') [66,65]65)

_ $OclExcludes$_ std :: $[('\tau,\ '\beta::collection)\ VAL,('\tau,\ '\alpha::bot)\ VAL]$ => $'\tau\ Boolean$
(_ `->excludes`'(_ ') [66,65]65)

_ $OclFlatten$_ std :: $('\tau,\ '\alpha::collection)\ VAL$ => $('\tau,\ '\beta::collection)\ VAL$
(_ `->flatten`'(') [66])

_ $OclSum$_ std :: $('\tau,\ '\alpha::collection)\ VAL$ => $'\tau\ Integer$
(_ `->sum()`'(') [66])

_ $OclAsSet$_ std :: $('\tau,\ '\alpha::collection)\ VAL$ => $('\tau,\ '\beta::collection)\ VAL$
(_ `->asSet()`'(') [66])

_ $OclAsSequence$_ std:: $('\tau,\ '\alpha::collection)\ VAL$ => $('\tau,\ '\beta::collection)\ VAL$
(_ `->asSequence()`'(') [66])

_ $OclAsBag$_ std :: $('\tau,\ '\alpha::collection)\ VAL$ => $('\tau,\ '\beta::collection)\ VAL$
(_ `->asBag()`'(') [66])

_ $OclAsOrderedSet$_ std:: $('\tau,\ '\alpha::collection)\ VAL$ => $('\tau,\ '\beta::collection)\ VAL$
(_ `->asOrderedSet()`'(') [66])

_ $OclIncludesAll$_ std:: $[('\tau,'\alpha::collection)\ VAL,('\tau,'\alpha::collection)\ VAL]$ => $'\tau\ Boolean$
(_ `->includesAll`'(_ ') [66,65]65)

_ $OclExcludesAll$_ std:: $[('\tau,'\alpha::collection)\ VAL,('\tau,'\alpha::collection)\ VAL]$ => $'\tau\ Boolean$
(_ `->excludesAll`'(_ ') [66,65]65)

_ $OclIsEmpty$_ std :: $('\tau,\ '\alpha::collection)\ VAL$ => $'\tau\ Boolean$
(_ `->isEmpty()`'(') [66])

```
_OclNotEmpty_std   :: ('τ, 'α::collection) VAL => 'τ Boolean
                      (_ ->notEmpty()'(') [66])

_OclIncluding_std  :: [('τ,'β::collection) VAL,('τ,'α::bot) VAL] ⇒ ('τ,'β) VAL
                      (_ ->including'( _ '))

_OclExcluding_std  :: [('τ,'β::collection) VAL,('τ,'α::bot) VAL] ⇒ ('τ,'β) VAL
                      (_ ->excluding'( _ '))

_OclComplement_std :: ('τ, 'β::collection) VAL => ('τ, 'β) VAL
                      (_ ->complement()'('))

_OclUnion_std      :: [('τ,'β::collection) VAL, ('τ,'β) VAL] => ('τ,'β) VAL
                      (_ ->union'( _ ')         [66,65]65)

syntax
  _OclSize_ascii     :: ('τ, 'α::collection) VAL => 'τ Integer
                        (_ ->size'(') [66])

  _OclCount_ascii    :: [('τ,'β::collection) VAL,('τ,'α::bot) VAL] => 'τ Integer
                        (_ ->count'(_') [66,65]65)

  _OclIncludes_ascii :: [ ('τ, 'β::collection) VAL,('τ, 'α) VAL] => 'τ Boolean
                        (_ ->includes'(_') [66,65]65)

  _OclExcludes_ascii :: [('τ, 'β::collection) VAL,('τ, 'α::bot) VAL] => 'τ Boolean
                        (_ ->excludes'(_') [66,65]65)

  _OclFlatten_ascii  :: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
                        (_ ->flatten'(') [66])

  _OclSum_ascii      :: ('τ, 'α::collection) VAL => 'τ Integer
                        (_ ->sum'(') [66])

  _OclAsSet_ascii    :: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
                        (_ ->asSet'(') [66])

  _OclAsSequence_ascii:: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
                        (_ ->asSequence'(') [66])

  _OclAsBag_ascii    :: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
                        (_ ->asBag'(') [66])

  _OclAsOrderedSet_ascii:: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
                        (_ ->asOrderedSet'(') [66])

  _OclIncludesAll_ascii :: [('τ,'α::collection) VAL,('τ,'α::collection) VAL] => 'τ Boolean
                        (_ ->includesAll'(_') [66,65]65)
```

$\_OclExcludesAll\_ascii$ ::[('τ,'α::*collection*) *VAL*,('τ,'α::*collection*) *VAL*] => 'τ *Boolean*
(_ −>*excludesAll*'(_ ') [66,65]65)

$\_OclIsEmpty\_ascii$ :: ('τ, 'α::*collection*) *VAL* => 'τ *Boolean*
(_ −>*isEmpty*'(') [66])

$\_OclNotEmpty\_ascii$ :: ('τ, 'α::*collection*) *VAL* => 'τ *Boolean*
(_ −>*notEmpty*'(') [66])

$\_OclIncluding\_ascii$ :: [('τ,'β::*collection*) *VAL*,('τ,'α::*bot*) *VAL*] ⇒ ('τ,'β) *VAL*
(_ −>*including*'(_ '))

$\_OclExcluding\_ascii$ :: [('τ,'β::*collection*) *VAL*,('τ,'α::*bot*) *VAL*] ⇒ ('τ,'β) *VAL*
(_ −>*excluding*'(_ '))

$\_OclComplement\_ascii$ :: ('τ, 'β::*collection*) *VAL* => ('τ, 'β) *VAL*
(_ −>*complement*'('))

$\_OclUnion\_ascii$ :: [('τ,'β::*collection*) *VAL*, ('τ,'β) *VAL*] => ('τ,'β) *VAL*
(_ −>*union*'(_ ') [66,65]65)

$\_OclIntersection\_ascii$:: [('τ,'β::*collection*) *VAL*, ('τ,'β) *VAL*] => ('τ,'β) *VAL*
(_ −>*intersection*'(_ ') [71,70]70)

syntax (*xsymbols*)
$\_OclSize\_math$ :: ('τ, 'α::*collection*) *VAL* => 'τ *Integer*
(‖_‖ [66])

$\_OclCount\_math$ :: [('τ,'β::*collection*) *VAL*,('τ,'α::*bot*) *VAL*] => 'τ *Integer*
(_ `->count` _ [66,65]65)

$\_OclIncludes\_math$ :: [('τ, 'α) *VAL*, ('τ, 'β::*collection*) *VAL*] => 'τ *Boolean*
(_ ∈ _ [66,65]65)

$\_OclExcludes\_math$ :: [('τ, 'α::*bot*) *VAL*, ('τ, 'β::*collection*) *VAL*] => 'τ *Boolean*
(_ ∉ _ [66,65]65)

$\_OclFlatten\_math$ :: ('τ, 'α::*collection*) *VAL* => ('τ, 'β::*collection*) *VAL*
(⟦_⟧ [66])

$\_OclSum\_math$ :: ('τ, 'α::*collection*) *VAL* => 'τ *Integer*
(`->sum()` _ [66])

$\_OclAsSet\_math$ :: ('τ, 'α::*collection*) *VAL* => ('τ, 'β::*collection*) *VAL*
(`->asSet()` _ [66])

$\_OclAsSequence\_math$:: ('τ, 'α::*collection*) *VAL* => ('τ, 'β::*collection*) *VAL*

(->asSequence() _ [66])

_ OclAsBag_ math :: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
(->asBag() _ [66])

_ OclAsOrderedSet_ math:: ('τ, 'α::collection) VAL => ('τ, 'β::collection) VAL
(->asOrderedSet() _ [66])

_ OclIncludesAll_ math :: [('τ,'α::collection) VAL,('τ,'α::collection) VAL] => 'τ Boolean
(_ ⊆'(_') [66,65]65)

_ OclExcludesAll_ math ::[('τ,'α::collection) VAL,('τ,'α::collection) VAL] => 'τ Boolean
(_ ⊄'(_') [66,65]65)

_ OclIsEmpty_ math :: ('τ, 'α::collection) VAL => 'τ Boolean
(≐ ∅_ [66])

_ OclNotEmpty_ math :: ('τ, 'α::collection) VAL => 'τ Boolean
(≠ ∅_ [66])

_ OclIncluding_ math :: [('τ,'β::collection) VAL,('τ,'α::bot) VAL] ⇒ ('τ,'β) VAL
(_ ->including _)

_ OclExcluding_ math :: [('τ,'β::collection) VAL,('τ,'α::bot) VAL] ⇒ ('τ,'β) VAL
(_ ->excluding _)

_ OclComplement_ math :: ('τ, 'β::collection) VAL => ('τ, 'β) VAL
(_ $^{-1}$)

_ OclUnion_ math :: [('τ,'β::collection) VAL, ('τ,'β) VAL] => ('τ,'β) VAL
(_ ∪ _ [66,65]65)

Operations on Collection Types with Ordering

consts

OclPrepend :: [('τ,('α::collection)) VAL, ('τ,'β::bot) VAL] => ('τ,'α) VAL
OclAppend :: [('τ,('α::collection)) VAL, ('τ,'β::bot) VAL] => ('τ,'α) VAL
OclInsertAt :: [('τ,'α::collection) VAL,'τ Integer,('τ,'β) VAL] => ('τ,'α) VAL
OclAt :: [('τ,'α::collection) VAL,'τ Integer] => ('τ, 'β) VAL
OclIndexOf :: [('τ,'α::collection) VAL,('τ, 'β) VAL] => 'τ Integer
OclFirst :: ('τ, 'α::collection) VAL => ('τ,'β) VAL
OclLast :: ('τ, 'α::collection) VAL => ('τ,'β) VAL

syntax

_ OclPrepend_ std :: [('τ,('α::collection)) VAL, ('τ,'β::bot) VAL] => ('τ,'α) VAL
(_ ->prepend'(_ ') [66,65]65)
_ OclAppend_ std :: [('τ,('α::collection)) VAL, ('τ,'β::bot) VAL] => ('τ,'α) VAL

```
                    (_ ->append'( _ ') [66,65]65)
  _OclInsertAt_std :: [('τ,'α::bot)Sequence,'τ Integer,('τ,'α) VAL] => ('τ,'α )Sequence
                    (_ ->insertAt'( _ , _ ') [66,65,65])
  _OclAt_std       :: [('τ,'α::collection ) VAL,'τ Integer] => ('τ, 'β) VAL
                    (_ ->at'( _ ') [66,65])
  _OclIndexOf_std  :: [('τ,'α::collection ) VAL,('τ, 'β) VAL] => 'τ Integer
                    (_ ->indexOf '( _ ') [66,65])
  _OclFirst_std    :: ('τ, 'α::collection ) VAL => ('τ,'β) VAL
                    (_ ->first() [66])
  _OclLast_std     :: ('τ, 'α::collection) VAL => ('τ,'β) VAL
                    (_ ->last() [66])
```

syntax

```
  _OclPrepend_ascii  :: [('τ,('α::collection) ) VAL, ('τ,'β::bot ) VAL] => ('τ,'α ) VAL
                      (_ ->prepend'( _ ') [66,65]65)
  _OclAppend_ascii   :: [('τ,('α::collection) ) VAL, ('τ,'β::bot ) VAL] => ('τ,'α ) VAL
                      (_ ->append'( _ ') [66,65]65)
  _OclInsertAt_ascii :: [('τ,'α::bot)Sequence,'τ Integer,('τ,'α) VAL] => ('τ,'α )Sequence
                      (_ ->insertAt'( _ , _ ') [66,65,65])
  _OclAt_ascii       :: [('τ,'α::collection ) VAL,'τ Integer] => ('τ, 'β) VAL
                      (_ ->at'( _ ') [66,65])
  _OclIndexOf_ascii  :: [('τ,'α::collection ) VAL,('τ, 'β) VAL] => 'τ Integer
                      (_ ->indexOf'( _ ') [66,65])
  _OclFirst_ascii    :: ('τ, 'α::collection ) VAL => ('τ,'β) VAL
                      (_ ->first'(') [66])
  _OclLast_ascii     :: ('τ, 'α::collection) VAL => ('τ,'β) VAL
                      (_ ->last'(') [66])
```

syntax (*xsymbols*)

```
  _OclPrepend_math  :: [('τ,('α::collection) ) VAL, ('τ,'β::bot ) VAL] => ('τ,'α ) VAL
                     (_ : _  [66,65]65)
  _OclAppend_math   :: [('τ,('α::collection) ) VAL, ('τ,'β::bot ) VAL] => ('τ,'α ) VAL
                     (_ :: _  [66,65]65)
  _OclInsertAt_math :: [('τ,'α::bot)Sequence,'τ Integer,('τ,'α) VAL] => ('τ,'α )Sequence
                     (_ ->insertAt '( _ , _ ') [66,65,65])
  _OclAt_math       :: [('τ,'α::collection ) VAL,'τ Integer] => ('τ, 'β) VAL
                     (_ ♮ _  [66,65])
  _OclIndexOf_math  :: [('τ,'α::collection ) VAL,('τ, 'β) VAL] => 'τ Integer
                     (_ ♮? _  [66,65])
  _OclFirst_math    :: ('τ, 'α::collection ) VAL => ('τ,'β) VAL
                     (♮1 _  [66])
  _OclLast_math     :: ('τ, 'α::collection) VAL => ('τ,'β) VAL
                     (♮$ _  [66])
```

Operations on Collection Types without Ordering

consts

```
  OclIntersection:: [('τ,'β::collection) VAL, ('τ,'β) VAL] ⇒ ('τ,'β) VAL
```

syntax
  $\_\,OclIntersection\_std$:: $[('\tau,'\beta::collection)\,VAL,\ ('\tau,'\beta)\,VAL] => ('\tau,'\beta)\,VAL$
  (_ `->intersection'( _ ')`   [71,70] 70)

syntax
  $\_\,OclIntersection\_ascii$:: $[('\tau,'\beta::collection)\,VAL,\ ('\tau,'\beta)\,VAL] => ('\tau,'\beta)\,VAL$
  (_ −>*intersection'*(_ ')   [71,70] 70)

syntax (*xsymbols*)
  $\_\,OclIntersection\_math$:: $[('\tau,'\beta::collection)\,VAL,\ ('\tau,'\beta)\,VAL] => ('\tau,'\beta)\,VAL$
  (_ ∩ _   [71,70] 70)

The Syntax of Iterators

This section contains the operators of OCL involving binding, which as subsumed under the category "iterators" in the standard. Since we use a shallow embedding, we heavily use higher-order abstract syntax (HOAS) here.

However, quantifiers were not defined on the basis of iterate in HOL-OCL. This is since iterate is undefined for all infinite collections, in contrast to quantifiers, which do not necessarily have computational content.

consts

$OclForAll$ $::[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => '\tau\ Boolean] => '\tau\ Boolean$

$OclSelect$ $::[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => '\tau\ Boolean]$
$=> ('\tau,'\beta::collection)\,VAL$

$OclReject$ $::[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => '\tau\ Boolean]$
$=> ('\tau,'\beta::collection)\,VAL$

$OclCollect$ $::[('\tau,'\gamma::collection)\ VAL,$
$('\tau,'\alpha::bot)\,VAL => ('\tau,'\beta::bot)\,VAL]$
$=> ('\tau,'\delta::collection)\,VAL$

$OclCollectNested$ $::[('\tau,'\gamma::collection)\ VAL,$
$('\tau,'\alpha::bot)\,VAL => ('\tau,'\beta::bot)\,VAL]$
$=> ('\tau,'\delta::collection)\,VAL$

$OclIterate$ $::[('\tau,'\gamma::collection)\ VAL,$
$[('\tau,'\alpha::bot)\,VAL,('\tau,'\beta::bot)\,VAL] => ('\tau,'\beta::bot)\,VAL,$
$('\tau,'\beta::bot)\,VAL]$
$=> ('\tau,'\beta::bot)\,VAL$

$OclIsUnique ::[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => ('\tau,'\gamma::bot)\,VAL] => '\tau\ Boolean$

$OclOne \quad ::[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => '\tau\ Boolean] => '\tau\ Boolean$

$OclAny \quad ::[('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => '\tau\ Boolean]$
$=> ('\tau,'\alpha::bot)\,VAL$

constdefs
$OclExists \ :: [('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL => '\tau\ Boolean] => '\tau\ Boolean$
$OclExists\ S\ P \equiv \neg(OclForAll\ S\ (\lambda\ x.\ \neg\ (P\ x)))$

syntax
$\_OclForAll\_std\ ::\ ('\tau,'\beta::collection)\ VAL \Rightarrow idt \Rightarrow '\tau\ Boolean$
(_ ->forall'(_ | _') [71,100,70]50)

$\_OclExists\_std\ ::\ ('\tau,'\beta::collection)\ VAL \Rightarrow idt \Rightarrow '\tau\ Boolean$
(_ ->exists'(_ | _') [71,100,70]50)

$\_OclSelect\_std\ ::\ [('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL,\ '\tau\ Boolean]$
$=> ('\tau,'\beta::collection)\,VAL$
(_ ->select'(_ | _') [71,100,70]50)

$\_OclReject\_std\ ::\ [('\tau,'\beta::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL,\ '\tau\ Boolean]$
$=> ('\tau,'\beta::collection)\,VAL$
(_ ->reject'(_ | _') [71,100,70]50)

$\_OclCollect\_std\ ::\ [('\tau,'\gamma::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL,\ ('\tau,'\beta::bot)\,VAL]$
$=> ('\tau,'\delta::collection)\,VAL$
(_ ->collect'(_ | _') [71,100,70]50)

$\_OclCollectNested\_std\ ::\ [('\tau,'\gamma::collection)\ VAL,\ ('\tau,'\alpha::bot)\,VAL,\ ('\tau,'\beta::bot)\,VAL]$
$=> ('\tau,'\delta::collection)\,VAL$
(_ ->collectNested'(_ | _') [71,100,70]50)

$\_OclIterate\_std\ ::\ [('\tau,'\beta::collection)\ VAL,\ idt,\ idt,\ '\gamma,\ '\delta]$
$=> ('\tau,'\delta)\,VAL$
(_ ->iterate'(_;_=_ | _') [71,100,70]50)

$\_OclOne\_std\ ::\ [('\tau,'\beta::collection)\ VAL,\ '\alpha,\ '\tau\ Boolean] => '\tau\ Boolean$
(_ ->one'(_ | _') [71,100,70]50)

$\_OclIsUnique\_std ::\ [('\tau,'\beta::collection)\ VAL,\ '\alpha,\ '\tau\ Boolean] => '\tau\ Boolean$
(_ ->isUnique'(_ | _') [71,100,70]50)

syntax
$\_OclForAll\_ascii\ ::\ ('\tau,'\beta::collection)\ VAL \Rightarrow idt \Rightarrow '\tau\ Boolean$

```
                        (_ ->forAll'(_ | _') [71,100,70]50)

  _OclExists_ascii  :: ('τ,'β::collection) VAL ⇒ idt ⇒ 'τ Boolean
                        (_ ->exists'(_ | _') [71,100,70]50)

  _OclSelect_ascii  :: [('τ,'β::collection) VAL, ('τ,'α::bot) VAL, 'τ Boolean]
                        => ('τ,'β::collection) VAL
                        (_ ->select'(_ | _') [71,100,70]50)

  _OclReject_ascii  :: [('τ,'β::collection) VAL, ('τ,'α::bot) VAL, 'τ Boolean]
                        => ('τ,'β::collection) VAL
                        (_ ->reject'(_ | _') [71,100,70]50)

  _OclCollect_ascii :: [('τ,'γ::collection) VAL, ('τ,'α::bot) VAL, ('τ,'β::bot) VAL]
                        => ('τ,'δ::collection) VAL
                        (_ ->collect'(_ | _') [71,100,70]50)

  _OclCollectNested_ascii :: [('τ,'γ::collection) VAL, ('τ,'α::bot) VAL, ('τ,'β::bot) VAL]
                              => ('τ,'δ::collection) VAL
                              (_ ->collectNested'(_ | _') [71,100,70]50)

  _OclIterate_ascii  :: [('τ,'β::collection) VAL, idt, idt, 'γ, 'δ]
                        => ('τ,'δ) VAL
                        (_ ->iterate'(_;_=_ | _') [71,100,70]50)

  _OclOne_ascii       :: [('τ,'β::collection) VAL, 'α, 'τ Boolean] => 'τ Boolean
                        (_ ->one'(_ | _') [71,100,70]50)

  _OclIsUnique_ascii :: [('τ,'β::collection) VAL, 'α, 'τ Boolean] => 'τ Boolean
                        (_ ->isUnique'(_ | _') [71,100,70]50)


syntax (xsymbols)
  _OclForAll_math :: idt ⇒ ('τ,'β::collection) VAL ⇒ 'τ Boolean
                        (∀ _ ∈ _ • _ [71,100,70]50)

  _OclExists_math :: idt ⇒ ('τ,'β::collection) VAL ⇒ 'τ Boolean
                        (∃ _ ∈ _ • _ [71,100,70]50)

  _OclSelect_math  :: [('τ,'β::collection) VAL, ('τ,'α::bot) VAL, 'τ Boolean]
                        => ('τ,'β::collection) VAL
                        (⦇_ : _ | _⦈ [71,100,70]50)

  _OclReject_math  :: [('τ,'β::collection) VAL, ('τ,'α::bot) VAL, 'τ Boolean]
                        => ('τ,'β::collection) VAL
                        (⦈_ : _ | _⦇ [71,100,70]50)

  _OclCollect_math :: [('τ,'γ::collection) VAL, ('τ,'α::bot) VAL, ('τ,'β::bot) VAL]
                        => ('τ,'δ::collection) VAL
```

$(\{| \_ \in \_ \mid \_ |\}$ $[71,100,70]50)$

$\_OclCollectNested\_math :: [('\tau,'\gamma::collection)\ VAL,\ ('\tau,'\alpha::bot)\ VAL,\ ('\tau,'\beta::bot)\ VAL]$
$=> ('\tau,'\delta::collection)\ VAL$
$(\{|* \_ \in \_ \mid \_ *|\}$ $[71,100,70]50)$

$\_OclIterate\_math :: [('\tau,'\beta::collection)\ VAL,\ idt,\ idt,\ '\gamma,\ '\delta]$
$=> ('\tau,'\delta)\ VAL$
$(\_\ \texttt{->iterate}'(\_;\_=\_ \mid \_')\ [71,100,70]50)$

$\_OclOne\_math :: [('\tau,'\beta::collection)\ VAL,\ '\alpha,\ '\tau\ Boolean] => '\tau\ Boolean$
$(\_\ \texttt{->one}'(\_ \mid \_')\ [71,100,70]50)$

$\_OclIsUnique\_ascii:: [('\tau,'\beta::collection)\ VAL,\ '\alpha,\ '\tau\ Boolean] => '\tau\ Boolean$
$(\_\ \texttt{->isUnique}'(\_ \mid \_')\ [71,100,70]50)$

Semantics: Derived Concepts

In principle, the following concepts could be defined—more or less directly as according their specification in the standard—conservatively as constant definition, similarly to an abbreviation. However, we refrained from this possibility due to our theory-morpher technique, which requires as many as possible operator definitions in combinator format and derives the resulting properties from there. Consequently, we stated the requirements following from the standard simply as unproven lemmas and prove them later.

The following lemmas are unproven–they mirror the requirements in the standard and are intended as typechecked equivalents. These collection are proven on the basis of the concrete definitions in the concrete collection class theories.

One might wonder why the Isabelle-concept of an axiomatic class is not used here. The reason is that the underlying type concept of Isabelle (Order-Sorted Polymorphism in Isabelle) is too weak to express the connection between the type constructors to be specified and their content; in the subsequent declaration for example, there is no connection between $'\beta::collection$ and text $'\alpha::bot$ as there should be. These types in the abstract class are therefore quite rough approximations. As a consequence, axiomatic classes would easily require very general properties that can not be met by any concrete instance.

Count

The standard requires the following definition:

lemma *REQ_11_7_1_4*:
$self\ {->}count(obj)\ \equiv$

self −>iterate(elem;acc=0 | if elem ≜ obj then acc + 1 else acc endif)
oops

Unfortunately, for the case of sets, this definition assumes that sets are "duplicate free lists" which is simply not the case in general, in particular if we admit infinite sets as in HOL-OCL.

Instead, we define the properties pointwise for each collection.

Excludes

The standard defines exclusion via `->count` , which is possible, but messy with hindsight to a standard set theory:

lemma *REQ11_7_1_3*: *self −>excludes(obj)* ≡ (*self −>count(obj)* ≜ *0*) **oops**

includesAll

The definition of `->includesAll` strictly follows the standard REQ11_7_1_5:

lemma *REQ11_7_1_5*:
(*self −>includesAll(obj)*)
≡ (*obj −>forAll(elem* | (*self −>includes(elem)*))) **oops**

Unfortunately, this form of definition is *not* conservative for type technical reasons: elem is too polymorphic and contains a free type variable not occuring in the lhs of this equation. Solution: `->includesAll` must be defined individually on each concrete collection.

With the following three definitions, the situation is the same.

lemma *REQ11_7_1_6*:
(*self −>excludesAll(obj)*) ≡ (*obj −>forAll(elem* | (*self −>excludes(elem)*)))
oops

lemma *REQ11_7_1_7*: *isEmpty self* ≡ (*self −>size()* ≜ *0*) **oops**

lemma *REQ11_7_1_8*: *notEmpty X* ≡ *not*(*isEmpty*(*X*)) **oops**
end

B.4.10. OCL Sequence

theory *OCL_Sequence*
imports
$*HOLOCL_HOME/src/library/collection/OCL_Collection*
$*HOLOCL_HOME/src/library/collection/HOL_Collections_ext*
begin

Theorems needed in this theory but not belonging to it

Very often used simplifier sets

lemma *DEF_def_both*:
  $DEF\ X \equiv (\bot \neq X \wedge X \neq \bot)$
  by(*simp add*: *DEF_def neq_commute*)

Lifting

lemma *not_down_lift_drop* [*simp*]:
  $x \neq down \Longrightarrow \llcorner\ulcorner x\urcorner\lrcorner = x$
  by(*drule not_down_exists_lift*[*THEN iffD1*], *auto*)

Logic core

lemma *and_equiv_sand*:
  $[\![\ \tau \vDash \partial\ X; \tau \vDash \partial\ Y\ ]\!] \Longrightarrow\ (X \wedge\ Y)\ \tau =\ (X\ \dot{\wedge}\ Y)\ \tau$
  by(*simp add*: *OclSand_def OclAnd_def localValidDefined2sem ss_lifting*)

lemma *ocl_if_totally_undefined* [*simp*]:
  (*if* P *then* $\bot$ *else* $\bot$ *endif*) $= \bot$
  by(*rule ext*, *simp add*: *OclIf_def OclUndefined_def ss_lifting'*)

lemma *if_distrib_defined*:
  $[\![\ \tau \vDash \partial\ X;\ cp\ (P{::}('\tau,'\alpha{::}bot)\,VAL \Rightarrow ('\tau,'c{::}bot)\,VAL)\ ]\!] \Longrightarrow$
   P(*if* X *then* Y *else* Z *endif*) $\tau =$ (*if* X *then* $P\ Y$ *else* $P\ Z$ *endif*) τ
  apply(*auto simp*: *OclIf_def localValidDefined2sem ss_lifting'*)
  apply(*rule_tac P=P* **in** *cp_charn*, *simp_all*)+
  done

lemma *if_distrib_strict_alt*:
  $[\![$ *cp P*; $P\ \bot = \bot\ ]\!] \Longrightarrow P$ (*if* X *then* Y*else* Z*endif*) = (*if* X *then* ($P\ Y$)*else* ($P\ Z$)*endif*)
  by(*simp add*: *if_distrib_strict isStrict_def OclUndefined_def lift0_def*)

lemma *if_contract*:
  (*if* P *then* (*if* Q *then* X *else* Y *endif*)*else* Y *endif*) = (*if* ($P\ \dot{\wedge}\ (P \longrightarrow Q)$) *then* X *else* Y *endif*)
  by(*rule ext*, *simp add*: *OclIf_def OclSand_def OclImplies_def o_def*
      OclOr_def OclNot_def OclAnd_def ss_lifting)

lemma *if_not*: (*if* ($\neg\ X$) *then* Y *else* Z *endif*) = (*if* X *then* Z *else* Y *endif*)
  by(*rule ext*, *simp add*: *OclIf_def OclNot_def ss_lifting*)

Some cp theory to simplify the later proofwork

Adding currently missing cp-ness results to the simpset

lemmas *cp_hol_imp* [*simp,intro!*] = *cp_lift2*[*simplified lift2_def*, *of _ _ op* $\longrightarrow$]
lemmas *cp_hol_and* [*simp,intro!*] = *cp_lift2*[*simplified lift2_def*, *of _ _ op* $\wedge$]

lemmas *cp_hol_or* [*simp,intro!*] = *cp_lift2*[*simplified lift2_def, of _ _ op* ∨]
lemmas *cp_hol_not* [*simp,intro!*] = *cp_lift1*[*simplified lift1_def, of _* λ *x.* ¬ *x*]

lemma *cp_or* [*simp,intro!*]: ⟦ *cp P*; *cp P′* ⟧ ⟹ *cp* (λ *X.* (*P X*) ∨ (*P′ X*))
by(*simp add*: *OclOr_def*)

declare *cp_strongEq*[*simp,intro!*]

lemma *cp_sor* [*simp,intro!*]: ⟦ *cp P*; *cp P′* ⟧ ⟹ *cp* (λ *X.* (*P X*) $\dot{\vee}$ (*P′ X*))
by(*simp add*: *OclSor_def*)

lemma *cp_sand* [*simp,intro!*]: ⟦ *cp P*; *cp P′* ⟧ ⟹ *cp* (λ *X.* (*P X*) $\dot{\wedge}$ (*P′ X*))
by(*simp add*: *OclSor_def*)

lemma *cp_sxor* [*simp,intro!*]: ⟦ *cp P*; *cp P′* ⟧ ⟹ *cp* (λ *X.* (*P X*) $\dot{\oplus}$ (*P′ X*))
by(*simp add*: *OclSxor_def*)

A very useful substitution rule

lemma *cp_subst*:
⟦ *cp P* ⟧ ⟹ *P x* τ = *P* (*lift0* (*x* τ)) τ
by(*subst* (*asm*) *cp_by_cpify*, *auto*)

Setup of the environment

Special parameters

Properties of Datatype Adaption

The following rules are transformed versions of the automatically generated rules for datatype adaption

lemma *Abs_Sequence_0_inject_charn*:
⟦ (⊥::′τ::*bot*) ∉ *set x*; (⊥::′τ::*bot*) ∉ *set y* ⟧ ⟹
(((*Abs_Sequence_0* ⌞*x*⌟) = (*Abs_Sequence_0* ⌞*y*⌟)) = (*x* = *y*))
by(*subst Abs_Sequence_0_inject*,
simp_all add: *OCL_Sequence_type.Sequence_0_def smash_def*)

These are derived rules from the above one that allow the simplifier to detect false assumptions which make a goal trivially true.

lemma *Abs_Sequence_0_inject_absurd11* [*simp*]:
⟦ ⊥ ∉ *set x* ⟧ ⟹ ((*Abs_Sequence_0 down*) = (*Abs_Sequence_0* ⌞*x*⌟)) = *False*
by(*subst Abs_Sequence_0_inject*,
simp_all add: *OCL_Sequence_type.Sequence_0_def smash_def*)

lemma *Abs_Sequence_0_inject_absurd12* [*simp*]:

$[\![\perp \notin \mathit{set}\ x]\!] \Longrightarrow ((\mathit{Abs\_Sequence\_0}\ \llcorner x \lrcorner) = (\mathit{Abs\_Sequence\_0}\ \mathit{down})) = \mathit{False}$
by(*subst Abs_Sequence_0_inject,*
simp_all add: OCL_Sequence_type.Sequence_0_def smash_def)

lemma *Abs_Sequence_0_inject_absurd21* [*simp*]:
$[\![\perp \notin \mathit{set}\ x]\!] \Longrightarrow (\perp = (\mathit{Abs\_Sequence\_0}\ \llcorner x \lrcorner)) = \mathit{False}$
by(*simp add: UU_Sequence_def*)

lemma *Abs_Sequence_0_inject_absurd22* [*simp*]:
$[\![\perp \notin \mathit{set}\ x]\!] \Longrightarrow ((\mathit{Abs\_Sequence\_0}\ \llcorner x \lrcorner) = \perp) = \mathit{False}$
by(*simp add: UU_Sequence_def*)

lemma *Abs_Sequence_0_inverse_charn1* [*simp*]:
$(\perp \notin \mathit{set}\ x) \Longrightarrow \mathit{Rep\_Sequence\_0}\ (\mathit{Abs\_Sequence\_0}\ \llcorner x \lrcorner) = \llcorner x \lrcorner$
by (*subst Abs_Sequence_0_inverse,*
simp_all add: OCL_Sequence_type.Sequence_0_def smash_def)

lemma *Abs_Sequence_0_inverse_charn2* [*simp*]:
$(\perp \notin \mathit{set}\ x) \Longrightarrow \ulcorner \mathit{Rep\_Sequence\_0}\ (\mathit{Abs\_Sequence\_0}\ \llcorner x \lrcorner) \urcorner = x$
by (*subst Abs_Sequence_0_inverse,*
simp_all add: OCL_Sequence_type.Sequence_0_def smash_def)

lemma *Abs_Sequence_0_cases_charn*:
assumes *bottomCase* : $[\![x = \perp]\!] \Longrightarrow P$
assumes *listCase* : $\bigwedge y.\ ([\![x = \mathit{Abs\_Sequence\_0}\ \llcorner y \lrcorner;\ \perp \notin \mathit{set}\ y]\!] \Longrightarrow P)$
shows P
apply(*rule_tac x=x* **in** *Abs_Sequence_0_cases*)
apply(*case_tac* $y = \perp$)
apply(*rule bottomCase, simp add: UU_Sequence_def*)
apply(*frule not_down_exists_lift2*[*THEN iffD1*])
apply(*rule_tac* $y{=}\ulcorner y \urcorner$ **in** *listCase*)
apply(*auto simp: OCL_Sequence_type.Sequence_0_def smash_def*)
done

lemma *Abs_Sequence_0_induct_charn*:
assumes *bottomCase* : $P\ \perp$
assumes *stepCase* : $\bigwedge y.\ ([\![\perp \notin \mathit{set}\ y]\!] \Longrightarrow P\ (\mathit{Abs\_Sequence\_0}\ \llcorner y \lrcorner))$
shows $P\ x$
apply(*rule_tac x=x* **in** *Abs_Sequence_0_induct*)
apply(*rule_tac x=Abs_Sequence_0 y* **in** *Abs_Sequence_0_cases_charn*)
apply(*auto intro!: bottomCase elim: stepCase*)
done

These rules stem from the set theory, that was the first collection theory. It should actually be decided which of them are really needed anymore because the above rules cover most cases.

lemma *inj_on_Abs_Sequence*: *inj_on Abs_Sequence_0 Sequence_0*
by(*rule inj_on_inverseI,rule Sequence_0.Abs_Sequence_0_inverse,assumption*)

lemma *inj_Rep_Sequence* : *inj Rep_Sequence_0*
  by(*rule inj_on_inverseI*, *rule Sequence_0.Rep_Sequence_0_inverse*)

lemma *smashed_sequence_charn*:
  $(\bot \notin set\ X) = ((\llcorner X \lrcorner) \in Sequence\_0)$
  by(*unfold smash_def Sequence_0_def UU_Sequence_def,auto*)

lemma *UU_in_smashed_sequence* [*simp*]:
  $\bot \in Sequence\_0$
  by(*unfold smash_def Sequence_0_def UU_Sequence_def,auto*)

lemma *down_in_smashed_sequence* [*simp*]:
  $down \in Sequence\_0$
  by(*unfold smash_def Sequence_0_def UU_Sequence_def,auto*)

lemma *mt_in_smashed_sequence*[*simp*]:
  $(\llcorner[] \lrcorner) \in Sequence\_0$
  by (*unfold smash_def Sequence_0_def,auto*)

lemma *DEF_Abs_Sequence*: $\bigwedge X.\ (\bot \notin set\ X) \Longrightarrow DEF\ (Abs\_Sequence\_0\ (\llcorner X \lrcorner))$
  apply(*unfold DEF_def UU_Sequence_def, simp*)
  done

lemma *DEF_Rep_Sequence*:
  $\bigwedge X.\ DEF\ X \Longrightarrow DEF\ (Rep\_Sequence\_0\ X)$
  apply(*unfold DEF_def UU_Sequence_def,auto*)
  apply(*drule_tac f = Abs_Sequence_0* **in** *arg_cong*)
  apply(*simp add*: *Rep_Sequence_0_inverse*)
  done

lemma *not_DEF_Rep_Sequence*:
  $\bigwedge X.\ \neg\ DEF\ X \Longrightarrow \neg\ DEF\ (Rep\_Sequence\_0\ X)$
  apply(*unfold DEF_def UU_Sequence_def,auto*)
  apply(*simp add*: *Abs_Sequence_0_inverse*)
  done

lemma *exists_lift_Sequence*:
  $\bigwedge X.\ DEF\ X \Longrightarrow \exists\, c.\ Rep\_Sequence\_0\ X = (\llcorner c \lrcorner)$
  by (*drule DEF_Rep_Sequence,simp add*: *DEF_X_up*)

lemma *exists_lift_Sequence2*:
  $\bigwedge X.\ DEF\ X \Longrightarrow \exists\, c.\ \bot \notin set\ c \wedge Rep\_Sequence\_0\ X = (\llcorner c \lrcorner)$
  apply(*frule exists_lift_Sequence*)
  apply(*auto*)
  apply(*drule sym*, *rule swap*)
  prefer *2* **apply** *assumption***back**
  apply(*subst smashed_sequence_charn*)
  apply(*simp add*: *Rep_Sequence_0*)

done

lemma *Rep_Sequence_cases*:
$Rep\_Sequence\_0\ X = \bot \lor (\exists\, c.\ \bot \notin set\ c \land Rep\_Sequence\_0\ X = (\lfloor c \rfloor))$
apply(*case_tac DEF X*)
apply(*drule exists_lift_Sequence2*)
apply(*drule_tac [2] not_DEF_Rep_Sequence*)
apply(*simp_all add: DEF_def*)
done

These two *DEF_X_Sequence* lemmas are a central part in almost every proof that unfolds the OCL definitions. Furthermore they will also be used by the thy_morpher to automatically lift theorems about HOL lists to theorems about OCL sequences.

lemma *DEF_X_Sequence_0* : $DEF(X) = (\exists\, c.\ \bot \notin set\ c \land Rep\_Sequence\_0\ X = (\lfloor c \rfloor))$
apply(*insert Rep_Sequence_cases [of X], auto*)
apply(*drule exists_lift_Sequence,auto*)
apply(*rule swap*)
prefer *2*
apply(*rule not_DEF_Rep_Sequence, auto*)
done

lemma *DEF_X_Sequence*: $DEF\ X = (\exists\, c.\ \bot \notin set\ c \land X = Abs\_Sequence\_0\ (\lfloor c \rfloor))$
apply (*auto simp: DEF_Abs_Sequence*)
apply (*simp add:DEF_X_Sequence_0*)
apply (*erule exE, rule exI, auto*)
apply (*rule injD [OF inj_Rep_Sequence]*)
apply (*auto simp: smashed_sequence_charn Abs_Sequence_0_inverse*)
done

Because in most cases both versions of the above theorems are used they are packed conveniently in this lemma.

lemma *DEF_X_Sequence'*:
$[\![\ X \neq \bot\ ]\!] \implies$
$(\exists\, c.\ (\bot \notin set\ c) \land (Rep\_Sequence\_0\ X = \lfloor c \rfloor)) \land$
$(\exists\, c.\ (\bot \notin set\ c) \land (X = (Abs\_Sequence\_0\ \lfloor c \rfloor)))$
apply(*fold DEF_def*)
apply(*frule DEF_X_Sequence[THEN iffD1]*)
apply(*drule DEF_X_Sequence_0[THEN iffD1]*)
apply(*simp*)
done

And to enable a fast case splitting over (un)definedness of a specific sequence together with its semantic representation these two theorems are very useful. Note that there exists a *_0* version which does not require cp-ness. This *_0* version exists actually for almost all more complex theorems because not in all cases one has or can show cp-ness.

lemma *Sequence_sem_cases_0*:

```
  assumes defC: ⋀ c d. ⟦ X ≠ ⊥; ⊥ ≠ X;
                         (⊥ ∉ set c); (⊥ ∉ set d);
                         Rep_Sequence_0 X = ⌊c⌋;
                         X = Abs_Sequence_0 ⌊d⌋ ⟧ ⟹ P X
  and   undefC: ⟦ X = ⊥ ⟧ ⟹ P X
  shows P X
  apply(rule Abs_Sequence_0_cases_charn, erule undefC)
  apply(rule defC, auto)
  done

lemma Sequence_sem_cases:
  assumes defC: ⋀ c d. ⟦ (X τ) ≠ ⊥; ⊥ ≠ (X τ);
                         (⊥ ∉ set c); (⊥ ∉ set d);
                         Rep_Sequence_0 (X τ) = ⌊c⌋;
                         (X τ) = Abs_Sequence_0 ⌊d⌋ ⟧ ⟹ P X τ
  and   undefC: ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ
  and      cpP: cp P
  shows P X τ
  apply(rule_tac P2=P in subst[OF sym[OF cp_subst]], rule cpP)
  apply(rule Sequence_sem_cases_0)
  apply(rule_tac P1=P in subst[OF cp_subst], rule cpP)
  apply(rule defC) prefer 7
  apply(rule_tac P1=P in subst[OF cp_subst], rule cpP)
  apply(rule undefC, simp_all)
  done
```

And a version for the case that we're reasoning over equalities of functions.

```
lemma Sequence_sem_cases_ext:
  assumes defC: ⋀ c d τ. ⟦ (X τ) ≠ ⊥; ⊥ ≠ (X τ);
                         (⊥ ∉ set c); (⊥ ∉ set d);
                         Rep_Sequence_0 (X τ) = ⌊c⌋;
                         (X τ) = Abs_Sequence_0 ⌊d⌋ ⟧ ⟹ P X τ = Q X τ
  and   undefC: ⋀ τ. ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ = Q X τ
  and      cpP: cp P
  and      cpQ: cp Q
  shows P X = Q X
  apply(rule ext)
  apply(rule_tac X=X in Sequence_sem_cases)
  apply(rule defC) prefer 7
  apply(rule undefC)
  apply(simp_all add: cpP cpQ)
  done
```

These four lemmas are used by ocl_setup_op to reason about definedness:

```
lemma lift2_strict_is_isdef_fw_Sequence_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0 ⌊g ⌈Rep_Sequence_0
x⌉ y⌋)))
  and     inv_g: !!a b.( ⟦ ⊥ ∉ set a; (⊥::'τ::bot) ≠ b ⟧ ⟹ ⊥ ∉ set (g a b))
  shows          ∂(f X Y) = (∂ X ∧ ∂ Y)
```

```
  apply(rule ext)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                  o_def ss_lifting)
  apply(rule_tac X=X x in Sequence_sem_cases_0)
  apply(rule impI, drule_tac b=(Y x) in  inv_g)
  apply(simp_all add: neq_commute)
  done

lemma lift2_strict_is_isdef_fw_Sequence_Sequence:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0 ⌊g ⌈Rep_Sequence_0
x⌉ ⌈Rep_Sequence_0 y⌉⌋)))
  and     inv_g: !!a b.( ⟦ ⊥ ∉ set a; ⊥ ∉ set b ⟧ ⟹ ⊥ ∉ set (g a b))
  shows          ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(rule ext)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                  o_def ss_lifting)
  apply(rule_tac X=X x in Sequence_sem_cases_0)
  apply(rule_tac X=Y x in Sequence_sem_cases_0)
  apply(rule impI, drule_tac b=ca in  inv_g)
  apply(simp_all add: neq_commute)
  done

lemma lift2_strictify_implies_LocalValid_defined_Sequence_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0 ⌊g ⌈Rep_Sequence_0
x⌉ y⌋)))
  and     inv_g: !!a b.( ⟦ ⊥ ∉ set a; (⊥::'τ::bot) ≠ b⟧ ⟹ ⊥ ∉ set (g a b))
  shows          (τ ⊨ ∂(f X Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
  apply(insert f_def)
  apply(drule_tac X=X and  Y=Y in lift2_strict_is_isdef_fw_Sequence_Val)
  apply(rule inv_g, assumption+)
  apply(simp add: OclAnd_def  OclTrue_def o_def
              OclIsDefined_def lift0_def lift1_def lift2_def
              OclLocalValid_def)
  done

lemma lift2_strictify_implies_LocalValid_defined_Sequence_Sequence:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0 ⌊g ⌈Rep_Sequence_0
x⌉ ⌈Rep_Sequence_0 y⌉⌋)))
  and     inv_g: !!a b.( ⟦ ⊥ ∉ set a; ⊥ ∉ set b ⟧ ⟹ ⊥ ∉ set (g a b))
  shows          (τ ⊨ ∂(f X Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
  apply(insert f_def)
  apply(drule_tac X=X and  Y=Y in lift2_strict_is_isdef_fw_Sequence_Sequence)
  apply(rule inv_g, assumption+)
  apply(simp add: OclAnd_def  OclTrue_def o_def
              OclIsDefined_def lift0_def lift1_def lift2_def
              OclLocalValid_def)
  done
```

Building a canonic representation of sequences

The empty sequence

constdefs
$OclMtSequence$ :: $('\tau, '\alpha{::}bot\ Sequence\_0)\ VAL$
$OclMtSequence \equiv lift0(Abs\_Sequence\_0(\llcorner[]\lrcorner))$

syntax
$\_OclMtSequence\_std$ :: $('\tau,'\alpha{::}bot)\ VAL \Rightarrow '\tau\ Boolean$ ([])
syntax
$\_OclMtSequence\_ascii$ :: $('\tau,'\alpha{::}bot)\ VAL \Rightarrow '\tau\ Boolean$ ($mtSequence$)
syntax ($xsymbols$)
$\_OclMtSequence\_math$ :: $('\tau,'\alpha{::}bot)\ VAL \Rightarrow '\tau\ Boolean$ ([])

lemma $OCL\_is\_isdef\_OclMtSequence$ [$simp$]:
$\models \partial\ []$
by($simp\ add$: $OclValid\_def\ OclIsDefined\_def\ OclTrue\_def$
$OclMtSequence\_def\ ss\_lifting'$)

lemma $OCL\_is\_defopt\_OclMtSequence$ [$simp$]:
$\tau \models \partial\ []$
by($simp\ add$: $valid\_elim$)

And the 'cons' operation: including

defs

$OclIncluding\_def$:
$OclIncluding \equiv lift2(strictify(\lambda S.\ strictify\ (\lambda e.\ Abs\_Sequence\_0$
$(\llcorner concat\ [\ulcorner Rep\_Sequence\_0\ S\urcorner,\ [e]\ ]\lrcorner))))$

ocl_setup_op [$OclIncluding$]

lemma $OCL\_is\_def\_OclIncluding$:
$\partial(OclIncluding\ (X{::}('\tau,'\alpha{::}bot)Sequence)\ (Y{::}('\tau,'\alpha{::}bot)\ VAL)) = (\partial\ X \wedge \partial\ Y)$
by($rule\ lift2\_strict\_is\_isdef\_fw\_Sequence\_Val[OF\ OclIncluding\_def],\ simp$)

lemma $OCL\_is\_defopt\_OclIncluding$:
$(\tau \models \partial(OclIncluding\ (X{::}('\tau,'\alpha{::}bot)Sequence)\ (Y{::}('\tau,'\alpha{::}bot)\ VAL))) = ((\tau \models \partial\ X) \wedge (\tau \models \partial\ Y))$
by($rule\ lift2\_strictify\_implies\_LocalValid\_defined\_Sequence\_Val[OF\ OclIncluding\_def],$
$simp$)

The syntax translation $mkSequence$ builds now our sequences

syntax
$@OclFinSequence$ :: $args \Rightarrow ('\tau,'\alpha\ Sequence\_0)\ VAL$ ($mkSequence[(\_)]$)

translations
$mkSequence[x,\ xs]$ == $OclIncluding\ (mkSequence[xs])\ x$
$mkSequence[x]$ == $OclIncluding\ OclMtSequence\ x$

lemma *test* : *mkSequence*[*1*,*2*,*3*,*4*,*5*] = *?X* **oops**

Relation of mtSequence and including

These two theorems should actually suffice to decide inequality of sequences

lemma *including_notstrongeq_mtSequence*[*simp*]:
(¬ ((*X*::('τ,'α::*bot*)*Sequence*) ->including (*a*::('τ,'α) *VAL*) ≜ [])) = T
apply(*rule_tac X*=*X* **in** *Sequence_sem_cases_ext*)
apply(*simp_all add*: *cp_strongEq*)
apply(*simp_all add*: *OclIncluding_def OclMtSequence_def OclStrongEq_def*
OclTrue_def OclNot_def localValid2sem ss_lifting'
Abs_Sequence_0_inject_charn neq_commute)
done

lemma *including_notstricteq_mtSequence*[*simp*]:
⟦ τ ⊨ ∂ (*a*::('τ,'α) *VAL*); τ ⊨ ∂ (*X*::('τ,'α::*bot*)*Sequence*) ⟧ ⟹
τ ⊨ (*X* ->including *a*) ‘<>‘ []
apply(*rule_tac X*=*X* **in** *Sequence_sem_cases*)
apply(*simp_all add*: *cp_strongEq cp_strictEq localValidDefined2sem*)
apply(*simp_all add*: *OclIncluding_def OclMtSequence_def OclStrictEq_def*
OclTrue_def OclNot_def localValid2sem ss_lifting'
Abs_Sequence_0_inject_charn neq_commute)
done

Case exhaustion over the canonic representation

lemma *Sequence_cases_including_0*:
assumes *undefCase*: ⟦ (*X*::'τ::*bot Sequence_0*) = ⊥ ⟧ ⟹ *P*
and *mtCase*: ⟦ *X* = ([] τ) ⟧ ⟹ *P*
and *stepCase*: ⋀ (*Y*::'τ::*bot Sequence_0*) (*a*::'τ::*bot*).
⟦ *DEF Y*; *DEF a*; *X* = ((*lift0 Y*) ->including (*lift0 a*)) τ⟧ ⟹ *P*
shows *P*
apply(*rule Abs_Sequence_0_cases_charn*, *erule undefCase*)
apply(*rule_tac xs*=*y* **in** *rev_cases*)
apply(*rule mtCase*)
apply(*simp add*: *OclMtSequence_def ss_lifting'*)
apply(*rule_tac Y*=*Abs_Sequence_0* ⌞*ys*⌟ **and** *a*=*ya* **in** *stepCase*)
apply(*simp_all add*: *OclIncluding_def ss_lifting' neq_commute*)
done

lemma *Sequence_cases_including*:
assumes *undefCase* : ⟦ τ ⊨ (*X*::('τ,'α::*bot*)*Sequence*) ≜ ⊥ ⟧ ⟹ *P* τ
and *mtCase*: ⟦ τ ⊨ *X* ≜ [] ⟧ ⟹ *P* τ
and *stepCase*: ⋀ (*Y*::('τ,'α::*bot*)*Sequence*) (*a*::('τ,'α::*bot*) *VAL*).
⟦ τ ⊨ *X* ≜ (*Y* ->including *a*); τ ⊨ ∂ *Y*; τ ⊨ ∂ *a*⟧ ⟹ *P* τ

```
  shows             P τ
  apply(rule_tac X=X τ and τ=τ in Sequence_cases_including_0)
  apply(rule undefCase) prefer 2
  apply(rule mtCase) prefer 3
  apply(rule stepCase, simp_all add: localValidDefined2sem)
  apply(simp_all add: localValid2sem OclStrongEq_def
                 OclUndefined_def ss_lifting')
  done
```

Induction over the canonic representation

To simplify reasoning there exists a _0 version which doesn't pose any restrictions on the involved functions with regard to context-passingness.

```
lemma Sequence_induct_including_0:
  assumes undefCase: P ⊥
  and        mtCase: P ([] τ)
  and      stepCase: ⋀ (X::'τ::bot Sequence_0) (a::'τ::bot).
                 ⟦ P X; DEF X; DEF a ⟧ ⟹ (P ((OclIncluding (lift0 X) (lift0 a)) τ))
  shows             P (X::'τ::bot Sequence_0)
  apply(rule Abs_Sequence_0_induct_charn, rule undefCase)
  apply(erule rev_mp)
  apply(rule rev_induct)
  apply(insert mtCase, simp add: OclMtSequence_def ss_lifting')
  apply(auto)
  apply(rotate_tac -1, drule_tac a=x and X=Abs_Sequence_0 ⌞xs⌟ in stepCase)
  apply(erule DEF_Abs_Sequence)
  apply(simp_all add: OclIncluding_def neq_commute ss_lifting')
  done

lemma Sequence_induct_including:
  assumes undefCase : P ⊥ τ
  and        mtCase: P [] τ
  and      stepCase: ⋀ (X::('τ,'α::bot)Sequence) (a::('τ,'α::bot) VAL).
                 ⟦ P X τ; τ ⊨ ∂ X; τ ⊨ ∂ a⟧ ⟹ P (OclIncluding X a) τ
  and           cpP: cp P
  shows             P (X::('τ,'α::bot)Sequence) τ
  apply(insert cpP)
  apply(frule cp_by_cpify[THEN iffD1])
  apply(rule subst, rule sym)
  apply(rule_tac x=X in cp_subst, simp)
  apply(rule_tac τ=τ in Sequence_induct_including_0)
  apply(insert undefCase)
  apply(auto simp: OclUndefined_def intro!: mtCase)
  apply(drule stepCase)
  apply(simp_all add: localValidDefined2sem lift0_def)
  done
```

A second sequence constructor on the OCL level

defs

$OclCollectionRange\_def$:
$OclCollectionRange \equiv lift2\ (strictify\ (\lambda\ x{::}Integer\_0.\ strictify\ (\lambda\ y.$
$\quad Abs\_Sequence\_0\ \llcorner map\ (\lambda\ z{::}nat.\ \llcorner(int\ z) + \ulcorner x\urcorner\lrcorner)$
$\quad\quad [0..<(nat\ (\ulcorner y\urcorner - \ulcorner x\urcorner + 1))]\lrcorner)))$

ocl_setup_op [$OclCollectionRange$]

lemma $OCL\_is\_defopt\_OclCollectionRange$ [$simp$]:
$\tau \models \partial\ ((OclCollectionRange\ (a{::}('\tau\ Integer))\ b){::}('\tau,\ Integer\_0)Sequence) =$
$((\tau \models \partial\ a) \wedge (\tau \models \partial\ b))$
apply($subgoal\_tac\ \bot \notin set\ (map\ (\lambda z.\ \llcorner((int\ z) + \ulcorner(a\ \tau)\urcorner)\lrcorner)\ [0..<(nat\ ((\ulcorner(b\ \tau)\urcorner - \ulcorner(a\ \tau)\urcorner) + 1))])$)
apply($auto\ simp{:}\ localValidDefined2sem\ OclCollectionRange\_def\ ss\_lifting$)
done

lemma $collectionRange\_mtSequence\_conv$:
$[\![\ \tau \models (b{::}'\tau\ Integer) < (a{::}'\tau\ Integer)\ ]\!] \Longrightarrow$
$OclCollectionRange\ a\ b\ \tau = ([]{::}('\tau,Integer\_0)Sequence)\ \tau$
apply($case\_tac\ \tau \models \not\partial\ a,\ ocl\_hypsubst,\ simp$)
apply($case\_tac\ \tau \models \not\partial\ b,\ ocl\_hypsubst,\ simp$)
apply($simp\ add{:}\ localValidDefined2sem\ DEF\_def\_both,\ clarify$)
apply($simp\_all\ add{:}\ DEF\_def\ OclCollectionRange\_def\ OclMtSequence\_def$
$\quad OclLess\_def\ localValid2sem\ ss\_lifting'$)
done

lemma $collectionRange\_singleton\_UC$[$simp$]:
$OclCollectionRange\ a\ a = []{::}('\tau,Integer\_0)Sequence$ `->including` $(a{::}('\tau,Integer\_0)\ VAL)$
apply($rule\ ext,\ case\_tac\ x \models \not\partial(a),\ ocl\_hypsubst,\ simp$)
apply($simp\ add{:}\ OclCollectionRange\_def\ OclMtSequence\_def\ OclIncluding\_def$
$\quad localValidDefined2sem\ ss\_lifting'$)
done

lemma $collectionRange\_expand\_including$:
$[\![\ \tau \models (a{::}'\tau\ Integer) \leq b\ ]\!] \Longrightarrow$
$((OclCollectionRange\ a\ b){::}('\tau,Integer\_0)Sequence)\ \tau =$
$(OclCollectionRange\ a\ (b - 1))$ `->including` $b\ \tau$
apply($subgoal\_tac\ \bot \notin set\ (map\ (\lambda z.\ \llcorner((int\ z) + \ulcorner(a\ \tau)\urcorner)\lrcorner)\ [0..<(nat\ ((\ulcorner(b\ \tau)\urcorner - \ulcorner(a\ \tau)\urcorner) + 1))])$)
apply($subgoal\_tac\ \bot \notin set\ (map\ (\lambda z.\ \llcorner((int\ z) + \ulcorner(a\ \tau)\urcorner)\lrcorner)\ [0..<(nat\ (\ulcorner(b\ \tau)\urcorner - \ulcorner(a\ \tau)\urcorner))])$)
apply($auto\ simp{:}\ OclCollectionRange\_def\ OclIncluding\_def$
$\quad OclLe\_def\ minus\_def\ One\_ocl\_int\_def\ localValid2sem\ ss\_lifting'$)
apply($subst\ Abs\_Sequence\_0\_inject\_charn,\ auto$)
apply($subgoal\_tac\ (nat\ (\ulcorner(b\ \tau)\urcorner - \ulcorner(a\ \tau)\urcorner + 1)) = (nat\ ((\ulcorner(b\ \tau)\urcorner - \ulcorner(a\ \tau)\urcorner)) + (nat\ 1))$)

apply(*simp, arith*)
done

The conversion operators

Sequence to sequence

defs

$OclAsSequence\_def$:
$OclAsSequence \quad \equiv id$

lemma *asSequence_identity*[*simp*]:
$OclAsSequence\ (S{::}('\tau,'\alpha{::}bot)Sequence) = S$
by(*simp add: OclAsSequence_def*)

Sequence to bag

defs

$OclAsBag\_def$:
$OclAsBag \quad \equiv lift1\ (strictify\ (\lambda X.\ Abs\_Bag\_0\ \llcorner multiset\_of\ \ulcorner Rep\_Sequence\_0\ X\urcorner\lrcorner))$

ocl_setup_op [*OclAsBag*]

Sequence to ordered set

defs

$OclAsOrderedSet\_def$:
$OclAsOrderedSet \quad \equiv lift1\ (strictify\ (\lambda X.\ Abs\_OrderedSet\_0\ \llcorner rev\ (remdups\ (rev\ \ulcorner Rep\_Sequence\_0\ X\urcorner))\lrcorner))$

ocl_setup_op [*OclAsOrderedSet*]

Sequence to set

defs

$OclAsSet\_def$:
$OclAsSet \quad \equiv lift1\ (strictify\ (\lambda X.\ Abs\_Set\_0\ \llcorner set\ \ulcorner Rep\_Sequence\_0\ X\urcorner\lrcorner))$

ocl_setup_op [*OclAsSet*]

OclIterate

defs

$OclIterate\_def$:
$OclIterate \equiv (lift3'\ lift\_arg0\ lift\_arg2\ lift\_arg0)\ (strictify\ (\lambda\ S\ P\ A.\ (foldl\ (\lambda\ x\ y.\ P\ y\ x)\ A\ (\ulcorner Rep\_Sequence\_0\ S\urcorner))))$

ocl_setup_op [*OclIterate*]

The characteristic behaviour of iterate on the canonical representation

lemma *iterate_of_mtSequence[simp]*:
  $(OclIterate\ ([]::('\tau,'\alpha::bot)Sequence)$
      $(P::(('\tau \Rightarrow '\alpha) \Rightarrow (('\tau \Rightarrow 'c::bot) \Rightarrow ('\tau \Rightarrow 'c))))$
      $A) = A$
  apply(*rule ext*)
  apply(*subgoal_tac* $x \models (\partial\ [])$)
  apply(*simp_all add: OclIterate_def OclIncluding_def OclMtSequence_def localValidDefined2sem ss_lifting'*)
 done

lemma *iterate_of_including_0*:
  assumes *defS*: $\tau \models \partial(S::('\tau,('\alpha::bot))Sequence)$
  and *defx*: $\tau \models \partial(x::('\tau \Rightarrow '\alpha::bot))$
  shows $((OclIterate\ (OclIncluding\ S\ x)\ P\ A)\ \tau) =$
    $(P\ (lift0\ (x\ \tau))\ (lift0\ ((OclIterate\ (S::(('\tau,'\alpha::bot)Sequence))\ P\ A)\ \tau))\ \tau)$
  apply(*insert defx defS*)
  apply(*frule_tac Y1=x* **in** *conjI[THEN OCL_is_defopt_OclIncluding[THEN iffD2]], assumption*)
  apply(*simp add: localValidDefined2sem DEF_def*)
  apply(*frule DEF_X_Sequence', clarify*)
  apply(*simp add: OclIterate_def OclIncluding_def ss_lifting'*)
  apply(*frule DEF_X_Sequence', clarify, simp*)
  apply(*subst (asm) Abs_Sequence_0_inject_charn, auto*)
  done

lemma *iterate_of_including*:
  assumes *defS*: $\tau \models \partial(S::('\tau,('\alpha::bot))Sequence)$
  and *defx*: $\tau \models \partial(x::('\tau \Rightarrow '\alpha::bot))$
  and *cpP_1*: $\bigwedge\ y.\ cp\ (\lambda\ x.\ P\ x\ y)$
  and *cpP_2*: $\bigwedge\ x.\ cp\ (P\ x)$
  shows $((OclIterate\ (OclIncluding\ S\ x)\ P\ A)\ \tau) =$
    $(P\ x\ (OclIterate\ (S::(('\tau,'\alpha::bot)Sequence))\ P\ A)\ \tau)$
  apply(*insert defS defx cpP_1 cpP_2*)
  apply(*drule_tac x=x* **and** *P=P* **and** *A=A* **in** *iterate_of_including_0*)
  apply(*simp_all add: cp_by_cpify*)
  done

lemma *iterate_opt_of_including*:
  assumes *P_undef_1*: $!!y.\ \ P\ \bot\ y = \bot$
  and *P_undef_2*: $!!x.\ \ P\ x\ \bot = \bot$
  and *cpP_1*: $\bigwedge\ y.\ cp\ (\lambda\ x.\ P\ x\ y)$
  and *cpP_2*: $\bigwedge\ x.\ cp\ ((P::('\tau,'\alpha::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL)\ x)$
  shows $(OclIterate\ (OclIncluding\ S\ x)\ P\ A) =$
    $(P\ x\ (OclIterate\ (S::(('\tau,'\alpha::bot)Sequence))\ P\ A))$

```
  apply(insert P_undef_1 P_undef_2 cpP_1 cpP_2)
  apply(rule ext)
  apply(case_tac xa ⊨ ∂ S)
  apply(case_tac xa ⊨ ∂ x)
  apply(rule iterate_of_including, simp_all)
  apply(frule isUndefined_charn_local[THEN iffD2])
  apply(rule trans, erule_tac A=x in cp_charn, simp, simp)
  apply(rule sym[OF trans], erule_tac A=x in cp_charn, simp, simp)
  apply(frule isUndefined_charn_local[THEN iffD2])
  apply(rule trans, erule_tac A=S in cp_charn, simp, simp)
  apply(rule sym[OF trans], erule_tac A=S in cp_charn)
  apply(rule_tac P=P x in cp_compose2, simp_all)
  done
```

The universal and fusion property

Fusion and universal theorems, as described in [25].

```
lemma iterate_universal_0:
  assumes g_cp: cp g
  and g_undef:  g (⊥::('τ,'α::bot)Sequence) τ = (⊥::('τ,'c::bot) VAL) τ
  and g_mtSeq:  g (OclMtSequence::('τ,'α::bot)Sequence) τ = A τ
  and g_incl:   ⋀ S x. ⟦ τ ⊨ ∂ S; τ ⊨ ∂ x ⟧ ⟹
                      g (OclIncluding S x) τ = P (lift0 (x τ)) (lift0 (g S τ)) τ
  shows g S τ = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A τ
  apply(rule_tac X=S in Sequence_induct_including)
  apply(simp_all add: g_cp g_undef g_mtSeq)
  apply(subst iterate_of_including_0, simp_all add: g_incl)
  done

lemma iterate_universal:
  assumes g_cp: cp g
  and     cpP_1: ⋀ y. cp (λ x. P x y)
  and     cpP_2: ⋀ x. cp (P x)
  and  g_undef: g ⊥ τ = (⊥::('τ,'c::bot) VAL) τ
  and  g_mtSeq: g (OclMtSequence::('τ,'α::bot)Sequence) τ = A τ
  and   g_incl: ⋀ S x. ⟦ τ ⊨ ∂ S; τ ⊨ ∂ x ⟧ ⟹
                      g (OclIncluding S x) τ = P x (g S) τ
  shows g S τ = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A τ
  by(insert cpP_1 cpP_2 g_cp, rule iterate_universal_0,
     simp_all add: g_undef g_mtSeq g_incl cp_by_cpify)


lemma iterate_universal_0_opt:
  assumes g_cp: cp g
  and  g_mtSeq: g (OclMtSequence::('τ,'α::bot)Sequence) τ = A τ
  and   g_incl: ⋀ S x. ⟦ τ ⊨ ∂ S; τ ⊨ ∂ x ⟧ ⟹
                      g (OclIncluding S x) τ = P (lift0 (x τ)) (lift0 (g S τ)) τ
  and     defS: τ ⊨ ∂ S
```

```
  shows g S τ = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A τ
  apply(insert defS, erule rev_mp)
  apply(rule_tac X=S in Sequence_induct_including)
  apply(simp_all add: g_cp g_mtSeq)
  apply(subst iterate_of_including_0, simp_all add: g_incl)
  done

lemma iterate_universal_opt:
  assumes g_cp: cp g
  and     cpP_1: ⋀ y. cp (λ x. P x y)
  and     cpP_2: ⋀ x. cp (P x)
  and   g_mtSeq: g (OclMtSequence::('τ,'α::bot)Sequence) τ = A τ
  and    g_incl: ⋀ S x. ⟦ τ ⊨ ∂ S; τ ⊨ ∂ x ⟧ ⟹
                       g (OclIncluding S x) τ = P x (g S) τ
  and      defS: τ ⊨ ∂ S
  shows g S τ = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A τ
  by(insert cpP_1 cpP_2 g_cp, rule iterate_universal_0_opt,
     simp_all add: g_mtSeq g_incl cp_by_cpify)



lemma iterate_fusion_0_opt:
  ⟦ cp h;
    DEF (S τ);
    h B τ = A τ;
    !!x y. h (Q (lift0 (x τ)) (lift0 (y τ))) τ =
           P (lift0 (x τ)) (lift0 (h y τ)) τ
   ⟧ ⟹
     h (OclIterate (S::('τ,'α::bot)Sequence)
                   (Q::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) B) τ
   = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'d::bot) VAL ⇒ ('τ,'d::bot) VAL) A τ
   apply(rule iterate_universal_0_opt)
   apply(rule_tac P=h in cp_compose2)
   apply(simp_all)
   apply(rule trans, rule_tac P=h in cp_subst, simp)
   apply(subst iterate_of_including_0)
   apply(simp_all add: cp_by_cpify localValidDefined2sem)
   done

lemma iterate_fusion_opt:
  ⟦ cp h;
    ⋀ x. cp (P x);
    τ ⊨ ∂ S;
    h B τ = A τ;
    ⋀ x y. h (Q x y) τ = P x (h y) τ
   ⟧ ⟹
     h (OclIterate (S::('τ,'α::bot)Sequence)
                   (Q::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) B) τ
   = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'d::bot) VAL ⇒ ('τ,'d::bot) VAL) A τ
```

```
apply(rule iterate_fusion_0_opt)
apply(assumption)+
apply(simp add: localValidDefined2sem)
apply(assumption)
apply(subgoal_tac h y τ = h (lift0 (y τ)) τ)
apply(simp)
apply(rule trans)
apply(rule_tac P=λ x. P ?X x in cp_subst)
apply(simp_all add: lift0_def cp_by_cpify)
done
```

```
lemma iterate_fusion_0:
 ⟦ cp h;
   h (⊥::('τ,'c::bot) VAL) τ = (⊥::('τ,'d::bot) VAL) τ;
   h B τ = A τ;
   !!x y. h (Q (lift0 (x τ)) (lift0 (y τ))) τ =
          P (lift0 (x τ)) (lift0 (h y τ)) τ
 ⟧ ⟹
   h (OclIterate (S::('τ,'α::bot)Sequence)
                 (Q::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) B) τ
 = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'d::bot) VAL ⇒ ('τ,'d::bot) VAL) A τ
  apply(rule iterate_universal_0)
  apply(rule_tac P=h in cp_compose2)
  apply(simp_all)
  apply(rule trans, rule_tac P=h in cp_subst, simp)
  apply(subst iterate_of_including_0)
  apply(simp_all add: cp_by_cpify)
  done
```

```
lemma iterate_fusion:
 ⟦ cp h;
   ⋀ x. cp (P x);
   h (⊥::('τ,'c::bot) VAL) τ = (⊥::('τ,'d::bot) VAL) τ;
   h B τ = A τ;
   ⋀ x y. h (Q x y) τ = P x (h y) τ
 ⟧ ⟹
   h (OclIterate (S::('τ,'α::bot)Sequence)
                 (Q::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) B) τ
 = OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'d::bot) VAL ⇒ ('τ,'d::bot) VAL) A τ
 apply(rule iterate_fusion_0)
 apply(assumption)+
 apply(subgoal_tac h y τ = h (lift0 (y τ)) τ)
 apply(simp)
 apply(rule trans)
 apply(rule_tac P=λ x. P ?X x in cp_subst)
 apply(simp_all add: lift0_def cp_by_cpify)
 done
```

Further propierties of iterate

Undefinedness the second part

lemma *OCL_undef_2_OclIterate* [*simp*]:
  assumes *P_undef_2*: $\bigwedge x.\ \ P\ x\ \bot = \bot$
  shows $OclIterate\ (S{::}('\tau,'\alpha{::}bot) Sequence)$
  $(P{::}('\tau,'\alpha{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL)$
  $\bot$
  $= (\bot{::}('\tau,'c{::}bot)\ VAL)$
  apply(*insert P_undef_2, rule ext, rule sym*)
  apply(*rule iterate_universal_0*)
  apply(*simp_all add: lift0_def OclUndefined_def*)
  done

lemma *OCL_is_defopt_OclIterate*:
  assumes *defopt_P*: $!!x\ y.\ \ (\tau \vDash \partial\ (P\ x\ y))$
  $= ((\tau \vDash \partial\ x) \wedge (\tau \vDash \partial\ y))$
  and *cpP_1*: $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
  and *cpP_2*: $\bigwedge x.\ cp\ ((P{::}('\tau,'\alpha{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL)\ x)$
  shows $\tau \vDash \partial(OclIterate\ (S{::}('\tau,'\alpha{::}bot) Sequence)\ \ P\ A) = ((\tau \vDash (\partial\ S)) \wedge (\tau \vDash (\partial\ A)))$
  apply(*insert defopt_P cpP_1 cpP_2*)
  apply(*rule_tac X=S* **and** *τ=τ* **in** *Sequence_induct_including, simp_all*)
  apply(*rule trans*)
  apply(*rule_tac x=(X* `->including` *a* `->iterate`(*u;ua=A* | (*P u ua*))) **in** *cp_subst, simp*)
  apply(*subst iterate_of_including*)
  apply(*simp_all*)
  apply(*subst OCL_is_defopt_OclIncluding*)
  apply(*rule trans, rule sym*)
  apply(*rule_tac x=(P a (OclIterate X P A))* **in** *cp_subst, simp_all*)
  done

Forward rules to transfer properties about iterate to functions definable by iterate

For *unary* functions

lemma *f1_by_iterate_cp*:
  assumes *f_by_it*: $\bigwedge S.\ (f\ (S{::}('\tau,'\alpha{::}bot) Sequence)) =$
  $(OclIterate\ S\ (P{::}('\tau,'\alpha{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL)\ \ A)$
  and *cpF*: *cp F*
  and *cpF'*: *cp F'*
  shows $cp\ (\lambda X.\ (f\ ((F\ X){::}(('\tau,'\alpha{::}bot) Sequence))))$
  by(*simp add: f_by_it cpF*)

lemma *f1_by_iterate_undef_1*:
  $[\![\ \bigwedge S.\ f\ (S{::}('\tau,'\alpha{::}bot) Sequence) =$
  $OclIterate\ S\ (P{::}('\tau,'\alpha{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL \Rightarrow ('\tau,'c{::}bot)\ VAL)\ \ A\ ]\!]$
  $\Longrightarrow$

$f \perp = \perp$
by(*simp*)

lemma *f1_by_iterate_defopt*:
assumes *f_by_it*: $\bigwedge S.\ (f\ (S::('\tau,'\alpha::bot)Sequence)) =$
$(OclIterate\ S\ (P::('\tau,'\alpha::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL)\ A)$
and *defopt_P*: $!!x\ y.\quad (\tau \vDash \partial\ (P\ x\ y))$
$= ((\tau \vDash \partial\ x) \wedge (\tau \vDash \partial\ y))$
and *cpP_1*: $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
and *cpP_2*: $\bigwedge x.\ cp\ ((P::('\tau,'\alpha::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL)\ x)$
shows $\tau \vDash \partial(f\ (S::('\tau,'\alpha::bot)Sequence)) = ((\tau \vDash (\partial\ S)) \wedge (\tau \vDash (\partial\ A)))$
by(*simp add*: *f_by_it OCL_is_defopt_OclIterate defopt_P cpP_1 cpP_2*)

lemma *f1_by_iterate_mtSequence*:
$[\![\ \bigwedge S.\ f\ (S::('\tau,'\alpha::bot)Sequence) =$
$OclIterate\ S\ (P::('\tau,'\alpha::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL)\ A\ ]\!]$
$\Longrightarrow$
$f\ [] = A$
by(*simp*)

lemma *f1_by_iterate_including*:
assumes *f_by_it*: $\bigwedge S.\ (f\ (S::('\tau,'\alpha::bot)Sequence)) =$
$(OclIterate\ S\ (P::('\tau,'\alpha::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL)\ A)$
and *defS*: $\tau \vDash \partial(S::('\tau,('\alpha::bot))Sequence)$
and *defx*: $\tau \vDash \partial(x::('\tau \Rightarrow\ '\alpha::bot))$
and *cpP_1*: $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
and *cpP_2*: $\bigwedge x.\ cp\ (P\ x)$
shows $(f\ (OclIncluding\ S\ x)\ \tau) = (P\ x\ (f\ S)\ \tau)$
by(*simp add*: *f_by_it iterate_of_including defS defx cpP_1 cpP_2*)

lemma *f1_by_iterate_opt_including*:
assumes *f_by_it*: $\bigwedge S.\ (f\ (S::('\tau,'\alpha::bot)Sequence)) =$
$(OclIterate\ S\ (P::('\tau,'\alpha::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL \Rightarrow ('\tau,'c::bot)\,VAL)\ A)$
and *P_undef_1*: $!!y.\ \ P \perp y = \perp$
and *P_undef_2*: $!!x.\ \ P\ x \perp = \perp$
and *cpP_1*: $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
and *cpP_2*: $\bigwedge x.\ cp\ (P\ x)$
shows $f\ (OclIncluding\ S\ x) = P\ x\ (f\ S)$
by(*simp add*: *f_by_it iterate_opt_of_including*
P_undef_1 P_undef_2 cpP_1 cpP_2)

For *binary* fuctions

This cp-ness result holds only for binary functions whose second argument is the neutral element of iterate. Currently the only function of this style is union.

lemma *f2_by_iterate_cp*:
  assumes *f_by_it*: $\bigwedge S\ A.\ (f\ (S::('\tau,'\alpha::bot)Sequence)\ A) =$
  $(OclIterate\ S\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ A)$
  and *cpF*: $cp\ F$
  and *cpF'*: $cp\ F'$
  shows $cp\ (\lambda X.\ (f\ ((F\ X)::(('\tau,'\alpha::bot)Sequence))\ (F'\ A)))$
  by(*simp add*: *f_by_it cpF*)

lemma *f2_by_iterate_undef_1*:
  $[\![\ \bigwedge S\ A.\ f\ (S::('\tau,'\alpha::bot)Sequence)\ A =$
  $OclIterate\ S\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ A\ ]\!]$
  $\Longrightarrow$
  $f\ \bot\ A = \bot$
  by(*simp*)

lemma *f2_by_iterate_undef_2*:
  assumes *f_by_it*: $\bigwedge S\ A.\ (f\ (S::('\tau,'\alpha::bot)Sequence))\ A =$
  $(OclIterate\ S\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ A)$
  and *undef_2_P*: $\bigwedge x.\ P\ x\ \bot = \bot$
  shows $f\ S\ \bot = \bot$
  by(*simp add*: *f_by_it undef_2_P*)

lemma *f2_by_iterate_defopt*:
  assumes *f_by_it*: $\bigwedge S\ A.\ (f\ (S::('\tau,'\alpha::bot)Sequence))\ A =$
  $(OclIterate\ S\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ A)$
  and *defopt_P*: $!!x\ y.\ \ (\tau \vDash \partial\ (P\ x\ y))$
  $= ((\tau \vDash \partial\ x) \wedge (\tau \vDash \partial\ y))$
  and *cpP_1*: $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
  and *cpP_2*: $\bigwedge x.\ cp\ ((P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ x)$
  shows $\tau \vDash \partial(f\ (S::('\tau,'\alpha::bot)Sequence)\ A) = ((\tau \vDash (\partial\ S)) \wedge (\tau \vDash (\partial\ A)))$
  by(*simp add*: *f_by_it OCL_is_defopt_OclIterate defopt_P cpP_1 cpP_2*)

lemma *f2_by_iterate_mtSequence*:
  $[\![\ \bigwedge S\ A.\ f\ (S::('\tau,'\alpha::bot)Sequence)\ A =$
  $OclIterate\ S\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ A\ ]\!]$
  $\Longrightarrow$
  $f\ []\ A = A$
  by(*simp*)

lemma *f2_by_iterate_including*:
  assumes *f_by_it*: $\bigwedge S\ A.\ (f\ (S::('\tau,'\alpha::bot)Sequence)\ A) =$
  $(OclIterate\ S\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL)\ A)$
  and *defS*: $\tau \vDash \partial(S::('\tau,('\alpha::bot))Sequence)$
  and *defx*: $\tau \vDash \partial(x::('\tau \Rightarrow\ '\alpha::bot))$
  and *cpP_1*: $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
  and *cpP_2*: $\bigwedge x.\ cp\ (P\ x)$
  shows $(f\ (OclIncluding\ S\ x)\ A\ \tau) = (P\ x\ (f\ S\ A)\ \tau)$

```
  by(simp add: f_by_it iterate_of_including defS defx cpP_1 cpP_2)


lemma f2_by_iterate_opt_including:
  assumes f_by_it: ⋀ S A. (f (S::('τ,'α::bot)Sequence) A) =
                     (OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A)
  and    P_undef_1:  !!y.  P ⊥ y = ⊥
  and    P_undef_2:  !!x.  P x ⊥ = ⊥
  and         cpP_1:  ⋀ y. cp (λ x. P x y)
  and         cpP_2:  ⋀ x. cp (P x)
  shows f (OclIncluding S x) A = P x (f S A)
  by(simp add: f_by_it iterate_opt_of_including
              P_undef_1 P_undef_2 cpP_1 cpP_2)
```

Union

defs

```
    OclUnion_def    :
    OclUnion        ≡ lift2(strictify(λX. strictify(λY. Abs_Sequence_0
                     (⌊(⌈Rep_Sequence_0 X⌉ @ ⌈Rep_Sequence_0 Y⌉)⌋))))

ocl_setup_op [OclUnion]

lemma OCL_is_def_OclUnion:
  ∂(OclUnion (X::('τ,'α::bot)Sequence)  Y) = (∂ X ∧ ∂  Y)
  apply(rule lift2_strict_is_isdef_fw_Sequence_Sequence[OF OclUnion_def])
  apply(simp)
  done

lemma OCL_is_defopt_OclUnion:
  (τ ⊨ ∂(OclUnion (X::('τ,'α::bot)Sequence)  Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
 apply(rule lift2_strictify_implies_LocalValid_defined_Sequence_Sequence[OF OclUnion_def])
  apply(simp)
  done
```

Its computational characterisation using iterate

```
lemma union_by_iterate:
  ((X::('τ,'α::bot)Sequence) ∪ (Y::('τ,'α::bot)Sequence)) =
   (Y->iterate(x;y=X | y->including (x::('τ,'α::bot) VAL) ))
  apply(rule ext)
  apply(rule iterate_universal, simp_all)
  apply(rule_tac X=X in Sequence_sem_cases)
  apply(simp add: OclUnion_def OclMtSequence_def ss_lifting')
  apply(simp add: OclUnion_def OclMtSequence_def ss_lifting')
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(frule DEF_X_Sequence', clarify)
  apply(rule_tac X=X in Sequence_sem_cases, simp_all)
  apply(simp_all add: OclUnion_def OclIncluding_def ss_lifting' neq_commute)
```

done

thm *f2_by_iterate_including*[*OF union_by_iterate*, *simplified*]

lemmas *union_of_mtSequence1* [*simp*] = *f2_by_iterate_mtSequence*[*OF union_by_iterate*]

lemma *union_of_mtSequence2* [*simp*]:
$[] \cup A = (A::('\tau,'\alpha::bot) Sequence)$
apply(*rule_tac X=A* **in** *Sequence_sem_cases_ext*)
apply(*simp_all*)
apply(*simp_all add: OclMtSequence_def OclUnion_def ss_lifting'*)
done

lemmas *union_of_including*[*simp*] = *f2_by_iterate_opt_including*[*OF union_by_iterate*, *simplified*]

lemma *union_assoc* [*simp*]:
$(A \cup (B \cup C)) = ((A \cup B) \cup (C::('\tau,'\alpha::bot) Sequence))$
apply(*rule_tac X=A* **in** *Sequence_sem_cases_ext*)
apply(*rule_tac X=B* **in** *Sequence_sem_cases*)
apply(*rule_tac X=C* **in** *Sequence_sem_cases*, *simp_all*)
apply(*simp_all add: OclUnion_def ss_lifting'*)
done

Properties of iterate over a union of two sequences

lemma *iterate_of_union*:
$[\![\tau \models \partial\ X]\!] \Longrightarrow$
$OclIterate\ ((X::('\tau,'\alpha::bot) Sequence) \cup Y)$
$(P::('\tau,'\alpha::bot) VAL \Rightarrow ('\tau,'c::bot) VAL \Rightarrow ('\tau,'c::bot) VAL)\ A\ \tau =$
$OclIterate\ Y\ P\ (OclIterate\ X\ P\ A)\ \tau$
apply(*case_tac* $\tau \models \not\partial\ Y$)
apply(*ocl_hypsubst*, *simp*)
apply(*simp add: localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Sequence'*)
apply(*rotate_tac 1*, *frule DEF_X_Sequence'*, *clarify*)
apply(*simp add: OclUnion_def OclIterate_def ss_lifting'*)
done

lemma *iterate_of_union_opt*:
$[\![\bigwedge x.\ P\ x \perp = \perp]\!] \Longrightarrow$
$OclIterate\ ((X::('\tau,'\alpha::bot) Sequence) \cup Y)$
$(P::('\tau,'\alpha::bot) VAL \Rightarrow ('\tau,'c::bot) VAL \Rightarrow ('\tau,'c::bot) VAL)\ A=$
$OclIterate\ Y\ P\ (OclIterate\ X\ P\ A)$
apply(*rule ext*)
apply(*case_tac* $x \models \partial\ X$)
apply(*simp add: iterate_of_union*)

```
  apply(simp add: sym[OF isUndefined_charn_local])
  apply(rule_tac A1=X in trans[OF cp_charn], simp_all)
  apply(rule_tac A2=X in sym[OF trans[OF cp_charn]], simp_all)
  apply(simp)
  done


lemma iterate_distrib_union:
  assumes    mtC: ⋀ B. g B A τ = B τ
  and        stepC: ⋀ B x y. g B (P x y) τ = P x (g B y) τ
  and        cp2_g: ⋀ x. cp (g x)
  and        cp2_P: ⋀ x. cp (P x)
  and         defX: τ ⊨ ∂ X
  and         defY: τ ⊨ ∂ Y
  shows OclIterate ((X::('τ,'α::bot)Sequence) ∪ Y)
                  (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A τ =
        g (OclIterate X P A) (OclIterate Y P A) τ
  by(simp add: iterate_of_union defX defY cp2_g cp2_P
             sym[OF iterate_fusion_opt] mtC stepC)


lemma iterate_distrib_union_opt:
  assumes undefC: ⋀ B. g B ⊥ = ⊥
  and          mtC: ⋀ B. g B A = B
  and        stepC: ⋀ B x y. g B (P x y) = P x (g B y)
  and    undef_2_P: ⋀ x. P x ⊥ = ⊥
  and        cp2_g: ⋀ x. cp (g x)
  and        cp2_P: ⋀ x. cp (P x)
  shows OclIterate ((X::('τ,'α::bot)Sequence) ∪ Y)
                  (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A =
        g (OclIterate X P A) (OclIterate Y P A)
  by(rule ext, simp add: iterate_of_union_opt undef_2_P cp2_g cp2_P
                       sym[OF iterate_fusion] undefC mtC stepC)
```

And the corresponding forward rules for *unary* functions

```
lemma f1_by_iterate_distrib_union:
  assumes f_by_it: ⋀ S. (f (S::('τ,'α::bot)Sequence)) =
                      (OclIterate S (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) A)
  and          mtC: ⋀ B. g B A τ = B τ
  and        stepC: ⋀ B x y. g B (P x y) τ = P x (g B y) τ
  and        cp2_g: ⋀ x. cp (g x)
  and        cp2_P: ⋀ x. cp (P x)
  and         defX: τ ⊨ ∂ X
  and         defY: τ ⊨ ∂ Y
  shows f (X ∪ Y) τ = g (f X) (f Y) τ
  by(simp add: f_by_it iterate_distrib_union defX defY cp2_g cp2_P mtC stepC)

lemma f1_by_iterate_distrib_union_opt:
```

assumes *f_by_it*: $\bigwedge$ *S*. (*f* (*S*::('τ,'α::*bot*)*Sequence*)) =
(*OclIterate S* (*P*::('τ,'α::*bot*) *VAL* ⇒ ('τ,'c::*bot*) *VAL* ⇒ ('τ,'c::*bot*) *VAL*) *A*)
and *undefC*: $\bigwedge$ *B*. *g B* ⊥ = ⊥
and *mtC*: $\bigwedge$ *B*. *g B A* = *B*
and *stepC*: $\bigwedge$ *B x y*. *g B* (*P x y*) = *P x* (*g B y*)
and *undef_2_P*: $\bigwedge$ *x*. *P x* ⊥ = ⊥
and *cp2_g*: $\bigwedge$ *x*. *cp* (*g x*)
and *cp2_P*: $\bigwedge$ *x*. *cp* (*P x*)
shows *f* (*X* ∪ *Y*) = *g* (*f X*) (*f Y*)
by(*simp add*: *f_by_it iterate_distrib_union_opt*
undef_2_P cp2_g cp2_P undefC mtC stepC)

Append

defs

OclAppend_def :
OclAppend ≡ *lift2* (*strictify* (λ*S*. *strictify* (λ*e*. *Abs_Sequence_0*
(⌊(*concat* [⌈*Rep_Sequence_0 S*⌉, [*e*]])⌋))))

ocl_setup_op [*OclAppend*]

lemma *append_including_UC* [*simp*]:
OclAppend (*S*::('τ,'α::*bot*)*Sequence*) (*x*::('τ,'α) *VAL*) = *S* `->including` *x*
by(*simp add*: *OclAppend_def OclIncluding_def*)

Prepend

defs

OclPrepend_def :
OclPrepend ≡ *lift2* (*strictify* (λ*S*. *strictify* (λ*e*. *Abs_Sequence_0*
(⌊*e* # ⌈*Rep_Sequence_0 S*⌉⌋))))

ocl_setup_op [*OclPrepend*]

lemma *prepend_union_UC* [*simp*]:
OclPrepend (*X*::('τ,'α::*bot*)*Sequence*) (*a*::('τ,'α) *VAL*) = (([] `->including` *a*) ∪ *X*)
apply(*rule_tac X*=*X* **in** *Sequence_sem_cases_ext*)
apply(*simp_all*)
apply(*simp_all add*: *OclPrepend_def OclIncluding_def OclUnion_def*
OclMtSequence_def ss_lifting' neq_commute)
done

lemma *test''*: (*OclPrepend* (*OclPrepend* (*X*::('τ,'α::*bot*)*Sequence*) (*a*::('τ,'α) *VAL*)) (*a*::('τ,'α) *VAL*))
= *bla*
apply(*simp*) **oops**

lemma *OCL_is_def_OclPrepend*:
∂(*OclPrepend* (*X*::('τ,'α::*bot*)*Sequence*) (*Y*::('τ,'α::*bot*) *VAL*)) = (∂ *X* $\wedge$ ∂ *Y*)
apply(*rule lift2_strict_is_isdef_fw_Sequence_Val*[*OF OclPrepend_def*])
apply(*simp*)
done

lemma *OCL_is_defopt_OclPrepend*:
($\tau \models \partial$(*OclPrepend* (*X*::('τ,'α::*bot*)*Sequence*) (*Y*::('τ,'α::*bot*) *VAL*))) = (($\tau \models \partial$ *X*) $\wedge$ ($\tau \models$ ∂ *Y*))
apply(*rule lift2_strictify_implies_LocalValid_defined_Sequence_Val*[*OF OclPrepend_def*])
apply(*simp*)
done

Some sort of case exhaustion over prepend

lemma *prepend_charn1*:
$[\![\bot \notin$ *set xs*; *xs* $\neq$ [] $]\!] \Longrightarrow$
$\exists$ (*S*::'α::*bot Sequence_0*) *a*. (*Abs_Sequence_0* $\llcorner$*xs*$\lrcorner$) = (*OclPrepend* (*lift0 S*) ((*lift0* (*a*::'α)))) τ
apply(*cases xs*, *simp_all*)
apply(*rule_tac x*=*Abs_Sequence_0* $\llcorner$*list*$\lrcorner$ **in** *exI*)
apply(*rule_tac x*=*a* **in** *exI*)
apply(*auto simp*: *OclMtSequence_def OclIncluding_def OclUnion_def ss_lifting'*)
done

lemma *prepend_charn3*:
$\tau \models$ *OclNotEmpty* (*S*::('τ,'α::*bot*)*Sequence*) $\Longrightarrow$
$\exists$ (*S'*::('τ,'α::*bot*)*Sequence*) *a*. $\tau \models$ (*S* $\triangleq$ (*OclPrepend S'* (*a*::('τ,'α) *VAL*)))
oops

lemma *prepend_charn2*:
$[\![\bot \notin$ *set xs*; *xs* $\neq$ [] $]\!] \Longrightarrow$
$\exists$ (*S*::('τ,'α::*bot*)*Sequence*) *a*. *Abs_Sequence_0* $\llcorner$*xs*$\lrcorner$ = (*OclPrepend S* (*a*::('τ,'α) *VAL*)) τ
apply(*cases xs*, *simp_all*)
apply(*rule_tac x*=*lift0* (*Abs_Sequence_0* $\llcorner$*list*$\lrcorner$) **in** *exI*)
apply(*rule_tac x*=*lift0 a* **in** *exI*)
apply(*auto simp*: *OclMtSequence_def OclIncluding_def OclUnion_def ss_lifting'*)
done

Reverse induction on sequences: induction over prepend

lemma *Sequence_induct_prepend_0*:
assumes *undefCase* : *P* $\bot$
and *mtCase* : *P* ([] τ)
and *stepCase* : $\bigwedge$ (*S*::'τ::*bot Sequence_0*) (*x*::'τ::*bot*).
$[\![$*P S*; *DEF S*; *DEF x*$]\!] \Longrightarrow$ (*P* ((*OclPrepend* (*lift0 S*) (*lift0* (*x*::'τ))) τ))
shows *P* (*S*::'τ::*bot Sequence_0*)

```
  apply(rule Abs_Sequence_0_induct_charn, rule undefCase)
  apply(erule rev_mp)
  apply(induct_tac y)
  apply(insert mtCase, simp add: OclMtSequence_def lift0_def)
  apply(auto)
  apply(rotate_tac -1, drule_tac x=a and S=Abs_Sequence_0 ⌊list⌋ in stepCase)
  apply(erule DEF_Abs_Sequence)
  apply(simp_all add: OclPrepend_def DEF_def neq_commute
                      lift0_def lift2_def strictify_def)
  done

lemma Sequence_induct_prepend:
  assumes undefCase : P ⊥ τ
  and mtCase    : P [] τ
  and stepCase  : ⋀ (S::('τ,'α::bot)Sequence) (x::('τ,'α::bot) VAL).
                   ⟦ P S τ; τ ⊨ ∂ S; τ ⊨ ∂ x ⟧ ⟹ P (OclPrepend S (x::('τ,'α) VAL)) τ
  and cpP       : cp P
  shows                P (S::('τ,'α::bot)Sequence) τ
  apply(insert cpP)
  apply(frule cp_by_cpify[THEN iffD1])
  apply(erule_tac x=S in allE)
  apply(erule_tac x=τ in allE)
  apply(simp)
  apply(frule cp_by_cpify[THEN iffD1])
  apply(rule_tac τ=τ in Sequence_induct_prepend_0)
  apply(insert undefCase mtCase)
  apply(simp_all add: OclUndefined_def lift0_def)
  apply(drule stepCase)
  apply(simp_all add: localValidDefined2sem lift0_def)
  done
```

iterate on prepend

```
lemma iterate_of_prepend:
  assumes defS: τ ⊨ ∂(S::('τ,('α::bot))Sequence)
  and      defx: τ ⊨ ∂(x::('τ ⇒ 'α::bot))
  and     cpP_1: ⋀ y. cp (λ x. P x y)
  and     cpP_2: ⋀ x. cp (P x)
  shows ((OclIterate (OclPrepend S (x::('τ,'α) VAL)) P A) τ) =
        ((OclIterate (S::(('τ,'α::bot)Sequence)) P (P x A)) τ)
  apply(insert defS defx cpP_1 cpP_2, simp)
  apply(subst iterate_of_union)
  apply(simp add: OCL_is_defopt_OclIncluding)
  apply(rule_tac A=(OclIterate [] ->including x  P A) in cp_charn)
  apply(simp_all add: iterate_of_including)
  done
```

Includes and excludes

defs

$OclIncludes\_def: OclIncludes \equiv lift2(strictify\ (\lambda X.\ strictify\ (\lambda x.$
$\llcorner x \in set \ulcorner Rep\_Sequence\_0\ X \urcorner \lrcorner)))$

$OclExcludes\_def: OclExcludes \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda x.$
$\llcorner x \notin set \ulcorner Rep\_Sequence\_0\ X \urcorner \lrcorner)))$

ocl_setup_op $[OclIncludes, OclExcludes]$

lemma *includes_charn1* :
$(\tau \vDash x \in S) \Longrightarrow ((x\ \tau) \in set \ulcorner Rep\_Sequence\_0\ (S\ \tau)\urcorner)$
apply(*case_tac* $(S\ \tau) \neq \bot$)
apply(*case_tac* $(x\ \tau) \neq \bot$)
apply(*auto simp*: *OclIncludes_def localValid2sem ss_lifting'*)
done

lemma *includes_charn2* :
$[\![\ DEF(S\ \tau);\ (x\ \tau) \in set \ulcorner Rep\_Sequence\_0\ (S\ \tau)\urcorner\ ]\!] \Longrightarrow \tau \vDash x \in S$
apply(*unfold DEF_def, frule DEF_X_Sequence'*)
apply(*auto simp*: *OclIncludes_def localValid2sem ss_lifting'*)
done

Higher Properties of includes and excludes

lemma *exludes_not_includes*[*simp*]:
$x \notin X = \neg\ (x{::}('\tau,\ '\alpha)\ VAL) \in (X{::}('\tau,\ '\alpha{::}bot)\ Sequence)$
apply(*rule ext*)
apply(*simp add*: *OclExcludes_def OclIncludes_def OclNot_def ss_lifting'*)
done

Currently a non-strict body is taken although a strict one is possible. This should be corrected as the experience has shown that strict functions are much nicer to work with.

lemma *includes_by_iterate*:
$a \in (X{::}('\tau,'\alpha{::}bot)Sequence) =$
$OclIterate\ X\ (\lambda\ (x{::}('\tau,'\alpha{::}bot)\ VAL)\ y.\ (x \doteq a)\ \dot{\vee}\ y)$
$(if\ (\partial\ a)\ then\ \mathsf{F}\ else\ \bot\ endif)$
apply(*rule ext*)
apply(*rule iterate_universal, simp_all*)
apply(*simp add*: *OclIncludes_def OclMtSequence_def OclFalse_def OclIf_def OclIsDefined_def OclUndefined_def ss_lifting'*)
apply(*simp add*: *localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Sequence', clarify*)
apply(*auto simp*: *OclIncludes_def OclIncluding_def OclTrue_def OclSor_def OclNot_def OclSand_def OclStrictEq_def*

```
                   ss_lifting' neq_commute)
  done

lemmas includes_mtSequence [simp] = f1_by_iterate_mtSequence[OF includes_by_iterate]
thm f1_by_iterate_opt_including[OF includes_by_iterate, simplified]

lemma excludes_by_iterate:
  a ∉ (X::('τ,'α::bot)Sequence) =
   OclIterate X (λ (x::('τ,'α::bot) VAL) y. (x ‘<>‘ a) ∧̇ y)
               (if (∂ a) then ⊤ else ⊥ endif)
  apply(rule ext, simp add: includes_by_iterate)
  apply(rule iterate_fusion, simp_all)
  apply(simp_all add: OclNot_def OclTrue_def OclFalse_def OclIf_def
                      OclSand_def OclSor_def OclUndefined_def ss_lifting')
  done
```

containment in the empty list can be reduced to definedness

```
lemma includes_of_mtSequence[simp]:
  ¬ (τ ⊨ (x::('τ, 'α) VAL) ∈ ([]::('τ,'α::bot)Sequence))
  by(simp add: OclIncludes_def OclMtSequence_def OclNot_def
               localValid2sem ss_lifting')


lemma includes_charn1_including[simp]:
  ⟦ τ ⊨ ∂ (a::('τ,'α) VAL); τ ⊨ ∂ (X::('τ,'α::bot)Sequence) ⟧ ⟹
   τ ⊨ a ∈ (X->including a)
  apply(simp only: localValidDefined2sem DEF_def)
  apply(frule DEF_X_Sequence', clarify)
  apply(simp add: localValid2sem OclIncludes_def OclIncluding_def
                  ss_lifting' neq_commute)
  done

lemma includes_charn2_including[simp]:
  ⟦ τ ⊨ ∂ (a::('τ,'α) VAL); τ ⊨ ∂ (b::('τ,'α) VAL); τ ⊨ a ∈ (X::('τ,'α::bot)Sequence) ⟧ ⟹
   τ ⊨ a ∈ (X->including b)
  apply(case_tac X τ ≠ ⊥)
  apply(frule DEF_X_Sequence', clarify)
  apply(simp_all only: localValidDefined2sem DEF_def)
  apply(simp_all add: localValid2sem OclIncludes_def OclIncluding_def
                  ss_lifting' neq_commute)
  apply(drule neq_commute[THEN iffD1], simp)
  done



lemma includes_of_collectionRange:
  (x::('τ Integer)) ∈ ((OclCollectionRange a b)::('τ,Integer_0)Sequence) =
   ((a::('τ Integer)) ≤ x) ∧̇ (x ≤ (b::('τ Integer)))
  apply(rule ext)
```

apply(*case_tac DEF* (*x xa*), *case_tac DEF* (*a xa*), *case_tac DEF* (*b xa*))
apply(*simp add*: *DEF_def_both*, *clarify*)
apply(*subgoal_tac* $\perp \notin$ *set* (*map* ($\lambda z.$ $\llcorner$((*int z*) + $\ulcorner$(*a xa*)$\urcorner$)$\lrcorner$) [*0*..<(*nat* (($\ulcorner$(*b xa*)$\urcorner$ − $\ulcorner$(*a xa*)$\urcorner$) + *1*))]))
apply(*auto simp*: *OclIncludes_def OclCollectionRange_def OclLe_def*
OclSand_def ss_lifting′, *arith*)
apply(*rule_tac x*=*nat*($\ulcorner$(*x xa*)$\urcorner$ − $\ulcorner$(*a xa*)$\urcorner$) **in** *image_eqI*, *auto*)
done

lemma *includes_of_union* [*simp*]:
⟦ $\tau \models \partial$ *X*; $\tau \models \partial$ *Y* ⟧ $\Longrightarrow$
((*a*::(′τ,′α::*bot*) *VAL*) $\in$ ((*X*::(′τ,′α::*bot*)*Sequence*) $\cup$ *Y*)) τ =
((*a* $\in$ *X*) $\vee$ (*a* $\in$ *Y*)) τ
apply(*simp add*: *localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Sequence*′)
apply(*rotate_tac 1*, *frule DEF_X_Sequence*′, *clarify*)
apply(*simp add*: *OclIncludes_def OclUnion_def OclOr_def*
OclAnd_def OclNot_def ss_lifting′)
done

lemma *test*′′′: $\tau \models$ (*1*::′τ *Integer*) $\in$ ((*mkSequence*[*1*::′τ *Integer*])::(′τ, *Integer_0*)*Sequence*)
oops

lemma *not_includes_of_mtSequence*:
($\tau \models \neg$ (*x*::(′τ, ′α) *VAL*) $\in$ ([]::(′τ, ′α::*bot*) *Sequence*)) = ($\tau \models \partial$ *x*)
by(*simp add*: *localValidNot2sem localValidDefined2sem OclIncludes_def*
OclMtSequence_def ss_lifting′)

lemma *mtSequence_Includes_of_OLD* :
$\tau \models \partial$ *x* $\Longrightarrow$
$\tau \models$ (*x*::(′τ, ′α) *VAL*) $\notin$ ([]::(′τ, ′α::*bot*) *Sequence*)
by(*simp*, *ocl_subst*, *simp*)

Note: this will be subsumed by OCL_is_defopt_OclIncludes if `ocl_setup_op` is working correctly on collection types

Emptiness

defs

OclIsEmpty_def :
OclIsEmpty $\equiv$ *lift1* (*strictify* ($\lambda X.$ $\llcorner$($\ulcorner$*Rep_Sequence_0 X*$\urcorner$ = [])$\lrcorner$))

$OclNotEmpty\_def$:
$OclNotEmpty \equiv lift1\ (strictify\ (\lambda X.\ \llcorner(\ulcorner Rep\_Sequence\_0\ X\urcorner \neq [])\lrcorner))$

ocl_setup_op [*OclIsEmpty*, *OclNotEmpty*]

The normal form for emptiness testing:

we prefer a canical form were notEmpty is written using isEmpty

lemma *notEmpty_not_isEmpty_conv*[*simp*]:
$\not\doteq\emptyset(X::('\tau,\ '\alpha::bot)\ Sequence) = \neg\ (\doteq\emptyset\ X)$
apply(*rule ext*)
apply(*simp add*: *OclNotEmpty_def OclIsEmpty_def OclNot_def*
ss_lifting')
done

further on the emptiness test is reduced to an equality to a constant

lemma *isEmpty_stricteq_mtSequence_conv*[*simp*]:
$(\doteq\emptyset\ X) = ((X::('\tau,'\alpha::bot)Sequence) \doteq [])$
apply(*rule ext*)
apply(*case_tac* $X\ x \neq \bot$)
apply(*frule DEF_X_Sequence'*, *clarify*)
apply(*simp_all add*: *OclIsEmpty_def OclMtSequence_def OclStrictEq_def*
Abs_Sequence_0_inject_charn ss_lifting')
done

Properties of emptiness:

lemma *isEmpty_of_mtSequence*:
$\tau \models \doteq\emptyset\ []$
by (*simp*)

lemma *isEmpty_of_including*:
$[\![\ \tau \models \partial\ (a::('\tau,'\alpha)\ VAL);\ \tau \models \partial\ (X::('\tau,'\alpha::bot)Sequence)\ ]\!] \Longrightarrow$
$\tau \models \neg\ (\doteq\emptyset(X \text{->}\mathtt{including}\ a))$
by(*simp*)

lemma *isEmpty_union* [*simp*]:
$(((X::('\tau,'\alpha::bot)Sequence) \cup Y) \doteq []) = (X \doteq []) \mathbin{\dot\wedge} (Y \doteq [])$
apply(*rule_tac X=X* **in** *Sequence_sem_cases_ext*)
apply(*rule_tac X=Y* **in** *Sequence_sem_cases*, *simp_all*)
apply(*simp_all add*: *OclUnion_def OclStrictEq_def OclMtSequence_def OclSand_def*
ss_lifting' Abs_Sequence_0_inject_charn)
done

lemma *notEmpty_of_mtSequence*:
$\tau \models \neg\ (\not\doteq\emptyset\ [])$
by(*simp*)

lemma *notEmpty_of_including*:
  $\llbracket\ \tau \models \partial\ (a::('\tau,'\alpha)\ VAL);\ \tau \models \partial\ (X::('\tau,'\alpha::bot)Sequence)\ \rrbracket \Longrightarrow$
   $\tau \models \neq \emptyset\ (X$ `->including` $a)$
  by(*simp*)

Size

defs

$$OclSize\_def\ :\ OclSize \equiv lift1\ (strictify\ (\lambda X.\ \llcorner int\ (Nat.size\ \ulcorner Rep\_Sequence\_0\ X\urcorner)\lrcorner))$$

ocl_setup_op [*OclSize*]

lemma *size_by_iterate*:
  $((self::('\tau,'\alpha::bot)Sequence)\ {-}{>}size()) =$
   $OclIterate\ self\ (\lambda\ (x::('\tau,'\alpha::bot)\ VAL)\ (y::'\tau\ Integer).\ y + 1)\ 0$
  apply(*rule ext*)
  apply(*rule iterate_universal*)
  apply(*simp_all*)
  apply(*simp_all add: OclSize_def OclIterate_def plus_def OclMtSequence_def*
        OclUndefined_def OclIncluding_def localValidDefined2sem
        OCL_Integer.Zero_ocl_int_def OCL_Integer.One_ocl_int_def
        ss_lifting')
  apply(*frule DEF_X_Sequence'*)
  apply(*auto simp: neq_commute*)
  done

lemmas *size_of_mtSequence*[*simp*] = *f1_by_iterate_mtSequence*[*OF size_by_iterate*]

lemma *size_of_including*[*simp*]:
  $\llbracket\ \tau \models \partial\ (x::('\tau,'\alpha::bot)\ VAL)\ \rrbracket \Longrightarrow$
  $(\|(S::('\tau,'\alpha::bot)Sequence)$ `->including` $x\ \|)\ \tau = (\|S\| + 1)\ \tau$
  apply(*case_tac* $\tau \models \emptyset\ S$)
  apply(*ocl_hypsubst, simp*)
  apply(*simp add: f1_by_iterate_including*[*OF size_by_iterate, simplified*])
  done

lemma *size_eq_zero_isEmtpy_UC*[*simp*]:
  $(\|(S::('\tau,'\alpha::bot)Sequence)\| \doteq 0) = (S \doteq [])$
  apply(*rule ext*)
  apply(*case_tac* $S\ x \neq \bot$)
  apply(*frule DEF_X_Sequence', clarify*)
  apply(*simp_all add: OclSize_def OclStrictEq_def OclMtSequence_def*

```
                    OCL_Integer.Zero_ocl_int_def Abs_Sequence_0_inject_charn
                    localValid2sem ss_lifting')
  done


lemma size_of_union_UC [simp]:
  (‖((X::('τ,'α::bot)Sequence) ∪ Y)‖) = (‖X‖ + ‖Y‖)
  apply(rule f1_by_iterate_distrib_union_opt[OF size_by_iterate, of op +, simplified])
  apply(rule ext, simp add: plus_def Zero_ocl_int_def ss_lifting')
  apply(rule ext, simp add: plus_def One_ocl_int_def ss_lifting')
  done
```

Induction over two lists

```
lemma Sequence_induct2_including_0:
  assumes undefCase: P ⊥ ⊥
  and      equalSize: ‖(lift0 X)‖ τ = ‖(lift0 Y)‖ τ
  and        mtCase: P ([] τ) ([] τ)
  and      stepCase: ⋀ (X::'τ::bot Sequence_0) (a::'τ::bot) (Y::'α::bot Sequence_0) (b::'α::bot).

                    ⟦ P X Y; DEF X; DEF a; DEF Y; DEF b⟧ ⟹
                   (P ((OclIncluding (lift0 X) (lift0 a)) τ) ((OclIncluding (lift0 Y) (lift0 b)) τ))
  shows P (X::'τ::bot Sequence_0) (Y::'α::bot Sequence_0)
  apply(insert equalSize, erule rev_mp)
  apply(rule Abs_Sequence_0_induct_charn)
  apply(simp add: OclSize_def ss_lifting' undefCase)
  apply(erule rev_mp)+
  apply(rule_tac x=X in Abs_Sequence_0_induct_charn)
  apply(simp add: OclSize_def ss_lifting' undefCase)
  apply(rule impI)+

  apply(frule rev_mp) prefer 2 apply(assumption, rotate_tac 1)
  apply(frule rev_mp) prefer 2 apply(assumption, rotate_tac 1)
  apply(frule rev_mp) prefer 2 apply(assumption, rotate_tac 1)
  apply(rule list_rev_induct2)
  apply(insert mtCase)
  apply(auto simp: OclSize_def OclMtSequence_def ss_lifting')
  apply(rotate_tac -1, drule_tac a=x and X=Abs_Sequence_0 ⌞xs⌟ and
                              b=yb and Y=Abs_Sequence_0 ⌞ys⌟ in stepCase)
  apply(simp_all add: OclIncluding_def neq_commute ss_lifting')
  done


lemma Sequence_induct2_including:
  assumes equalSize: τ ⊨ ‖X‖ ≜ ‖Y‖
  and      undefCase: P ⊥ ⊥ τ
  and        mtCase: P [] [] τ
  and       stepCase: ⋀ (X::('τ,'α::bot)Sequence) (a::('τ,'α::bot) VAL)
                  (Y::('τ,'c::bot)Sequence) (b::('τ,'c::bot) VAL).
```

```
                  ⟦ P X Y τ; τ ⊨ ∂ X; τ ⊨ ∂ a; τ ⊨ ∂ Y; τ ⊨ ∂ b⟧ ⟹
                   P (OclIncluding X a) (OclIncluding Y b) τ
  and       cp1_P: ⋀ y. cp (λ x. P x y)
  and       cp2_P: ⋀ x. cp (P x)
  shows            P (X::('τ,'α::bot)Sequence) (Y::('τ,'c::bot)Sequence) τ
  apply(insert cp1_P cp2_P, simp add: cp_by_cpify)
  apply(insert cp1_P cp2_P)
  apply(rule subst, rule sym, rule_tac x=X in cp_subst, simp)
  apply(rule subst, rule sym, rule_tac x=Y in cp_subst, simp)
  apply(rule_tac τ=τ in Sequence_induct2_including_0)
  apply(simp add: lift0_def)
  oops
```

Count

defs

```
  OclCount_def : OclCount ≡ lift2 (strictify  (λX. strictify (λx.
                     ⌊int (Nat.size(List.filter (op = x) ⌜Rep_Sequence_0 X⌝))⌋)))
```

ocl_setup_op [*OclCount*]

Its computational characterisation by iterate

```
lemma count_by_iterate:
  ((S::('τ,'α::bot)Sequence) −>count((a::('τ,'α) VAL))) =
   (OclIterate S
           (λ (x::('τ,'α::bot) VAL) (y::'τ Integer).if  (x ≐ a)then (y+1) else (y) endif )
           (if (∂ a) then 0 else⊥ endif ) )
  apply(rule ext)
  apply(rule iterate_universal)
  apply(simp_all)
  apply(simp_all add: OclMtSequence_def OclCount_def
              OCL_Integer.Zero_ocl_int_def OCL_Integer.One_ocl_int_def
              OclIf_def OclIsDefined_def OclUndefined_def OclStrictEq_def
              OclIncluding_def OclLocalValid_def plus_def OclTrue_def
              ss_lifting')
  apply(frule DEF_X_Sequence')
  apply(auto simp: neq_commute)
  done

thm f1_by_iterate_including[OF count_by_iterate, simplified]

lemma count_of_mtSequence [simp]:
  τ ⊨ (∂ (x::('τ,'α) VAL)) ⟹
   (([]::('τ,'α::bot)Sequence) ->count x ) τ = 0 τ
  by(simp add: count_by_iterate, ocl_subst, simp)

thm f1_by_iterate_including[OF count_by_iterate, simplified]
thm f1_by_iterate_opt_including[OF count_by_iterate, simplified]
```

thm *f1_by_iterate_distrib_union*[*OF count_by_iterate, of op* +, *simplified*]
thm *f1_by_iterate_distrib_union_opt*[*OF count_by_iterate, of op* +, *simplified*]

lemma *plus_assoc1*: $((X::('\tau\ Integer)) + Y) + Z = X + (Y + Z)$
  by(*rule ext, simp add: plus_def ss_lifting'*)

lemma *plus_assoc2*: $(X::('\tau\ Integer)) + (Y + Z) = Y + (X + Z)$
  by(*rule ext, simp add: plus_def ss_lifting'*)

lemmas *plus_AC* = *plus_commute plus_assoc1 plus_assoc2*

lemma *plus_zero'* [*simp*]: $(X::('\tau\ Integer)) + 0 = X \wedge 0 + (X::('\tau\ Integer)) = X$
  by(*rule conjI*, (*rule ext, simp add: plus_def ss_lifting' Zero_ocl_int_def*)+)

lemma *count_of_union* [*simp*]:
  $(((X::('\tau,'\alpha::bot)Sequence) \cup Y) \text{->count } (a::('\tau,'\alpha)VAL)) = ((X \text{->count } a) + (Y \text{->count } a))$
  apply(*rule ext, case_tac x* $\models \partial$ *a, ocl_hypsubst, simp*)
  apply(*case_tac x* $\models \partial$ *X, ocl_hypsubst, simp*)
  apply(*case_tac x* $\models \partial$ *Y, ocl_hypsubst, simp*)
  apply(*simp add: count_by_iterate*)
  apply(*ocl_subst*)
  apply(*simp add: iterate_distrib_union plus_AC if_distrib_strict_alt*)
  done

lemma *includes_by_count_UC*[*simp*]:
  $S \text{->count } x > 0 = (x::('\tau,'\alpha)VAL) \in (S::('\tau,'\alpha::bot)Sequence)$
  apply(*rule Sequence_sem_cases_ext, simp_all*)
  apply(*auto simp: OclIncludes_def OclCount_def OclGreater_def filter_empty_conv*
    OclLess_def OclNot_def Zero_ocl_int_def ss_lifting')
  done

lemma *excludes_by_count_UC*[*simp*]:
  $S \text{->count } x \doteq 0 = \neg ((x::('\tau,'\alpha)VAL) \in (S::('\tau,'\alpha::bot)Sequence))$
  apply(*rule Sequence_sem_cases_ext, simp_all*)
  apply(*auto simp: OclIncludes_def OclCount_def OclStrictEq_def OclNot_def*
    Zero_ocl_int_def ss_lifting' filter_empty_conv)
  done

Subsequence

constdefs

$OclSubSequence :: [('\tau,'\alpha::bot)Sequence,'\tau\ Integer,'\tau\ Integer] => ('\tau,'\alpha\ )\ Sequence$

$$
\begin{aligned}
OclSubSequence \equiv\ & lift3\ (strictify\ (\lambda S.\ strictify\ (\lambda l.\ strictify\ (\lambda u.\ Abs\_Sequence\_0 \\
& (\llcorner take\ (nat\ (\ulcorner u \urcorner - \ulcorner l \urcorner + 1)) \\
& \quad (List.drop\ (nat\ (\ulcorner l \urcorner - 1)) \\
& \quad\quad (\ulcorner Rep\_Sequence\_0\ S \urcorner)) \lrcorner)))))
\end{aligned}
$$

ocl_setup_op [*OclSubSequence*]

Because the `ocl_setup_op` method doesn't deal with `lift3` we have to setup the simple properties by ourselves.

lemma *OCL_undef_1_OclSubsequence*[*simp*]:
$OclSubSequence\ (\bot::('\tau,'\alpha::bot)Sequence)\ a\ b = \bot$
by(*simp add: OclUndefined_def OclSubSequence_def ss_lifting′*)

lemma *OCL_undef_2_OclSubsequence*[*simp*]:
$OclSubSequence\ (S::('\tau,'\alpha::bot)Sequence)\ \bot\ b = \bot$
by(*rule ext, simp add: OclUndefined_def OclSubSequence_def ss_lifting′*)

lemma *OCL_undef_3_OclSubsequence*[*simp*]:
$OclSubSequence\ (S::('\tau,'\alpha::bot)Sequence)\ a\ \bot = \bot$
by(*rule ext, simp add: OclUndefined_def OclSubSequence_def ss_lifting′*)

lemma *OCL_cp_OclSubsequence*[*simp,intro!*]:
$[\![\ cp\ P;\ cp\ F;\ cp\ F'\ ]\!] \Longrightarrow$
$cp\ (\lambda\ X.\ OclSubSequence\ ((P\ X)::('\tau,'\alpha::bot)Sequence)\ (F\ X)\ (F'\ X))$
by(*simp add: OclSubSequence_def*)

Higher properties of subsequence

Note: `subsequence` is in our case not undefined except for undefined arguments. To avoid undefinedness is quite favorable in general as it makes proofs simpler and reduces the side conditions to check. But our implementation still fullfills the requirements of the standard.

lemma *size_of_subsequence*:
$[\![\ \tau \models 1 \leq a;\ \tau \models a \leq b - 1;\ \tau \models b \leq \|S\|\ ]\!] \Longrightarrow$
$\|(OclSubSequence\ (S::('\tau,'\alpha::bot)Sequence)\ a\ b)\|\ \ \tau = (b - a + 1)\ \tau$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValid2sem*)
apply(*case_tac* $b\ \tau \neq \bot$, *frule neq_commute*[*THEN iffD1*])
apply(*case_tac* $a\ \tau \neq \bot$, *rotate_tac* -1, *frule neq_commute*[*THEN iffD1*])
apply(*auto simp: OclIncluding_def OclSubSequence_def OclSize_def*
One_ocl_int_def plus_def minus_def ss_lifting′
OclLe_def OclLess_def notin_set_take notin_set_drop)
apply(*arith, case_tac* $((b\ \tau) = down)$, *simp_all*)
done

lemma *subsequence_mtSequence_conv*:
$\llbracket\ \tau \models \partial\ S;\ \tau \models b < a\ \rrbracket \Longrightarrow$
$(OclSubSequence\ (S::('\tau,'\alpha::bot)Sequence)\ a\ b)\ \tau = []\ \tau$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValidDefined2sem DEF_def*)
apply(*case_tac b* $\tau \neq \bot$, *frule neq_commute[THEN iffD1]*)
apply(*case_tac a* $\tau \neq \bot$, *rotate_tac −1, frule neq_commute[THEN iffD1]*)
apply(*auto simp: OclIncluding_def OclSubSequence_def OclMtSequence_def localValid2sem OclLess_def notin_set_take notin_set_drop ss_lifting'*)
done

InsertAt

defs

OclInsertAt_def :
$OclInsertAt \equiv lift3(strictify(\lambda S.\ strictify(\lambda i.\ strictify(\lambda x.\ Abs\_Sequence\_0($
$if\ (1 <= \ulcorner i\urcorner \wedge \ulcorner i\urcorner <= (1+int(Nat.size\ulcorner Rep\_Sequence\_0\ S\urcorner)))$
$then\ (_\llcorner concat\ [take\ (nat\ (op - \ulcorner i\urcorner\ 1))\ (\ulcorner Rep\_Sequence\_0\ S\urcorner),\ [x],$
$List.drop\ (nat\ (op - \ulcorner i\urcorner\ 1))\ (\ulcorner Rep\_Sequence\_0\ S\urcorner)]_\lrcorner)$
$else\ \bot\ )))))$

ocl_setup_op [*OclInsertAt*]

Because the `ocl_setup_op` method doesn't deal with `lift3` we have to setup the simple properties by ourselves.

lemma *OCL_cp_OclInsertAt[simp,intro!]*:
$\llbracket\ cp\ P;\ cp\ F;\ cp\ F'\ \rrbracket \Longrightarrow$
$cp\ (\lambda\ X.\ OclInsertAt\ ((P\ X)::('\tau,'\alpha::bot)Sequence)\ (F\ X)\ ((F'\ X)::('\tau,'\alpha)\ VAL))$
by(*simp add: OclInsertAt_def*)

lemma *OCL_undef_1_OclInsertAt[simp]*:
$OclInsertAt\ (\bot::('\tau,'\alpha::bot)Sequence)\ i\ (x::('\tau,'\alpha)\ VAL) = \bot$
by(*simp add: OclUndefined_def OclInsertAt_def ss_lifting'*)

lemma *OCL_undef_2_OclInsertAt[simp]*:
$OclInsertAt\ (S::('\tau,'\alpha::bot)Sequence)\ \bot\ (x::('\tau,'\alpha)\ VAL) = \bot$
by(*rule ext, simp add: OclUndefined_def OclInsertAt_def ss_lifting'*)

lemma *OCL_undef_3_OclInsertAt[simp]*:
$OclInsertAt\ (S::('\tau,'\alpha::bot)Sequence)\ i\ (\bot::('\tau,'\alpha)\ VAL) = \bot$
by(*rule ext, simp add: OclUndefined_def OclInsertAt_def ss_lifting'*)

Higher properties of insertAt

lemma *includes_of_insertAt*:
$\llbracket\ \tau \models \partial\ (S::('\tau,'\alpha::bot)Sequence);\ \tau \models \partial\ (x::('\tau,'\alpha)\ VAL);$

$\tau \models 1 \leq i;\ \tau \models i \leq (\|S\| + 1)\ \rrbracket \Longrightarrow$
$\tau \models x \in (OclInsertAt\ S\ i\ x)$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValidDefined2sem DEF_def*)
apply(*case_tac i* $\tau \neq \perp$, *frule neq_commute*[*THEN iffD1*])
apply(*case_tac x* $\tau \neq \perp$, *rotate_tac* -1, *frule neq_commute*[*THEN iffD1*])
apply(*auto simp: OclInsertAt_def OclIncludes_def OclLe_def OclSize_def plus_def One_ocl_int_def ss_lifting' localValid2sem notin_set_take notin_set_drop*)
done

lemma *size_of_insertAt*:
$\llbracket \tau \models \partial\ (x::('\tau,'\alpha::bot)\,VAL);\ \tau \models 1 \leq i;\ \tau \models i \leq (\|S\| + 1)\ \rrbracket \Longrightarrow$
$\|OclInsertAt\ S\ i\ x\|\ \tau = (\|(S::('\tau,'\alpha)Sequence)\| + 1)\ \tau$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValidDefined2sem DEF_def*)
apply(*case_tac i* $\tau \neq \perp$, *frule neq_commute*[*THEN iffD1*])
apply(*case_tac x* $\tau \neq \perp$, *rotate_tac* -1, *frule neq_commute*[*THEN iffD1*])
apply(*auto simp: OclInsertAt_def OclIncludes_def OclLe_def OclSize_def plus_def One_ocl_int_def ss_lifting' localValid2sem notin_set_take notin_set_drop, arith*)
done

at

defs

OclAt_def :
$OclAt \equiv lift2(strictify(\lambda X.\ strictify(\lambda x.\ if\ \ulcorner x \urcorner <= 0 \vee int(Nat.size \ulcorner Rep\_Sequence\_0\ X \urcorner) < \ulcorner x \urcorner\ then\ \perp\ else\ \ulcorner Rep\_Sequence\_0\ X \urcorner\ !\ nat\ (\ulcorner x \urcorner - 1))))$

ocl_setup_op [*OclAt*]

lemma *at_lower_bound*:
$\llbracket \tau \models i < 1\ \rrbracket \Longrightarrow OclAt\ (S::('\tau,'\alpha::bot)Sequence)\ i\ \tau = (\perp::('\tau,'\alpha)\,VAL)\ \tau$
by(*auto simp: localValid2sem OclLess_def OclAt_def One_ocl_int_def OclUndefined_def ss_lifting'*)

lemma *at_upper_bound*:
$\llbracket \tau \models \|S\| < i\ \rrbracket \Longrightarrow OclAt\ (S::('\tau,'\alpha::bot)Sequence)\ i\ \tau = (\perp::('\tau,'\alpha)\,VAL)\ \tau$
by(*auto simp: localValid2sem OclLess_def OclAt_def One_ocl_int_def OclSize_def OclUndefined_def ss_lifting'*)

lemma *OCL_is_defopt_OclAt*:
$\llbracket \tau \models 1 \leq i;\ \tau \models i \leq \|S\|\ \rrbracket \Longrightarrow$
$\tau \models \partial((OclAt\ (S::('\tau,'\alpha::bot)Sequence)\ i)::('\tau,'\alpha)\,VAL) = (\tau \models \partial\ S)$

```
  apply(case_tac i τ = ⊥)
  apply(simp add: localValid2sem OclLe_def One_ocl_int_def ss_lifting′)
  apply(rule_tac X=S in Sequence_sem_cases, simp_all add: localValid2sem)
  apply(auto simp: OclIsDefined_def OclAt_def OclLe_def
                   One_ocl_int_def OclSize_def ss_lifting′)
  apply(simp add: in_set_conv_nth)
  apply(erule_tac x=nat (⌜(i τ)⌝ − 1) in allE, arith)
  done

lemma at_of_mtSequence[simp]:
  OclAt ([]::(′τ,′α::bot)Sequence) i = (⊥::(′τ,′α) VAL)
  by(rule ext, auto simp: OclAt_def OclMtSequence_def
                          OclUndefined_def ss_lifting′)

lemma at_last_of_including:
  ⟦ τ ⊨ i ≐ (‖S‖+1) ⟧ ⟹
   OclAt ((S::(′τ,′α::bot)Sequence) ->including x) i τ =
   (x::(′τ,′α) VAL) τ
  apply(case_tac i τ = ⊥)
  apply(simp add: localValid2sem OclStrictEq_def ss_lifting′)
  apply(case_tac τ ⊨ ∂ x)
  apply(ocl_hypsubst, simp)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(frule neq_commute[THEN iffD1])
  apply(rotate_tac 2, frule neq_commute[THEN iffD1])
  apply(auto simp: OclAt_def OclIncluding_def OclSize_def localValid2sem
                   OclStrictEq_def plus_def One_ocl_int_def ss_lifting′)
  done

lemma at_inside_of_including:
  ⟦ τ ⊨ ∂ x; τ ⊨ i ≤ ‖S‖ ⟧ ⟹
   OclAt (S ->including (x::(′τ,′α) VAL)) i τ =
   ((OclAt (S::(′τ,′α::bot)Sequence) i τ)::′α)
  apply(case_tac i τ = ⊥)
  apply(simp add: localValid2sem OclLe_def ss_lifting′)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(frule neq_commute[THEN iffD1])
  apply(auto simp: OclAt_def OclIncluding_def OclSize_def localValid2sem
                   OclLe_def plus_def One_ocl_int_def ss_lifting′)
  apply(simp add: nth_append, arith)
  done

lemma at_of_union1:
  ⟦ τ ⊨ ∂ S2; τ ⊨ i ≤ ‖S1‖ ⟧ ⟹
```

```
  OclAt (S1 ∪ S2) i τ =
  ((OclAt (S1::('τ,'α::bot)Sequence) i τ)::'α)
 apply(rule_tac X=S1 in Sequence_sem_cases)
 apply(rule_tac X=S2 in Sequence_sem_cases)
 apply(simp_all add: localValidDefined2sem DEF_def)
 apply(auto simp: OclAt_def OclUnion_def OclSize_def localValid2sem
                  OclLe_def plus_def ss_lifting')
 apply(simp add: nth_append, arith)
 done

lemma at_of_union2:
 ⟦ τ ⊨ ‖S1‖ < i; τ ⊨ i ≤ (‖S1‖+‖S2‖) ⟧ ⟹
  OclAt (S1 ∪ S2) i τ =
  ((OclAt (S2::('τ,'α::bot)Sequence) (i−‖S1‖) τ)::'α)
 apply(rule_tac X=S1 in Sequence_sem_cases)
 apply(rule_tac X=S2 in Sequence_sem_cases)
 apply(simp_all add: localValidDefined2sem DEF_def)
 apply(auto simp: OclAt_def OclUnion_def OclSize_def minus_def localValid2sem
                  OclLe_def OclLess_def plus_def ss_lifting')
 apply(auto simp: nth_append, arith)
 apply(subgoal_tac ((nat (⌜(i τ)⌝ − 1)) − (length c)) = (nat ((⌜(i τ)⌝ − (int (length c))) −
1)))
 apply(simp, arith)
 done

lemma at_first_of_prepend:
 ⟦ τ ⊨ ∂ S ⟧ ⟹
  OclAt (OclPrepend (S::('τ,'α::bot)Sequence) x) 1 τ = (x::('τ,'α) VAL) τ

 apply(case_tac τ ⊨ ∂̸ x)
 apply(ocl_hypsubst, simp)
 apply(rule_tac X=S in Sequence_sem_cases)
 apply(simp_all add: localValidDefined2sem DEF_def)
 apply(frule neq_commute[THEN iffD1])
 apply(auto simp: OclAt_def OclIncluding_def OclUnion_def OclSize_def localValid2sem
                  OclMtSequence_def One_ocl_int_def ss_lifting')
 done

lemma at_inside_of_prepend:
 ⟦ τ ⊨ ∂ x; τ ⊨ 1 < i ⟧ ⟹
  OclAt (OclPrepend (S::('τ,'α::bot)Sequence) (x::('τ,'α) VAL)) i τ =
  ((OclAt S (i − 1) τ)::'α)
 apply(case_tac τ ⊨ ∂̸ S)
 apply(ocl_hypsubst, simp_all)
 apply(case_tac τ ⊨ ∂̸ i)
 apply(ocl_hypsubst, simp_all)


 apply(rule_tac X=S in Sequence_sem_cases)
```

```
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(frule neq_commute[THEN iffD1])
  apply(rotate_tac 2, frule neq_commute[THEN iffD1])
  apply(case_tac τ ⊨ ‖S‖ + 1 < i)
  apply(auto simp: OclAt_def OclIncluding_def OclUnion_def OclSize_def localValid2sem
                OclMtSequence_def OclLess_def minus_def plus_def
                One_ocl_int_def ss_lifting')
  apply(subgoal_tac ((x τ) # c) = ([x τ] @ c))
  apply(simp only: nth_append)
  apply(subgoal_tac ((nat (⌈⌈(i τ)⌉⌉ − 1)) − (Suc 0)) = (nat (−2 + ⌈⌈(i τ)⌉⌉)))
  apply(auto, arith, arith)
  done


lemma at_of_subsequence:
  ⟦ τ ⊨ 1 ≤ a; τ ⊨ a < b; τ ⊨ b ≤ ‖S‖;
    τ ⊨ 1 ≤ i; τ ⊨ i ≤ (a−b+1) ⟧ ⟹
   OclAt (OclSubSequence (S::('τ,'α::bot)Sequence) a b) i τ =
   ((OclAt S (a + i − 1) τ)::'α)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValid2sem)
  apply(case_tac i τ ≠ ⊥, frule neq_commute[THEN iffD1])
  apply(case_tac a τ ≠ ⊥, rotate_tac −1, frule neq_commute[THEN iffD1])
  apply(case_tac b τ ≠ ⊥, rotate_tac −1, frule neq_commute[THEN iffD1])
  apply(simp_all add: OclIncluding_def OclAt_def OclSubSequence_def
                  One_ocl_int_def plus_def minus_def ss_lifting'
                  OclLe_def OclLess_def notin_set_take notin_set_drop)
  done


lemma at_of_insertAt_before:
  ⟦ τ ⊨ ∂ (x::('τ,'α) VAL); τ ⊨ i ≤ (‖S‖ + 1); τ ⊨ j < i ⟧ ⟹
   OclAt (OclInsertAt (S::('τ,'α::bot)Sequence) i x) j τ  = ((OclAt S j τ)::'α)
  apply(case_tac τ ⊨ ∂̸ S, ocl_hypsubst, simp)
  apply(case_tac τ ⊨ ∂̸ i, ocl_hypsubst, simp)
  apply(case_tac τ ⊨ ∂̸ j, ocl_hypsubst, simp)
  apply(simp add: localValidDefined2sem DEF_def_both)
  apply(clarify, frule DEF_X_Sequence', clarify)
  apply(case_tac (1 ≤ ⌈⌈(j τ)⌉⌉))
  apply(subgoal_tac ((nat (⌈⌈(j τ)⌉⌉ − 1)) < (length c)))
  apply(auto simp: OclInsertAt_def OclAt_def OclLess_def OclSize_def plus_def
                One_ocl_int_def ss_lifting' localValid2sem
                OclLe_def notin_set_take notin_set_drop nth_append)
  apply(arith, arith)
  done

lemma at_of_insertAt:
  ⟦ τ ⊨ 1 ≤ i; τ ⊨ i ≤ (‖S‖ + 1) ⟧ ⟹
   OclAt (OclInsertAt (S::('τ,'α::bot)Sequence) i (x::('τ,'α) VAL)) i τ  = x τ
```

```
  apply(case_tac τ ⊨ ∂ S)
  apply(ocl_hypsubst, simp)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(case_tac i τ ≠ ⊥, frule neq_commute[THEN iffD1])
  apply(case_tac x τ ≠ ⊥, rotate_tac -1, frule neq_commute[THEN iffD1])
  apply(auto simp: OclInsertAt_def OclAt_def OclLe_def OclSize_def plus_def
                  One_ocl_int_def ss_lifting' localValid2sem
                  notin_set_take notin_set_drop, arith)
  apply(subgoal_tac (nat (⌜(i τ)⌝ - 1)) = length (take (nat (⌜(i τ)⌝ - 1)) c))
  apply(rule_tac t=(nat (⌜(i τ)⌝ - 1)) in subst) back
  apply(rule sym, assumption, rule nth_append_length)
  apply(simp, arith)
  done

lemma at_of_insertAt_after:
  ⟦ τ ⊨ ∂ (x::('τ,'α) VAL);  τ ⊨ 1 ≤ i; τ ⊨ i ≤ (‖S‖ + 1); τ ⊨ i < j ⟧ ⟹
   OclAt (OclInsertAt (S::('τ,'α::bot)Sequence) i x) j τ  = ((OclAt S (j - 1) τ)::'α)
  apply(case_tac τ ⊨ ∂ S, ocl_hypsubst, simp)
  apply(case_tac τ ⊨ ∂ i, ocl_hypsubst, simp)
  apply(case_tac τ ⊨ ∂ j, ocl_hypsubst, simp)
  apply(simp add: localValidDefined2sem DEF_def_both)
  apply(clarify, frule DEF_X_Sequence', clarify)
  apply(subgoal_tac (min (length c) (nat (⌜(i τ)⌝ - 1))) = nat (⌜(i τ)⌝ - 1))
  apply(case_tac (((int (length c)) + 1) < ⌜(j τ)⌝))
  apply(auto simp: OclInsertAt_def OclAt_def OclLess_def OclSize_def plus_def
                  One_ocl_int_def ss_lifting' localValid2sem minus_def
                  OclLe_def notin_set_take notin_set_drop nth_append)
  apply(thin_tac ?P ≠ ?Q | thin_tac ?P = ?Q)+ apply(arith)
  apply(thin_tac ?P ≠ ?Q | thin_tac ?P = ?Q)+ apply(arith)
  apply(thin_tac ?P ≠ ?Q | thin_tac ?P = ?Q | thin_tac ?P ∉ ?Q)+
  apply(subgoal_tac (nat (⌜(j τ)⌝ - 1)) - (nat (⌜(i τ)⌝ - 1)) = Suc ((nat (⌜(j τ)⌝ - 2)) -
(nat (⌜(i τ)⌝ - 1))))
  apply(simp)
  apply(subst nth_drop)
  apply(thin_tac ?P = ?Q, arith)
  apply(subgoal_tac ((nat (⌜(i τ)⌝ - 1)) + ((nat (⌜(j τ)⌝ - 2)) - (nat (⌜(i τ)⌝ - 1)))) =
(nat (-2 + ⌜(j τ)⌝)))
  apply(simp)
  apply(rotate_tac 2, thin_tac ?P, thin_tac ?P, thin_tac ?P, arith)
  apply(rotate_tac 2, thin_tac ?P, thin_tac ?P, arith)
  apply((thin_tac ?P ≠ ?Q | thin_tac ?P = ?Q | thin_tac ?P ∉ ?Q)+)
  apply(rotate_tac 2, thin_tac ?P, thin_tac ?P, arith)
  done
```

IndexOf

defs

```
OclIndexOf_def :
OclIndexOf ≡ lift2(strictify(λX. strictify(λx. if x ∈ set ⌜Rep_Sequence_0 X⌝
                          then ⌞int(length (takeWhile (λ y. y ≠ x) ⌜Rep_Sequence_0
X⌝) + 1)⌟
                          else ⊥)))

ocl_setup_op [OclIndexOf]

lemma indexOf_excludes[simp]:
  ⟦ τ ⊨ ¬ ((x::('τ,'α) VAL) ∈ (S::('τ,'α::bot)Sequence)) ⟧ ⟹
   OclIndexOf S x τ = ⊥ τ
  apply(rule_tac X=S in Sequence_sem_cases, simp_all)
  apply(case_tac x τ = ⊥)
  apply(simp_all add: OclIndexOf_def OclIncludes_def OclNot_def o_def
                OclUndefined_def ss_lifting' localValid2sem)
  done

lemma notin_conv_all_noteq: a ∉ S = (∀ x ∈ S. x ≠ a)
  by (auto)

lemma in_set_conv_first_decomp:
  (x ∈ set xs) = (∃ ys zs. xs = ys @ x # zs ∧ x ∉ set ys)
  apply(auto, erule rev_mp)
  apply(rule length_induct, auto)
  apply(frule in_set_conv_decomp[THEN iffD1], clarify)
  apply(case_tac x ∈ set ys)
  apply(erule_tac x=ys in allE, auto)
  apply(rule_tac x=ysa in exI, simp)
  done
```

In the standard `self->at(i) = obj` is written where it should be written like `self->at(result) = obj`. Furthermore the standard doesn't specify that the returned index is the index of the first object that is equal to the searched one. This disables the combined use of *indexOf* and *subsequence* to visit all occurrences of a specific object.

Typo in Requirement 11.7.5.10

No specification of the returned index of `indexOf`

```
lemma at_indexOf:
  ⟦ τ ⊨ (x::('τ,'α) VAL) ∈ (S::('τ,'α::bot)Sequence) ⟧ ⟹
   OclAt S (OclIndexOf S x) τ = x τ
  apply(rule_tac X=S in Sequence_sem_cases, simp_all)
  apply(case_tac x τ ≠ ⊥)
  apply(frule neq_commute[THEN iffD1])
  apply(simp_all add: OclIndexOf_def OclIncludes_def OclNot_def OclAt_def
                  OclUndefined_def ss_lifting' localValid2sem
                  o_def)
  apply(frule in_set_conv_first_decomp[THEN iffD1])
  apply(clarify, simp add: notin_conv_all_noteq takeWhile_tail)
  done
```

First and Last

defs

$OclFirst\_def$:
$OclFirst \quad \equiv lift1\ (strictify\ (\lambda S.\ if\ \ulcorner Rep\_Sequence\_0\ S \urcorner = []\ then\ \bot$
$else\ hd\ (\ulcorner Rep\_Sequence\_0\ S \urcorner)))$

$OclLast\_def$:
$OclLast \quad \equiv lift1\ (strictify\ (\lambda S.\ if\ \ulcorner Rep\_Sequence\_0\ S \urcorner = []\ then\ \bot$
$else\ List.last\ (\ulcorner Rep\_Sequence\_0\ S \urcorner)))$

ocl_setup_op [*OclFirst*, *OclLast*]

lemma *first_at_UC*:
$OclFirst\ (S::('\tau,'\alpha::bot)Sequence) = ((OclAt\ S\ 1)::('\tau,'\alpha)\ VAL)$
apply(*rule Sequence_sem_cases_ext, simp_all*)
apply(*auto simp*: *OclFirst_def OclAt_def Zero_ocl_int_def One_ocl_int_def ss_lifting′ neq_Nil_conv*)
done

lemma *first_of_mtSequence*[*simp*]:
$OclFirst\ ([]::('\tau,'\alpha::bot)Sequence) = (\bot::('\tau,'\alpha)\ VAL)$
by(*simp add*: *first_at_UC*)

lemma *first_empty_of_including*:
$[\![\ \tau \models (S \doteq [])]\!] \Longrightarrow$
$OclFirst\ ((S::('\tau,'\alpha::bot)Sequence)\ \texttt{->including}\ (x::('\tau,'\alpha)\ VAL))\ \tau =$
$((x\ \tau)::'\alpha)$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add*: *localValid2sem*)
apply(*case_tac c*)
apply(*auto simp*: *OclIncluding_def OclFirst_def OclNot_def OclStrictEq_def OclMtSequence_def ss_lifting′ neq_commute*)
done

Note: strange writing because of normal form of emptiness test

lemma *first_not_empty_of_including*:
$[\![\ \tau \models \partial\ x;\ \tau \models \neg\ (S \doteq [])]\!] \Longrightarrow$
$OclFirst\ ((S::('\tau,'\alpha::bot)Sequence)\ \texttt{->including}\ (x::('\tau,'\alpha)\ VAL))\ \tau =$
$((OclFirst\ S\ \tau)::'\alpha)$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add*: *localValidDefined2sem DEF_def*)
apply(*case_tac c*)
apply(*auto simp*: *OclIncluding_def OclFirst_def OclNot_def OclStrictEq_def OclMtSequence_def ss_lifting′ neq_commute localValid2sem*)
done

lemma *last_at_UC*:
$OclLast\ (S::('\tau,'\alpha::bot)Sequence) = ((OclAt\ S\ \|S\|)::('\tau,'\alpha)\ VAL)$
apply(*rule Sequence_sem_cases_ext, simp_all*)
apply(*auto simp: OclLast_def OclAt_def OclSize_def*
Zero_ocl_int_def One_ocl_int_def
ss_lifting' last_conv_nth neq_Nil_conv)
done

lemma *last_of_mtSequence*[*simp*]:
$OclLast\ ([]::('\tau,'\alpha::bot)Sequence) = (\bot::('\tau,'\alpha)\ VAL)$
by(*simp add: last_at_UC*)

lemma *last_empty_of_including*:
$[\![\ \tau \models \partial\ S]\!] \Longrightarrow$
$OclLast\ ((S::('\tau,'\alpha::bot)Sequence)$ `->including` $(x::('\tau,'\alpha)\ VAL))\ \tau =$
$((x\ \tau)::'\alpha)$
apply(*rule_tac X=S* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValidDefined2sem DEF_def*)
apply(*simp add: OclIncluding_def OclLast_def*
ss_lifting' neq_commute)
done

Forall

defs

$OclForAll\_def$: $OclForAll \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify(\ \lambda\ S.\ (\lambda\ P.$
$if\ \ \forall x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ \ P\ x = \llcorner True\lrcorner$
$then\ \llcorner True\lrcorner$
$else\ if\ \exists x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ P\ x = \llcorner False\lrcorner$
$then\ \llcorner False\lrcorner$
$else\ \bot)))$

ocl_setup_op [*OclForAll*]

Its computation characterisation by iterate

lemma *forall_by_iterate*:
$(\forall\ x \in (S::('\tau,'\alpha::bot)Sequence)\ \bullet\ (P::('\tau,'\alpha::bot)\ VAL \Rightarrow\ '\tau\ Boolean)\ x) =$
$(S$ `->iterate`$(x;y= \top \mid (P\ x) \wedge y))$
apply(*rule ext*)
apply(*rule iterate_universal_0, simp_all*)
apply(*simp add: cp_by_cpify OclForAll_def lift0_def*)
apply(*simp_all add: OclForAll_def OclUndefined_def OclMtSequence_def*

```
                      OclTrue_def localValidDefined2sem ss_lifting')
  apply(frule DEF_X_Sequence', clarify)
  apply(frule neq_commute[THEN iffD1])
  apply(rotate_tac 1, frule neq_commute[THEN iffD1])
  apply(rule_tac x=P (λs. (xa x)) x in boolean_cases_sem)
  apply(simp_all add: OclIncluding_def OclAnd_def ss_lifting')
  done
```

Derived properties from the relation to iterate

```
lemma OCL_undef_2_OclForAll[simp]:
  ⟦ τ ⊨ ¬ ((S::('τ,'α::bot)Sequence) ≐ []) ⟧ ⟹
    (∀ x ∈ S • (λ x::('τ,'α) VAL. ⊥) x) τ = ⊥ τ
  apply(rule_tac X=S in Sequence_sem_cases, simp_all)
  apply(simp_all add: OclForAll_def OclUndefined_def OclMtSequence_def
                      OclStrictEq_def OclNot_def localValid2sem
                      ss_lifting' o_def Abs_Sequence_0_inject_charn)
  done
```

```
lemmas forall_of_mtSequence[simp] = f1_by_iterate_mtSequence[OF forall_by_iterate]

lemma forall_including [simp]:
  ⟦τ ⊨ ∂ (X::('τ,'α::bot)Sequence); τ ⊨ ∂ (a::('τ,'α) VAL); cp P ⟧ ⟹
   (∀ x ∈ (X->including a) • P (x::('τ,'α) VAL)) τ =  ((P a) ∧ (∀ x ∈ X • P x)) τ
  by(simp add: forall_by_iterate iterate_of_including)
```

```
thm f1_by_iterate_distrib_union[OF forall_by_iterate, of OclAnd, simplified]

lemma forall_union [simp]:
  ⟦τ ⊨ ∂ (X::('τ,'α::bot)Sequence); τ ⊨ ∂ Y ⟧ ⟹
   (∀ x ∈ (X ∪ Y) • P (x::('τ,'α) VAL)) τ =  ((∀ x ∈ X • P x) ∧ (∀ x ∈ Y • P x)) τ
  apply(simp add: forall_by_iterate)
  apply(rule iterate_distrib_union)
  apply(simp_all add: OCL_logic_ACI)
  done
```

Introduction rules for ForAll

```
lemma ForAllI:
  assumes defS : τ ⊨ ∂(S::('τ,('α::bot))Sequence)
  and     allP : ⋀ x. τ ⊨ x ∈ S ⟹ τ ⊨ P x
  shows          τ ⊨ (∀ x ∈ S • P(x::'τ ⇒ 'α::bot))
  apply(insert defS)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(subgoal_tac (∀x ∈ set c. P (λτ. x) τ = ⌞True⌟))
  apply(auto simp: OclForAll_def localValid2sem ss_lifting')
```

```
  apply(rule_tac τ1 =τ in localValid2sem[THEN iffD1], rule allP)
  apply(auto simp: localValid2sem OclIncludes_def ss_lifting')
  done


lemma  notForAllI:
  assumes isin : τ ⊨ x ∈ (S::('τ,('α::bot))Sequence)
  and     not  : τ ⊨ ¬ (P x)
  and     cpP  : cp P
  shows        τ ⊨ ¬ (∀ x ∈ S • P(x::'τ ⇒ 'α::bot))
  apply(insert isin not cpP)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(subgoal_tac (P x τ) = (P (λs. (x τ)) τ))
  apply(case_tac (P x) τ ≠ ⊥, frule neq_commute[THEN iffD1])
  apply(case_tac x τ ≠ ⊥, rotate_tac -1, frule neq_commute[THEN iffD1])
  apply(subgoal_tac (∃ x∈(set c). ((P (λs. x) τ) = ⌞False⌟)))
  apply(simp_all add: OclForAll_def OclIncludes_def OclNot_def
                   ss_lifting' localValid2sem o_def)
  apply(rule_tac x=x τ in bexI)
  apply(rule_tac x=P (λs. (x τ)) τ in boolean_cases_sem)
  apply(simp_all)
  apply(rule_tac x=x τ in bexI)
  apply(rule_tac x=P (λs. (x τ)) τ in boolean_cases_sem)
  apply(simp_all)
  apply(rule trans, rule_tac x=x in cp_subst)
  apply(simp_all add: lift0_def)
  done


lemma undefForAllI2:
  assumes exUndef1: τ ⊨ x ∈ (S::('τ,('α::bot))Sequence)
  and     exUndef2: τ ⊨  ∂ (P x)
  and     allP     : ⋀ x. τ ⊨ x ∈ S ⟹ τ ⊨ (P x) ∨ ∂(P x)
  and     cpP      : cp P
  shows          τ ⊨ ∂(∀ x ∈ S • P(x::'τ ⇒ 'α::bot))
  apply(insert exUndef1 exUndef2)
  apply(frule isDefined_if_valid)
  apply(drule OCL_is_defopt_OclIncludes[THEN iffD1])
  apply(simp add: OclLocalValid_def OclIsDefined_def OclTrue_def
               OclForAll_def OclNot_def OclIncludes_def
               ss_lifting')
  apply(erule conjE)
  apply(frule DEF_X_Sequence', clarify)
  apply(simp, safe)
  apply(rule_tac x = xa in bexI, simp_all) prefer 2
  apply(rule_tac x = x τ in bexI, simp_all)
  apply(subgoal_tac P (λτ a. x τ) τ = P x τ, simp)
  apply(rule_tac τ = τ and P = P in cp_charn, simp add: cpP, rule cpP)
```

apply(*erule_tac Q* = *P* ($\lambda\tau$. *xa*) τ = $\llcorner$*False*$\lrcorner$ **in** *contrapos_pp*)
apply(*rule_tac* τ*1* = τ **in** *localValidToNotFalse2sem* [*THEN iffD1*]) **back**
apply(*rule allP*)
apply(*simp add*: *includes_charn2 DEF_def*)
done

lemma *isdefForAllI2*:
assumes *exNot1*: $\tau \models x \in (S$::('τ,('α::*bot*))*Sequence*)
and *exNot2*: $\tau \models \neg\ (P\ x)$
and *cpP* : *cp P*
shows $\tau \models \partial(\forall\ x \in S\ .\ P(x$::'$\tau \Rightarrow$ 'α::*bot*))
apply(*insert exNot1 exNot2 cpP*)
apply(*drule_tac P*=*P* **in** *notForAllI*, *simp_all add*: *isDefined_if_invalid*)
done

lemmas *isdefForAllI3* = *ForAllI*[*THEN isDefined_if_valid*]

Exists

Alternative definition for the use with ocl_setup_op

lemma *OclExists_alt_def*:
OclExists ≡ (*lift2*′ *lift_arg0 lift_arg1*) (*strictify*(λ *S*. (λ *P*.
if ∃ *x* ∈ *set* ⌜*Rep_Sequence_0 S*⌝. *P x* = $\llcorner$*True*$\lrcorner$
then $\llcorner$*True*$\lrcorner$
else if ∀ *x* ∈ *set* ⌜*Rep_Sequence_0 S*⌝. *P x* = $\llcorner$*False*$\lrcorner$
then $\llcorner$*False*$\lrcorner$
else ⊥)))
apply(*rule eq_reflection*, (*rule ext*)+)
apply(*simp add*: *OclExists_def OclForAll_def OclNot_def ss_lifting*′)
apply(*auto*)
oops

The relation to iterate

lemma *exists_by_iterate*:
(∃ *x* ∈ (*source*::('τ,'α::*bot*)*Sequence*) . (*P*::('τ,'α::*bot*) *VAL* ⇒ 'τ *Boolean*) *x*) =
(*source* ->`iterate`(*x*;*y*= `F` | (*P x*) ∨ *y*))
apply(*simp add*: *OclExists_def forall_by_iterate*)
apply(*rule ext*, *rule iterate_fusion_0*, *simp_all*)

apply(*simp add*: *OclNot_def OclOr_def OclAnd_def DEF_def*
lift0_def lift1_def lift2_def strictify_def)
done

Derived properties from the relation to iterate

lemma *OCL_cp_OclExists* [*simp*,*intro*!]:
⟦ ⋀ *x*. *cp* (*P x*); *cp S* ⟧

$\Longrightarrow cp(\lambda X.\ \exists\ Y \in ((S\ X)::('\tau,'\alpha::bot)Sequence)\ .\ P\ (Y::('\tau,'\alpha)\ VAL)\ X)$
by(*simp add*: *exists_by_iterate*)

lemmas *exists_undef* [*simp*] = *f1_by_iterate_undef_1*[*OF exists_by_iterate*]
lemmas *exists_empty* [*simp*] = *f1_by_iterate_mtSequence*[*OF exists_by_iterate*]
thm *f1_by_iterate_including*[*OF exists_by_iterate, simplified*]
thm *f1_by_iterate_opt_including*[*OF exists_by_iterate, simplified*]

thm *f1_by_iterate_undef_1*[*OF exists_by_iterate*]
thm *f1_by_iterate_mtSequence*[*OF exists_by_iterate*]
thm *f1_by_iterate_including*[*OF exists_by_iterate, simplified*]
thm *f1_by_iterate_opt_including*[*OF exists_by_iterate, simplified*]

Introduction rules for exist

lemma *ExistsI*:
assumes *isin* : $\tau \models x \in (S::('\tau,('\alpha::bot))Sequence)$
and *not* : $\tau \models (P\ x)$
and *cpP* : *cp P*
shows $\tau \models (\exists\ x \in S\ .\ P(x::'\tau \Rightarrow '\alpha::bot))$
by (*auto simp*:*OclExists_def intro*!: *notForAllI isin not cpP*)

lemma *NotExistsI*:
assumes *defS* : $\tau \models \partial(S::('\tau,('\alpha::bot))Sequence)$
and *allP* : $\bigwedge x.\ \tau \models x \in S \Longrightarrow \tau \models\ \neg\ P\ x$
shows $\tau \models\ \neg(\exists\ x \in S\ .\ P(x::'\tau \Rightarrow '\alpha::bot))$
by (*auto simp*: *OclExists_def intro*!: *ForAllI defS allP*)

IncludesAll

defs

$OclIncludesAll\_def$: $OclIncludesAll \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.$
$\llcorner set \ulcorner Rep\_Sequence\_0\ Y\urcorner \subseteq set \ulcorner Rep\_Sequence\_0\ X\urcorner\lrcorner)))$

ocl_setup_op [*OclIncludesAll*]

Its computational characterisation by iterate

This formulation should be replaced by one with a strict body to simplify reasoning over it.

lemma *includesAll_by_iterate*:
$OclIncludesAll\ X\ (Y::('\tau,'\alpha::bot)Sequence) =$
$(Y$ ->**iterate**$(x;y=\ if\ (\partial\ X)\ then\ \top\ else\ \bot\ endif \mid ((x::('\tau,'\alpha)\ VAL) \in X)\ \dot{\wedge}\ y))$
apply(*rule ext*)

apply(*rule iterate_universal_0, simp_all*)
apply(*simp add: OclIncludesAll_def OclMtSequence_def OclIsDefined_def OclIf_def OclTrue_def OclUndefined_def ss_lifting'*)
apply(*simp add: localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Sequence', clarify*)
apply(*rotate_tac 1, frule neq_commute[THEN iffD1]*)
apply(*simp add: OclIncludesAll_def OclIncluding_def OclIncludes_def OclSand_def ss_lifting'*)
done

thm *f1_by_iterate_undef_1[OF includesAll_by_iterate]*
lemmas *includesAll_empty [simp] = f1_by_iterate_mtSequence[OF includesAll_by_iterate]*
lemmas *includesAll_including [simp] = f1_by_iterate_opt_including[OF includesAll_by_iterate, simplified]*

thm *includesAll_empty includesAll_including*

lemma *includesAll_union [simp]*:
$(OclIncludesAll\ Z\ ((X::('\tau,'\alpha::bot)Sequence) \cup Y)) =$
$(OclIncludesAll\ Z\ X)\ \dot{\wedge}\ (OclIncludesAll\ Z\ Y)$
apply(*rule ext*)
apply(*case_tac x* $\models \not\partial$ *X , ocl_hypsubst, simp*)
apply(*case_tac x* $\models \not\partial$ *Y, ocl_hypsubst, simp*)
apply(*case_tac x* $\models \not\partial$ *Z, ocl_hypsubst, simp*)
apply(*subst f1_by_iterate_distrib_union[OF includesAll_by_iterate]*)
apply(*simp_all*)
apply(*ocl_subst, simp*)

apply(*auto simp: OclIncludesAll_def OclSand_def OclIncludes_def OclTrue_def localValidDefined2sem ss_lifting'*)
done

lemma *includesAll_by_forall*:
$[\![\ \tau \models \partial(X::('\tau,'\alpha::bot)Sequence)\ ]\!] \Longrightarrow$
$(OclIncludesAll\ X\ Y)\ \tau = (\forall\ a \in\ Y\ .\ ((a::('\tau,'\alpha)\ VAL) \in X))\ \tau$
apply(*rule_tac X=X* **in** *Sequence_sem_cases*)
apply(*rule_tac X=Y* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValidDefined2sem DEF_def*)
apply(*auto simp: OCL_Sequence.OclForAll_def OCL_Sequence.OclIncludesAll_def OclStrongEq_def OCL_Sequence.OclIncludes_def ss_lifting' localValid2sem*)
done

ExcludesAll

defs

OclExcludesAll_def:
$OclExcludesAll \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.$

$\llcorner set \ulcorner Rep\_Sequence\_0\ X \urcorner \cap set \ulcorner Rep\_Sequence\_0\ Y \urcorner = \{\}\lrcorner)))$

ocl_setup_op [*OclExcludesAll*]

Its computational characterisation by iterate

This formulation should be replaced by one with a strict body to simplify reasoning over it.

```
lemma excludesAll_by_iterate:
  OclExcludesAll X (Y::('τ,'α::bot)Sequence) =
  (Y->iterate(x;y= if (∂ X) then ⊤ else ⊥ endif | ((x::('τ,'α)VAL) ∉ X) ∧ y))
  apply(rule ext)
  apply(rule iterate_universal_0, simp_all)
  apply(simp add: OclExcludesAll_def OclMtSequence_def OclIsDefined_def
                  OclIf_def OclTrue_def OclUndefined_def ss_lifting')
  apply(simp add: localValidDefined2sem DEF_def)
  apply(frule DEF_X_Sequence', clarify)
  apply(rotate_tac 1, frule neq_commute[THEN iffD1])
  apply(simp add: OclExcludesAll_def OclIncluding_def OclIncludes_def
                  OclSand_def OclNot_def ss_lifting')
  done


thm f1_by_iterate_undef_1[OF excludesAll_by_iterate]
lemmas excludesAll_empty [simp] = f1_by_iterate_mtSequence[OF excludesAll_by_iterate]
lemmas excludesAll_including [simp] = f1_by_iterate_opt_including[OF excludesAll_by_iterate,
simplified]

thm excludesAll_empty excludesAll_including

lemma excludesAll_union [simp]:
  (OclExcludesAll Z ((X::('τ,'α::bot)Sequence) ∪ Y)) =
  (OclExcludesAll Z X) ∧ (OclExcludesAll Z Y)
  apply(rule ext)
  apply(case_tac x ⊨ ∂̸ X , ocl_hypsubst, simp)
  apply(case_tac x ⊨ ∂̸ Y, ocl_hypsubst, simp)
  apply(case_tac x ⊨ ∂̸ Z, ocl_hypsubst, simp)
  apply(subst f1_by_iterate_distrib_union[OF excludesAll_by_iterate])
  apply(simp_all)
  apply(ocl_subst, simp)

  apply(auto simp: OclIncludesAll_def OclSand_def OclIncludes_def
                   OclTrue_def localValidDefined2sem ss_lifting')
  done

lemma excludesAll_by_forall:
  ⟦ τ ⊨ ∂(X::('τ,'α::bot)Sequence) ⟧ ⟹
```

$(OclExcludesAll\ X\ Y)\ \tau = (\forall\ a \in\ Y .\ ((a{::}('\tau,'\alpha)\,VAL) \notin X))\ \tau$
apply(*rule_tac X=X* **in** *Sequence_sem_cases*)
apply(*rule_tac X=Y* **in** *Sequence_sem_cases*)
apply(*simp_all add: localValidDefined2sem DEF_def*)
apply(*auto simp: OCL_Sequence.OclForAll_def OCL_Sequence.OclExcludesAll_def OclStrongEq_def*
OCL_Sequence.OclIncludes_def ss_lifting′ localValid2sem OclNot_def)
done

Select

defs

$$
\begin{array}{ll}
OclSelect\_def\!: & OclSelect \equiv\ (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.\\
& if\quad \forall\ x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ DEF\ (P\ x)\\
& then\ \ Abs\_Sequence\_0(\llcorner List.filter\ (\lambda\ x.\ P\ x = \llcorner True\lrcorner)\\
& \qquad\qquad\qquad\qquad\qquad \ulcorner Rep\_Sequence\_0\ S\urcorner\ \lrcorner)\\
& else\ \ \bot))
\end{array}
$$

ocl_setup_op [*OclSelect*]

Its computational characterisation by iterate

lemma *select_by_iterate*:
$((\!|x : (source{::}('\tau,'\alpha{::}bot)Sequence)\ |\ ((P{::}('\tau,'\alpha{::}bot)\,VAL \Rightarrow\ '\tau\ Boolean)\ x)|\!)) =$
$(source$ ->**iterate**$(iterator;\ result = []\ |$
$if\ \ (P\ iterator)\ then\ (OclIncluding\ result\ iterator)\ else\ (result)\ endif\ ))$
apply(*rule ext, rule iterate_universal_0, simp_all*)
apply(*simp add: OclSelect_def cp_by_cpify lift0_def*)
apply(*simp_all add: OclUndefined_def OclMtSequence_def OclSelect_def*
localValidDefined2sem ss_lifting′)
apply(*frule DEF_X_Sequence′, clarify*)
apply(*frule neq_commute*[*THEN iffD1*])
apply(*rotate_tac 1, frule neq_commute*[*THEN iffD1*])
apply(*rule_tac x=P* $(\lambda s.\ (xa\ x))\ x$ **in** *boolean_cases_sem*)
apply(*simp_all add: OclIf_def OclIncluding_def ss_lifting′*)
done

Properties derived from the correspondence to iterate

lemmas *select_of_mtSequence* [*simp*] = *f1_by_iterate_mtSequence*[*OF select_by_iterate*]

lemma *select_including* [*simp*]:
$[\![\ \tau \vDash \partial\ (x{::}('\tau,'\alpha{::}bot)\,VAL);\ cp\ P\ ]\!] \Longrightarrow$
$OclSelect\ (S{::}('\tau,'\alpha)Sequence$ ->**including** $x)\ P\ \tau =$
$(if\ (P\ x)\ then\ (OclSelect\ S\ P)$ ->**including** $x\ else\ (OclSelect\ S\ P) endif)\ \tau$
apply(*case_tac* $\tau \vDash \emptyset$ *S, ocl_hypsubst, simp*)
apply(*simp add: f1_by_iterate_including*[*OF select_by_iterate, simplified*])
done

lemma *select_including_strict* [*simp*]:

```
  ⟦ P ⊥ = ⊥; cp P ⟧ ⟹
  OclSelect (S::('τ,'α)Sequence ->including (x::('τ,'α::bot) VAL)) P τ =
  (if (P x) then (OclSelect S P) ->including x else (OclSelect S P) endif) τ
  by(simp add: f1_by_iterate_opt_including[OF select_by_iterate, simplified])

lemma select_union [simp]:
  (OclSelect ((X::('τ,'α::bot)Sequence) ∪ Y) (P::('τ,'α) VAL ⇒ 'τ Boolean)) =
   (OclSelect X P) ∪ (OclSelect Y P)
  apply(subst f1_by_iterate_distrib_union_opt[OF select_by_iterate])
  apply(simp_all)
  apply(rule_tac X=B in Sequence_sem_cases_ext, simp_all)
  apply(rule_tac X=y in Sequence_sem_cases, simp_all)
  apply(case_tac x τ ≠ ⊥, drule neq_commute[THEN iffD1])
  apply(simp_all add: OclIf_def OclUnion_def OclIncluding_def ss_lifting')
  done

lemma iterate_of_select:
  assumes undef_1_Q: ⋀ y. Q ⊥ y = ⊥
  and      undef_2_Q: ⋀ x. Q x ⊥  = ⊥
  and          cp1_Q: ⋀ y. cp (λx. Q x y)
  and          cp2_Q: ⋀ x. cp (Q x)
  and      undef_1_P: P ⊥ = ⊥
  and           cp_P: cp P
  shows OclIterate (OclSelect (S::('τ,'α::bot)Sequence) P) Q A =
         OclIterate S (λ (x::('τ,'α) VAL) y. if (P x) then Q x y else y endif) A
  apply(insert cp1_Q cp2_Q cp_P undef_1_P undef_2_Q undef_1_Q)
  apply(rule ext)
  apply(rule iterate_universal, simp_all add: select_by_iterate)
  apply(subst iterate_opt_of_including, simp_all)
  apply(rule_tac P=λ t. t x = ?Q in subst)
  apply(rule sym[OF if_distrib_strict_alt])
  apply(simp_all add: iterate_opt_of_including)
  done

lemma select_of_select:
  assumes undef_1_P: P ⊥ = ⊥
  and      undef_1_Q: Q ⊥ = ⊥
  and           cp_P: cp P
  and           cp_Q: cp Q
 shows OclSelect (OclSelect (S::('τ,'α::bot)Sequence) (P::('τ,'α) VAL ⇒ 'τ Boolean)) (Q::('τ,'α) VAL
⇒ 'τ Boolean) =
         OclSelect S (λ x. (P x) ∧̇ ((P x) ⟶ (Q x)))
  by(simp add: select_by_iterate iterate_of_select[simplified select_by_iterate]
               undef_1_Q undef_1_P cp_P cp_Q if_contract)
```

Reject

defs

$OclReject\_def$: $OclReject \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
$if\ \ \forall\ x \in set\ \ulcorner Rep\_Sequence\_0\ S \urcorner.\ DEF\ (P\ x)$
$then\ \ Abs\_Sequence\_0(\llcorner List.filter\ (\lambda\ x.\ P\ x = \llcorner False \lrcorner)$
$\ulcorner Rep\_Sequence\_0\ S \urcorner \lrcorner)$
$else\ \ \bot))$

ocl_setup_op [*OclReject*]

lemma *reject_by_select*:
$OclReject\ S\ P = (\!|x : (S::('\tau,'\alpha::bot)Sequence) \mid (\neg\ (P::('\tau,'\alpha::bot)VAL \Rightarrow '\tau\ Boolean)\ x)|\!))$
apply(*rule ext, auto simp: OclReject_def OclNot_def OclSelect_def ss_lifting'*)
apply($subgoal\_tac\ (\lambda xa.\ ((P\ (\lambda s.\ xa)\ x) = \llcorner False \lrcorner)) =$
$(\lambda xa.\ (((P\ (\lambda s.\ xa)\ x) \neq down) \wedge (((P\ (\lambda s.\ xa)\ x) \neq down) \longrightarrow (\neg \ulcorner(P\ (\lambda s.$
$xa)\ x)\urcorner)))))$
apply(*simp, rule ext*)
apply(*rule_tac x=P* $(\lambda s.\ xa)$ *x* **in** *boolean_cases_sem, simp_all*)
done

lemma *reject_by_iterate*:
$OclReject\ (source::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha::bot)VAL \Rightarrow '\tau\ Boolean) =$
$(source$ `->iterate`$(iterator;\ result = [\,] \mid$
$if\ \ (\neg\ P\ iterator)\ then\ (OclIncluding\ result\ iterator)\ else\ (result)\ endif\ ))$
by(*simp add: reject_by_select select_by_iterate*)

Properties derived from the correspondence to iterate

lemmas *reject_of_mtSequence* [*simp*] = *f1_by_iterate_mtSequence*[*OF reject_by_iterate*]

lemma *reject_of_union*:
$(OclReject\ ((X::('\tau,'\alpha::bot)Sequence) \cup Y)\ (P::('\tau,'\alpha)VAL \Rightarrow '\tau\ Boolean)) =$
$(OclReject\ X\ P) \cup (OclReject\ Y\ P)$
by(*simp add: reject_by_select*)

Excluding

defs

$OclExcluding\_def$:
$OclExcluding \equiv lift2(strictify(\lambda X.\ strictify\ (\lambda x.\ Abs\_Sequence\_0$
$(\llcorner List.filter\ (\lambda\ e.\ x \neq e)\ \ulcorner Rep\_Sequence\_0\ X \urcorner \lrcorner))))$

ocl_setup_op [*OclExcluding*]

Its computational characteristic by iterate

lemma *excluding_by_iterate*:
$(OclExcluding\ (S::('\tau,'\alpha::bot)Sequence)\ (x::('\tau,'\alpha)VAL)) =$
$OclIterate\ S\ (\lambda\ (a::('\tau,'\alpha)VAL)\ y.\ if\ (a \doteq x)\ then\ y\ else\ y$ `->including` $a\ endif)$
$(if\ (\partial\ x)\ then\ [\,]\ else\ \bot\ endif)$

apply(*rule ext*)
apply(*rule iterate_universal, simp_all add: localValidDefined2sem DEF_def*)
apply(*simp add: OclMtSequence_def OclExcluding_def ss_lifting′ OclIf_def OclIsDefined_def OclUndefined_def*)
apply(*case_tac xa* $\models$ $\emptyset$ *x*)
apply(*ocl_hypsubst, simp_all add: localValidDefined2sem*)
apply(*frule DEF_X_Sequence′, clarify*)
apply(*frule neq_commute*[*THEN iffD1*])
apply(*rotate_tac 1, frule neq_commute*[*THEN iffD1*])
apply(*simp add: OclExcluding_def OclIterate_def OclIf_def OclIncluding_def OclStrictEq_def ss_lifting′*)
done

lemmas *excluding_empty* [*simp*] = *f1_by_iterate_mtSequence*[*OF excluding_by_iterate*]
lemmas *excluding_including* [*simp*] = *f1_by_iterate_opt_including*[*OF excluding_by_iterate, simplified*]

lemma *excluding_union* [*simp*]:
$(((X::('\tau,'\alpha::bot)Sequence) \cup Y)$ `->excluding` $x) =$
$(X$ `->excluding` $x) \cup (Y$ `->excluding` $(x::('\tau,'\alpha)VAL))$
apply(*rule ext, case_tac xa* $\models$ $\emptyset$ *x, ocl_hypsubst, simp*)
apply(*case_tac xa* $\models$ $\emptyset$ *X, ocl_hypsubst, simp*)
apply(*case_tac xa* $\models$ $\emptyset$ *Y, ocl_hypsubst, simp*)
apply(*simp add: localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Sequence′, clarify*)
apply(*rotate_tac 2, frule DEF_X_Sequence′, clarify*)
apply(*simp add: OclExcluding_def OclUnion_def ss_lifting′*)
done

thm *excluding_empty excluding_including excluding_union*

Further propierties of excluding

lemma *includes_of_excluding*:
$[\![\ \tau \models \partial(x::('\tau,'\alpha)VAL);\ \ \tau \models \partial(S::('\tau,'\alpha::bot)Sequence)\ ]\!] \Longrightarrow$
$(x \in (S$ `->excluding` $x))\ \tau = \mathsf{F}\ \tau$
apply(*simp add: localValidDefined2sem DEF_def_both, clarify*)
apply(*frule DEF_X_Sequence′, clarify*)
apply(*simp add: OclIncludes_def OclExcluding_def OclFalse_def ss_lifting′*)
done

lemma *size_of_excluding*:
$\|(S::('\tau,'\alpha::bot)Sequence)$ `->excluding` $(x::('\tau,'\alpha)VAL)\| =$
$\|S\| - OclCount\ S\ x$
apply(*rule Sequence_sem_cases_ext, simp_all*)
apply(*case_tac x* $\tau \neq \bot$)
apply(*frule neq_commute*[*THEN iffD1*])
apply(*simp_all add: OclSize_def OclExcluding_def OclCount_def*

```
                    minus_def ss_lifting')
  apply(induct_tac c, auto)
  done

lemma excluding_by_select:
  ⟦ τ ⊨ ∂ x ⟧ ⟹
   ( (S::('τ,'α::bot)Sequence) ->excluding (x::('τ,'α) VAL)) τ =
   (⦇ a : S |  (¬ (a ≐ x)) ⦈) τ
  apply(rule_tac X=S in Sequence_induct_including)
  apply simp
  apply(simp,ocl_subst,simp)
  apply(simp_all add:if_not)


  apply(rule trans, rule_tac A=X ->excluding x in cp_charn)
  apply simp_all
  done
```

Sum

Like collect sum is specified as mapping to other OCL functions . . .this is because for every type supporting *plus*, *zero* the sum function must be available and it make no sense to add a new definition for every one of these

```
defs

  OclSum_def       :
  OclSum S         ≡ OclIterate (S::('τ, Integer_0)Sequence) (λ x y. x + y) 0




thm OclSum_def
thm OclIterate_def

lemma sum_by_iterate: OclSum S = OclIterate (S::('τ, Integer_0)Sequence) (λ x y. x + y)
0
  by(simp add: OclSum_def)

lemma OCL_cp_OclSum [simp,intro!]:
  cp P ⟹ cp(λ S. (OclSum (P S::('τ,Integer_0)Sequence)))
  by(simp add: OclSum_def)

lemmas OCL_undef_1_OclSum [simp] = f1_by_iterate_undef_1[OF sum_by_iterate]
lemmas sum_empty [simp] = f1_by_iterate_mtSequence[OF sum_by_iterate]
thm f1_by_iterate_including[OF sum_by_iterate, simplified]
lemmas sum_including [simp] = f1_by_iterate_opt_including[OF sum_by_iterate, simplified]
lemmas sum_union [simp] = f1_by_iterate_distrib_union_opt[OF sum_by_iterate, of op
+, simplified plus_AC, simplified]
```

thm *OCL_undef_1_OclSum sum_empty sum_including sum_union*

One

defs

$OclOne\_def$:
$OclOne \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
$\quad if\ \forall\, x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ DEF\ (P\ x)$
$\quad then\ \llcorner size\ (List.filter\ (\lambda x.\ P\ x = \llcorner True\lrcorner)\ \ulcorner Rep\_Sequence\_0\ S\urcorner) = 1\lrcorner$
$\quad else\ \bot))$

ocl_setup_op [*OclOne*]

Note that contrary to many other OCL functions it is *not possible* to represent the one operation directly by an iterate. The reason for it is, that the internal state space that is used has cardinality three whereas the state space of the output values has cardinality two.

lemma *one_by_size_of_select*:
$(OclOne\ (S{::}('\tau,'\alpha{::}bot)Sequence)\ P) =$
$\|((\!|x : S \mid P\ (x{::}('\tau,'\alpha)\ VAL)|\!))\| \doteq 1$
apply(*rule_tac X=S* **in** *Sequence_sem_cases_ext*)
apply(*auto simp*: *OclSelect_def OclSize_def OclStrictEq_def cp_by_cpify*
OclOne_def One_ocl_int_def ss_lifting')
done

lemma *one_of_mtSequence* [*simp*]:
$OclOne\ ([]{::}('\tau,'\alpha{::}bot)Sequence)\ (P{::}('\tau,'\alpha)\ VAL \Rightarrow ('\tau\ Boolean)) = \mathsf{F}$
apply(*simp add*: *one_by_size_of_select*)
apply(*simp add*: *OclStrongEq_def OclFalse_def ss_lifting'*
One_ocl_int_def Zero_ocl_int_def)
done

lemma *one_singleton* [*simp*]:
$[\![\ \tau \vDash \partial\ (x{::}('\tau,'\alpha)\ VAL);\ cp\ P\ ]\!] \Longrightarrow$
$OclOne\ ([]{::}('\tau,'\alpha{::}bot)Sequence\ \texttt{->including}\ x)\ (P{::}('\tau,'\alpha)\ VAL \Rightarrow ('\tau\ Boolean))\ \tau$
$= P\ x\ \tau$
apply(*simp only*: *one_by_size_of_select*)
apply(*rule trans*)
apply(*rule_tac* $A=(\!|x : []\ \texttt{->including}\ x \mid (P\ x)|\!)$ **in** *cp_charn*)
apply(*simp_all*)
apply(*rule_tac x=P x τ* **in** *boolean_cases_sem*)
apply(*simp_all add*: *OclSize_def OclStrictEq_def OclIncluding_def OclMtSequence_def*
OclIf_def One_ocl_int_def neq_commute
ss_lifting' localValidDefined2sem)
done

lemma *one_union* [*simp*]:
  $OclOne\ ((X::('\tau,'\alpha::bot)Sequence) \cup Y)\ P =$
   $(OclOne\ X\ P)\ \dot{\oplus}\ (OclOne\ Y\ (P::('\tau,'\alpha)\ VAL \Rightarrow '\tau\ Boolean))$
  apply(*rule ext*)
  apply(*case_tac* $x \models \emptyset\ X$, *ocl_hypsubst*, *simp*)
  apply(*simp add*: *OclSxor_def ss_lifting′*)
  apply(*case_tac* $x \models \emptyset\ Y$, *ocl_hypsubst*, *simp*)

  apply(*simp add*: *one_by_size_of_select*)
  oops

IsUnique

defs

   OclIsUnique_def :
   $OclIsUnique \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
      $if\ \ \forall\ x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ DEF\ (P\ x)$
      $then\ \llcorner distinct\ (map\ (\lambda x.\ P\ x)\ \ \ulcorner Rep\_Sequence\_0\ S\urcorner)\lrcorner$
      $else\ \bot))$

ocl_setup_op [*OclIsUnique*]

lemma *isUnique_by_forall*:
  $OclIsUnique\ (S::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha)\ VAL \Rightarrow ('\tau,'c::bot)\ VAL) =$
   $OclIterate\ S\ (\lambda\ x\ y.\ ((OclCount\ (OclCollectNested\ S\ P)\ (P\ x)) \doteq 1)\ \dot{\wedge}\ y)\ \mathsf{T}$
  oops

Any

defs
   OclAny_def :
  $OclAny \quad \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
     $if\ \ (\forall\ x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ DEF\ (P\ x)) \wedge$
       $(\exists\ x \in set\ \ulcorner Rep\_Sequence\_0\ S\urcorner.\ P\ x = \llcorner True\lrcorner)$
     $then\ hd\ (List.filter\ (\lambda\ x.\ P\ x = \llcorner True\lrcorner)\ \ulcorner Rep\_Sequence\_0\ S\urcorner)$
     $else\ \bot))$

ocl_setup_op [*OclAny*]

lemma *any_by_first_of_select*:
  $OclAny\ (S::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha)\ VAL \Rightarrow '\tau\ Boolean) =$
   $OclFirst\ ((OclAsSequence\ (OclSelect\ S\ P))::('\tau,'\alpha)Sequence))$
  apply(*rule_tac X=S* **in** *Sequence_sem_cases_ext*, *simp_all*)
  apply(*auto simp*: *OclAny_def OclSelect_def OclFirst_def*
      cp_by_cpify ss_lifting′ filter_empty_conv)
  done

lemma *any_of_mtSequence[simp]*:
  $OclAny\ ([]::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha)\ VAL \Rightarrow ('\tau\ Boolean)) = \bot$
  by(*simp add: any_by_first_of_select*)

collectNested

defs

$OclCollectNested\_def$:
$OclCollectNested \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
$\quad if\ \forall\ x \in set\ \ulcorner Rep\_Sequence\_0\ S \urcorner.\ DEF\ (P\ x)$
$\quad then\ Abs\_Sequence\_0\ \llcorner map\ (\lambda x.\ P\ x)\ \ulcorner Rep\_Sequence\_0\ S \urcorner \lrcorner$
$\quad else\ \bot))$

ocl_setup_op [*OclCollectNested*]

lemma *collectNested_by_iterate*:
 $((OclCollectNested\ (X::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha)\ VAL \Rightarrow ('\tau,'c)\ VAL))::('\tau,'c::bot)Sequence)$
$=$
  $(X$ `->iterate`$(x;\ y=[]\ |\ \ y$ `->including` $(P\ x)))$
  apply(*rule ext, rule iterate_universal_0, simp_all*)
  apply(*simp add: OclCollectNested_def cp_by_cpify lift0_def*)
  apply(*simp_all add: OclUndefined_def OclMtSequence_def OclCollectNested_def*
    localValidDefined2sem ss_lifting')
  apply(*frule DEF_X_Sequence', clarify*)
  apply(*frule neq_commute[THEN iffD1]*)
  apply(*rotate_tac 1, frule neq_commute[THEN iffD1]*)
  apply(*auto simp: OclIncluding_def ss_lifting'*)
  apply(*subst (asm) Abs_Sequence_0_inject_absurd22, auto*)
  apply(*subst Abs_Sequence_0_inverse_charn2, auto*)
  done

Properties derived from the correspondence to iterate

lemmas *collectNested_of_mtSequence [simp] = f1_by_iterate_mtSequence[OF collectNested_by_iterate]*
thm *f1_by_iterate_including[OF collectNested_by_iterate, simplified]*
lemmas *collectNested_of_including [simp] = f1_by_iterate_opt_including[OF collectNested_by_iterate, simplified]*
lemmas *collectNested_of_union [simp] = f1_by_iterate_distrib_union_opt[OF collectNested_by_iterate, of OclUnion, simplified]*

thm *OCL_undef_1_OclCollectNested collectNested_of_mtSequence collectNested_of_including collectNested_of_union*

Checking a general property of iterate... on foldl

This lemma could actually be made stronger because the two assumptions *undef_1_P* and *cp_P* are not needed. But they make the proof much simpler.

```
lemma iterate_of_collectNested:
  assumes undef_1_Q: ⋀ y. Q ⊥ y = ⊥
  and     undef_2_Q: ⋀ x. Q x ⊥ = ⊥
  and       cp1_Q: ⋀ y. cp (λx. Q x y)
  and       cp2_Q: ⋀ x. cp (Q x)
  and     undef_1_P: P ⊥ = ⊥
  and        cp_P: cp P
  shows OclIterate ((OclCollectNested (S::('τ,'α::bot)Sequence)
            (P::('τ,'α) VAL⇒('τ,'c::bot) VAL))::('τ,'c)Sequence) Q A =
        OclIterate S (λ x y. Q (P x) y) A
  apply(insert undef_1_Q undef_2_Q cp1_Q cp2_Q undef_1_P cp_P)
  apply(rule ext)
  apply(rule iterate_universal, simp_all)
  apply(rule_tac P'=P in cp_compose2)
  apply(simp_all add: collectNested_by_iterate iterate_opt_of_including )
  done
```

Definedness rules

```
lemma undefCollectNestedI:
  assumes exUndef1: τ ⊨ x ∈ (C::('τ,('α::bot))Sequence)
  and     exUndef2: τ ⊨ ∂ (P::('τ,'α::bot) VAL ⇒ ('τ,'c::bot) VAL) x
  and       cp_P: cp P
  shows            τ ⊨ ∂ ((OclCollectNested C P)::('τ,'c::bot)Sequence)
  apply(insert exUndef1 exUndef2)
  apply(frule isDefined_if_valid)
  apply(subst (asm) OCL_is_defopt_OclIncludes)
  apply(clarify, simp add: localValidDefined2sem DEF_def)
  apply(frule DEF_X_Sequence', clarify)
  apply(simp add: localValid2sem OclIsDefined_def OclNot_def OclCollectNested_def
              OclIncludes_def ss_lifting')
  apply(clarify)
  apply(erule_tac x=x τ in ballE)
  apply(insert cp_P, simp_all add: cp_by_cpify lift0_def)
  done
```

Flatten

This definition is currently not complete as it deals only with the type sequence of sequences of some type. There must be written a tactic that defines all needed instances of flatten on the different types. See the semester thesis for further information about the problems around flatten.

defs

```
OclFlatten_def: OclFlatten ≡ lift1(strictify (λ S.
                 Abs_Sequence_0 ⌊concat (map (λ x. ⌜Rep_Sequence_0 x⌝)
                                  ⌜Rep_Sequence_0 S⌝)⌋))
```

ocl_setup_op [*OclFlatten*]

lemma *OCL_is_defopt_OclFlatten* [*simp*]:
$(\tau \models \partial ((\llbracket X \rrbracket)::('\tau,'\alpha::bot)Sequence)) = (\tau \models \partial (X::('\tau,'\alpha\ Sequence\_0)Sequence))$
apply(*rule_tac X=X* **in** *Sequence_sem_cases, simp_all*)
apply(*simp_all add: localValidDefined2sem OclFlatten_def ss_lifting'*)
apply(*clarify*)
apply(*subst (asm) Abs_Sequence_0_inject_absurd22, auto*)
apply(*rule_tac x=a* **in** *Abs_Sequence_0_cases_charn, simp_all*)
done

Its computational characteristic by iterate

lemma *flatten_by_iterate*:
$((OclFlatten\ (X::('\tau,'\alpha::bot\ Sequence\_0)Sequence))::('\tau,'\alpha)Sequence) =$
$(X\ {-}{>}iterate(x;\ y{=}[]\ |\ \ y \cup ((x::('\tau,'\alpha)Sequence))))$
apply(*rule ext, rule iterate_universal_0, simp_all*)
apply(*simp_all add: OclUndefined_def OclMtSequence_def OclFlatten_def localValidDefined2sem ss_lifting'*)
apply(*frule DEF_X_Sequence', clarify*)
apply(*frule neq_commute[THEN iffD1]*)
apply(*rotate_tac 1, frule neq_commute[THEN iffD1]*)
apply(*frule DEF_X_Sequence', clarify*)
apply(*simp add: OclUnion_def OclIncluding_def ss_lifting'*)
apply(*subgoal_tac* $\bot \notin$ *set (concat (map* $(\lambda(x::'\alpha\ Sequence\_0).\ \ulcorner(Rep\_Sequence\_0\ x)\urcorner)$ *c)))*

apply(*auto*)
apply(*rule_tac x=a* **in** *Abs_Sequence_0_cases_charn, auto*)
done

Deriving the rules that follow from the correspondence to iterate

lemmas *flatten_of_mtSequence* [*simp*] = *f1_by_iterate_mtSequence*[*OF flatten_by_iterate*]
lemmas *flatten_of_including* [*simp*] = *f1_by_iterate_opt_including*[*OF flatten_by_iterate, simplified*]
lemmas *flatten_of_union* [*simp*] = *f1_by_iterate_distrib_union_opt*[*OF flatten_by_iterate, of OclUnion, simplified*]

thm *OCL_undef_1_OclFlatten flatten_of_mtSequence flatten_of_including flatten_of_union*

functions applied to flatten

lemma *f_of_flatten*:
assumes *g_by_iterate*: $g\ f\ X\ \tau = OclIterate\ X\ (\lambda\ x\ y.\ h\ y\ (f\ x))\ A\ \tau$
and *f_union*: $\bigwedge X\ Y.\ \llbracket \tau \models \partial\ X;\ \tau \models \partial\ Y \rrbracket \Longrightarrow f\ (X \cup Y)\ \tau = h\ (f\ X)\ (f\ Y)\ \tau$
and *f_empty*: $f\ []\ \tau = (A::('\tau,'c::bot)VAL)\ \tau$
and *f_undef*: $f\ \bot\ \tau = \bot\ \tau$
and *cp_f*: $cp\ f$
and *cp1_h*: $\bigwedge y.\ cp\ (\lambda\ x.\ h\ x\ y)$

```
and          cp2_h: ⋀ x. cp (h x)
shows f ((⨆(X::('τ,'α Sequence_0)Sequence)⨆)::('τ,'α::bot)Sequence) τ
      = ((g f X)::('τ,'c::bot) VAL) τ
apply(insert cp1_h cp2_h cp_f, subst g_by_iterate)
apply(rule_tac X=X and τ=τ in  Sequence_induct_including)
apply(simp_all add: f_undef f_empty f_union)
apply(subst iterate_of_including, simp_all)
apply(rule_tac P'=f in cp_compose2, simp_all)
apply(rule_tac A=(f ⨆X⨆) in cp_charn, simp_all)
apply(rule cp_eq, rule_tac P=f in cp_compose2, simp_all)
done

lemma isEmpty_flatten:
 (((⨆(X::('τ,'α::bot Sequence_0)Sequence)⨆)::('τ,'α)Sequence) ≐ []) =
 (∀ x ∈ X . ((x::('τ,'α)Sequence) ≐ []))
apply(rule ext, rule_tac X=X and h=OclAnd in f_of_flatten)
apply(simp_all add: forall_by_iterate OCL_logic_ACI
                and_equiv_sand[symmetric] OCL_is_defopt_OclStrictEq
            del: OCL_is_def_OclStrictEq)
done

lemma includes_flatten:
 ⟦ τ ⊨ ∂ (a::('τ,'α::bot) VAL) ⟧ ⟹
 (a ∈ ((⨆(X::('τ,'α Sequence_0)Sequence)⨆)::('τ,'α)Sequence)) τ =
 (∃ Y ∈ X . (a ∈ (Y::('τ,'α)Sequence))) τ
apply(rule_tac X=X and h=OclOr in f_of_flatten)
apply(simp_all add: includes_of_union exists_by_iterate)
apply(ocl_subst, simp add: OCL_logic_ACI)
done

lemma excludes_flatten:
 ⟦ τ ⊨ ∂ (a::('τ,'α::bot) VAL) ⟧ ⟹
 (a ∉ ((⨆(X::('τ,'α Sequence_0)Sequence)⨆)::('τ,'α)Sequence)) τ =
 (∀ Y ∈ X . (a ∉ (Y::('τ,'α)Sequence))) τ
apply(rule_tac X=X and h=OclAnd in f_of_flatten)
apply(simp_all add: includes_of_union forall_by_iterate)
apply(ocl_subst, simp add: OCL_logic_ACI)

apply(rule trans, rule_tac A=(a ∈ (X ∪ Y )) in cp_charn)
apply(rule includes_of_union, simp_all)
apply(simp add: OclAnd_def OclOr_def OclNot_def ss_lifting')
done
```

The current formulation is NOT true. But what is an elegant solution to it?

```
lemma includesAll_flatten:
 (OclIncludesAll X ((⨆(Y::('τ,'α Sequence_0)Sequence)⨆)::('τ,'α)Sequence))=
 (∀ Z ∈ Y . (OclIncludesAll X (Z::('τ,'α::bot)Sequence)))
oops
```

The current formulation is NOT true. But what is an elegant solution to it?

lemma *excludesAll_flatten*:
 $(OclExcludesAll\ X\ ((⟦(Y::('\tau,'\alpha\ Sequence\_0)Sequence)⟧)::('\tau,'\alpha)Sequence))=$
 $(\forall\ Z \in\ Y\ .\ (OclExcludesAll\ X\ (Z::('\tau,'\alpha::bot)Sequence)))$
 oops

lemma *forall_flatten*:
 $(\forall\ y \in ((⟦(X::('\tau,'\alpha\ Sequence\_0)Sequence)⟧)::('\tau,'\alpha)Sequence)\ .\ P\ y) =$
 $(OclForAll\ X\ (\lambda\ Y::('\tau,'\alpha)Sequence.\ (\forall\ y \in\ Y\ .\ P\ (y::('\tau,'\alpha::bot)\ VAL))))$
 apply(*rule ext*, *rule_tac X=X* **and** *h=OclAnd* **in** *f_of_flatten*)
 apply(*simp_all*, *simp_all add*: *forall_by_iterate OCL_logic_ACI*)
 done

lemma *exists_flatten*:
 $(\exists\ y \in ((⟦(X::('\tau,'\alpha\ Sequence\_0)Sequence)⟧)::('\tau,'\alpha)Sequence)\ .\ P\ y) =$
 $(OclExists\ X\ (\lambda\ Y::('\tau,'\alpha)Sequence.\ (\exists\ y \in\ Y\ .\ P\ (y::('\tau,'\alpha::bot)\ VAL))))$
 by(*simp add*: *OclExists_def forall_flatten*)

lemma *select_flatten*:
 $OclSelect\ ((⟦(X::('\tau,'\alpha\ Sequence\_0)Sequence)⟧)::('\tau,'\alpha::bot)Sequence)\ (P::(('\tau,'\alpha)\ VAL \Rightarrow$
$('\tau\ Boolean))) =$
 $⟦((OclCollectNested\ X\ (\lambda\ (Y::('\tau,'\alpha)Sequence).\ OclSelect\ Y\ P))::('\tau,'\alpha\ Sequence\_0)Sequence)⟧$
 apply(*rule ext*)
 apply(*rule_tac X=X* **and** *τ=x* **in** *Sequence_induct_including*, *simp_all*)

 apply(*rule trans*)
 apply(*rule_tac A=OclSelect* $⟦X⟧$ *P* **and** *τ=x* **in** *cp_charn*)
 apply(*simp_all*)
 done

lemma *reject_flatten*:
 $OclReject\ ((⟦(X::('\tau,'\alpha\ Sequence\_0)Sequence)⟧)::('\tau,'\alpha::bot)Sequence)\ (P::(('\tau,'\alpha)\ VAL \Rightarrow$
$('\tau\ Boolean))) =$
 $⟦((OclCollectNested\ X\ (\lambda\ (Y::('\tau,'\alpha)Sequence).\ OclReject\ Y\ P))::('\tau,'\alpha\ Sequence\_0)Sequence)⟧$
 by(*simp add*: *reject_by_select select_flatten*)

By rewriting any to `first_of_select it` is clear what the result of any on a flattened sequence will be.

Currently there is no simplification rule for the equalities. I just can't see a way how to represent the iterated comparison of a prefix of both sequences with its intertwined concatenations.

By rewriting one to `size_of_select` it is clear what the result of one on a flattened sequence will be.

lemma *size_flatten*:
 $\|((⟦(X::('\tau,'\alpha\ Sequence\_0)Sequence)⟧)::('\tau,'\alpha::bot)Sequence)\| =$
 $OclSum\ ((OclCollectNested\ X\ (OclSize::('\tau,'\alpha)Sequence \Rightarrow '\tau\ Integer))::('\tau,Integer\_0)Sequence)$
 apply(*rule ext*)

apply(*rule_tac X=X* **and** τ=*x* **in** *Sequence_induct_including, simp_all*)
apply(*rule trans*)
apply(*rule_tac A*=‖⌈⌈*X*⌉⌉‖ **and** τ =*x* **in** *cp_charn*)
apply(*simp_all add: plus_AC*)
done

lemma *count_flatten*:
⟦ τ ⊨ ∂ *a* ⟧ ⟹
(((⌈⌈(*X*::('τ,'α *Sequence_0*)*Sequence*)⌉⌉)::('τ,'α::*bot*)*Sequence*) ->count (*a*::('τ,'α) *VAL*)) τ =

OclSum ((*OclCollectNested X* (λ (*X*::('τ,'α)*Sequence*). *X*->count *a*))::('τ,*Integer_0*)*Sequence*)
τ
apply(*rule_tac X=X* **and** τ=τ **in** *Sequence_induct_including, simp_all*)
apply(*rule trans*)
apply(*rule_tac A*=(⌈⌈*X*⌉⌉ ->count *a*) **and** τ=τ **in** *cp_charn*)
apply(*simp_all add: plus_AC*)
done

lemma *sum_flatten*:
OclSum ((⌈⌈(*X*::('τ,*Integer_0 Sequence_0*)*Sequence*)⌉⌉)::('τ,*Integer_0*)*Sequence*) =
OclSum ((*OclCollectNested X* (*OclSum*::('τ,*Integer_0*)*Sequence*⇒'τ *Integer*))::('τ,*Integer_0*)*Sequence*)
apply(*rule ext*)
apply(*rule_tac X=X* **and** τ=*x* **in** *Sequence_induct_including, simp_all*)
apply(*rule trans*)
apply(*rule_tac A*=->sum() ⌈⌈*X*⌉⌉ **and** τ=*x* **in** *cp_charn*)
apply(*simp_all add: plus_AC*)
done

lemma *excluding_flatten*:
⟦ τ ⊨ ∂ *x* ⟧ ⟹
OclExcluding ((⌈⌈(*X*::('τ,'α *Sequence_0*)*Sequence*)⌉⌉)::('τ,'α::*bot*)*Sequence*) (*x*::('τ,'α) *VAL*)
τ =
⌈⌈((*OclCollectNested X* (λ (*Y*::('τ,'α)*Sequence*). *OclExcluding Y x*))::('τ,'α *Sequence_0*)*Sequence*)⌉⌉
τ
apply(*rule_tac X=X* **and** *h=OclUnion* **in** *f_of_flatten*)
apply(*simp_all*)
apply(*ocl_subst*)
apply(*simp add: flatten_by_iterate iterate_of_collectNested*)
done

lemma *collectNested_flatten*:
((*OclCollectNested* ((⌈⌈(*X*::('τ,'α *Sequence_0*)*Sequence*)⌉⌉)::('τ,'α::*bot*)*Sequence*)
(*P*::('τ,'α) *VAL*⇒('τ,'*c*::*bot*) *VAL*))::('τ,'*c*)*Sequence*) =
⌈⌈((*OclCollectNested X* (λ (*Y*::('τ,'α)*Sequence*). (*OclCollectNested Y P*)::('τ,'*c*)*Sequence*))::('τ,'*c*
Sequence_0)*Sequence*)⌉⌉
apply(*rule ext*)
apply(*rule_tac X=X* **and** τ=*x* **in** *Sequence_induct_including, simp_all*)
apply(*rule trans*)
apply(*rule_tac A=OclCollectNested* ⌈⌈*X*⌉⌉ *P* **and** τ=*x* **in** *cp_charn*)

```
  apply(simp_all)
  done

lemma test':
  ‖((⟦
      ((⟦(X::('τ,'α Sequence_0 Sequence_0)Sequence)⟧)::('τ,'α Sequence_0)Sequence)
    ⟧)::(('τ,'α::bot)Sequence))‖ = h
  apply(simp add: size_flatten collectNested_flatten sum_flatten)
  oops
```

Sequence specific functions

These are: `first`, `last`, `at`, `insertAt`, `indexOf`, `subSequence`

```
lemma last_empty [simp]:
  OclLast (⟦⟧::('τ,'α::bot)Sequence) = (⊥::('τ,'α) VAL)
  by(simp add: last_at_UC)

lemma last_including [simp]:
  ⟦ τ ⊨ ∂ S⟧ ⟹
   OclLast ((S::('τ,'α::bot)Sequence) ->including (x::('τ,'α) VAL)) τ =
   ((x τ)::'α)
  apply(rule_tac X=S in Sequence_sem_cases)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(simp add: OclIncluding_def OclLast_def
                  ss_lifting' neq_commute)
  done

lemma last_union:
  ⟦ τ ⊨ ∂ X; τ ⊨ (Y ‘<>‘ ⟦⟧) ⟧ ⟹
  OclLast ((X::('τ,'α::bot)Sequence) ∪ Y) τ = ((OclLast Y)::('τ,'α) VAL) τ
  apply(rule_tac X=X in Sequence_sem_cases)
  apply(rule_tac X=Y in Sequence_sem_cases, simp_all)
  apply(auto simp: localValid2sem OclNot_def OclStrictEq_def OclMtSequence_def
                   OclUnion_def OclLast_def OclIsDefined_def ss_lifting')
  done


lemma ForAllD:
  assumes allP : τ ⊨ (∀ x ∈ C • P(x::'τ ⇒ 'α::bot))
  and     cpP     : cp P
  shows   ⋀ x. τ ⊨ x ∈ (C::('τ,('α::bot))Sequence) ⟹ τ ⊨ P x
  apply (insert allP cpP)
  apply (simp add: localValid2sem OclForAll_def OclNot_def OclIncludes_def
                   ss_lifting')
  apply (simp add:   strictify_def
              split: split_if_asm)
  apply (erule_tac x=x τ in ballE)
  apply (simp_all add: cp_by_cpify lift0_def)
```

```
  done

lemma isdefSelectI:
  assumes localDef: τ ⊨ ∀ x ∈ C • (∂(P (x::('τ,'α::bot) VAL)))
  and        cp_P: cp P
  shows τ ⊨ ∂ (OclSelect (C::('τ,'α::bot)Sequence) P)
  apply(insert localDef)
  apply(erule rev_mp)
  apply(rule_tac X=C in Sequence_induct_including)
  apply(simp_all, clarify)
  oops

lemma last_flatten:
  ((OclLast ((⟦(X::('τ,'α Sequence_0)Sequence)⟧)::('τ,'α::bot)Sequence))::('τ,'α) VAL) =
   OclLast ((OclLast (OclSelect X (λ Y::('τ,'α)Sequence. Y ‘<>‘ [])))::('τ,'α)Sequence)
  apply(rule ext)
  apply(rule_tac X=X in Sequence_induct_including, simp_all)
  apply(case_tac x ⊨ (a ‘<>‘ []))

  apply(rule trans[symmetric])
  apply(rule_tac A=(a ‘<>‘ []) in cp_charn)

  apply(rotate_tac -1, subst (asm) is_TRUE_charn_local[symmetric])
  apply(simp_all)
  apply(simp add: last_union)
  apply(rule_tac B=a in cp_charn)
  apply(rule last_including)
  apply(simp_all)


  prefer 2
  apply(simp add: if_not)
  apply(subst (asm) localValidNot2not[symmetric])
  apply(simp add: weak_prop_LJE)
  apply(simp)
  apply(rotate_tac -1, ocl_subst)
  apply(simp add: weak_prop_LJE)
  apply(rule_tac  t=OclLast ((⟦X⟧) ∪ a ) x and
                  s=OclLast (⟦X⟧) x in subst)

  apply(rotate_tac 1, thin_tac ?P)
  apply(rule trans[symmetric])
  apply(rule_tac A=a and B=[] in cp_charn)
 apply(simp add: OclLast_def OclUnion_def localValid2sem OclStrictEq_def OclFlatten_def
                 OclIsDefined_def OclMtSequence_def ss_lifting')
  apply(simp_all)

  oops
```

Collect

Note: Currently collectNested is only defined here but it should actually be polymorphically defined in the theory OCL_Collection. Furthermore the collect operation could be defined right there because it should be defined on the OCL level to avoid having to define collect over all different types like we have to do it for flatten.

Defining collect based on collectNested according to the standard

defs

$OclCollect\_def$:
$(OclCollect::[('\tau,'\alpha::bot)Sequence, (('\tau,'\alpha)\,VAL \Rightarrow ('\tau,'c::bot)Sequence)] \Rightarrow ('\tau,'c::bot)Sequence)$
$S\ P$
$\equiv \lceil\!\lceil\ ((OclCollectNested\ S\ P)::('\tau,'c\ Sequence\_0)Sequence)\ \rceil\!\rceil$

$OclCollect\_Integer\_def$:
$(OclCollect::[('\tau,'\alpha::bot)Sequence, (('\tau,'\alpha)\,VAL \Rightarrow\ '\tau\ Integer)] \Rightarrow ('\tau,Integer\_0)Sequence)$
$S\ P$
$\equiv (OclCollectNested\ S\ P)$

lemma $collect\_by\_iterate$:
$[\![\ P\ \bot = \bot;\ cp\ P\ ]\!] \Longrightarrow$
$((OclCollect\ (X::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha)\,VAL \Rightarrow ('\tau,'c::bot)Sequence))::('\tau,'c)Sequence)$
$=$
$(X$ **->iterate**$(x;y=[]\ |\ (y \cup (P\ x)\ )))$
by$(simp\ add{:}\ OclCollect\_def\ flatten\_by\_iterate\ iterate\_of\_collectNested)$

lemma $OCL\_cp\_OclCollect[simp,intro!]$:
$[\![\ \bigwedge\ y.\ cp\ (\lambda\ x.\ P\ x\ y);\ cp\ S\ ]\!]$
$\Longrightarrow cp(\lambda X.\ (OclCollect\ ((S\ X)::('\tau,'\alpha::bot)Sequence)$
$((P\ X)::('\tau,'\alpha)\,VAL \Rightarrow ('\tau,'c)Sequence))::('\tau,'c::bot)Sequence)$
by$(simp\ add{:}\ OclCollect\_def)$

lemma $OCL\_undef\_1\_OclCollect[simp]$:
$OclCollect\ (\bot::('\tau,'\alpha::bot)Sequence)\ (P::('\tau,'\alpha)\,VAL \Rightarrow ('\tau,'c)Sequence) =$
$(\bot::('\tau,'c::bot)Sequence)$
by$(simp\ add{:}\ OclCollect\_def)$

end

B.4.11. OCL Bag

theory *OCL_Bag*
imports
$\$HOLOCL\_HOME/src/library/collection/smashed/OCL\_Sequence$
begin

Properties of the the Datatype Adaption

The following rules are transformed versions of the automatically generated rules for datatype adaption

lemma *Abs_Bag_0_inject_absurd11* [*simp*]:
$[\![\ count\ x\ \bot = 0\ ]\!] \Longrightarrow ((Abs\_Bag\_0\ down) = (Abs\_Bag\_0\ \llcorner x \lrcorner)) = False$
by(*subst Abs_Bag_0_inject,*
simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma *Abs_Bag_0_inject_absurd12* [*simp*]:
$[\![\ count\ x\ \bot = 0\ ]\!] \Longrightarrow ((Abs\_Bag\_0\ \llcorner x \lrcorner) = (Abs\_Bag\_0\ down)) = False$
by(*subst Abs_Bag_0_inject,*
simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma *Abs_Bag_0_inject_absurd21* [*simp*]:
$[\![\ count\ x\ \bot = 0\ ]\!] \Longrightarrow (\ \bot = (Abs\_Bag\_0\ \llcorner x \lrcorner)) = False$
by(*simp add: UU_Bag_def*)

lemma *Abs_Bag_0_inject_absurd22* [*simp*]:
$[\![\ count\ x\ \bot = 0\ ]\!] \Longrightarrow ((Abs\_Bag\_0\ \llcorner x \lrcorner) = \bot) = False$
by(*simp add: UU_Bag_def*)

lemma *Abs_Bag_0_inject_charn* [*simp*]:
$[\![\ count\ x\ (\bot::'a::bot) = 0;\ count\ y\ (\bot::'a::bot) = 0\ ]\!] \Longrightarrow$
$(((Abs\_Bag\_0\ \llcorner x \lrcorner) = (Abs\_Bag\_0\ \llcorner y \lrcorner)) = (x = y))$
by(*subst Abs_Bag_0_inject,*
simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma *Abs_Bag_0_inverse_charn* [*simp*]:
$(count\ x\ \bot = 0) \Longrightarrow Rep\_Bag\_0\ (Abs\_Bag\_0\ \llcorner x \lrcorner) = \llcorner x \lrcorner$
by (*subst Abs_Bag_0_inverse,*
simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma *Abs_Bag_0_inverse_charn2* [*simp*]:
$(count\ x\ \bot = 0) \Longrightarrow \ulcorner Rep\_Bag\_0\ (Abs\_Bag\_0\ \llcorner x \lrcorner) \urcorner = x$
by (*subst Abs_Bag_0_inverse,*
simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma *Abs_Bag_0_cases_charn*:
assumes *bottomCase* : $[\![\ x = \bot\ ]\!] \Longrightarrow P$
assumes *listCase* : $\bigwedge y.\ ([\![\ x = Abs\_Bag\_0\ \llcorner y \lrcorner;\ count\ y\ \bot = 0\ ]\!] \Longrightarrow P)$
shows P

```
  apply(rule_tac x=x in Abs_Bag_0_cases)
  apply(case_tac y = ⊥)
  apply(rule bottomCase, simp add: UU_Bag_def)
  apply(frule not_down_exists_lift2[THEN iffD1])
  apply(rule_tac y=⌜y⌝ in listCase)
  apply(simp add: OCL_Bag_type.Bag_0_def smash_def DEF_def)
  apply(case_tac (count ⌜y⌝ ⊥) = (0::nat))
  apply(auto simp: OCL_Bag_type.Bag_0_def smash_def DEF_def)
  done

lemma Abs_Bag_0_induct_charn:
  assumes bottomCase : P ⊥
  assumes stepCase   : ⋀y. (⟦ count y ⊥ = 0 ⟧ ⟹ P (Abs_Bag_0 ⌊y⌋))
  shows                P x
  apply(rule_tac x=x in Abs_Bag_0_induct)
  apply(rule_tac  x=Abs_Bag_0 y in Abs_Bag_0_cases_charn)
  apply(auto intro!: bottomCase stepCase)
  done

lemma inj_on_Abs_Bag_0: inj_on Abs_Bag_0 Bag_0
  by(rule inj_on_inverseI,rule Bag_0.Abs_Bag_0_inverse,assumption)

lemma inj_Rep_Bag_0 : inj Rep_Bag_0
  by(rule inj_on_inverseI, rule Bag_0.Rep_Bag_0_inverse)

lemma smashed_bag_charn:
  (count X ⊥ = 0) = ((⌊X⌋) ∈ Bag_0)
  by(unfold smash_def Bag_0_def UU_Bag_def,auto)

lemma UU_in_smashed_bag [simp]:
  ⊥ ∈ Bag_0
  by(unfold smash_def Bag_0_def UU_Bag_def,auto)

lemma down_in_smashed_bag [simp]:
  down ∈ Bag_0
  by(unfold smash_def Bag_0_def UU_Bag_def,auto)

lemma mt_in_smashed_bag[simp]:
  (⌊{#}⌋) ∈ Bag_0
  by (unfold smash_def Bag_0_def,auto)

lemma DEF_Abs_Bag: ⋀X. (count X ⊥ = 0) ⟹ DEF (Abs_Bag_0 (⌊X⌋))
  apply(unfold DEF_def UU_Bag_def, simp)
  done

lemma DEF_Rep_Bag:
  ⋀X. DEF X ⟹ DEF (Rep_Bag_0 X)
  apply(unfold DEF_def UU_Bag_def,auto)
  apply(drule_tac f = Abs_Bag_0 in arg_cong)
```

```
  apply(simp add: Rep_Bag_0_inverse)
  done

lemma not_DEF_Rep_Bag:
  ⋀X. ¬ DEF X ⟹ ¬ DEF (Rep_Bag_0 X)
  apply(unfold DEF_def UU_Bag_def,auto)
  apply(simp add: Abs_Bag_0_inverse)
  done


lemma exists_lift_Bag:
  ⋀X. DEF X ⟹ ∃ c. Rep_Bag_0 X = (⌊c⌋)
  by (drule DEF_Rep_Bag,simp add: DEF_X_up)

lemma exists_lift_Bag2:
  ⋀X. DEF X ⟹ ∃ c. count c ⊥ = 0 ∧ Rep_Bag_0 X = (⌊c⌋)
  apply(frule exists_lift_Bag)
  apply(auto simp: UU_Bag_def DEF_def)
  apply(drule sym, rule swap)
  apply assumption
  apply(unfold not_def)
  apply(subst (asm) smashed_bag_charn)
  apply(simp add: Rep_Bag_0)
  done

lemma Rep_Bag_cases:
  Rep_Bag_0 X = ⊥ ∨ (∃ c. count c ⊥ = 0 ∧ Rep_Bag_0 X = (⌊c⌋))
  apply(case_tac DEF X)
  apply(drule exists_lift_Bag2)
  apply(drule_tac [2] not_DEF_Rep_Bag)
  apply(simp_all add: DEF_def)
  done


lemma DEF_X_Bag_0 : DEF(X) = (∃ c. count c ⊥ = 0 ∧ Rep_Bag_0 X = (⌊c⌋))
  apply(insert Rep_Bag_cases [of X], auto)
  apply(drule exists_lift_Bag,auto)
  apply(rule swap)
  prefer 2
  apply(rule not_DEF_Rep_Bag, auto)
  done
```

The following lemma is crucial for thy_morpher:

```
lemma DEF_X_Bag: DEF X = (∃ c. count c ⊥ = 0 ∧ X = Abs_Bag_0 (⌊c⌋))

  apply (auto simp: DEF_Abs_Bag)
  apply (simp add:DEF_X_Bag_0)
  apply (erule exE, rule exI, auto)
  apply (rule injD [OF inj_Rep_Bag_0])
```

```
apply (auto simp: smashed_bag_charn Abs_Bag_0_inverse)
done
```

This lemma is very convenient in simplifing unfolded definitions

```
lemma DEF_X_Bag':
 ⟦ X ≠ ⊥ ⟧ ⟹
  (∃ c. (count c ⊥ = 0) ∧ (Rep_Bag_0 X = ⌊c⌋)) ∧
  (∃ c. (count c ⊥ = 0) ∧ (X = (Abs_Bag_0 ⌊c⌋)))
 apply(fold DEF_def)
 apply(frule DEF_X_Bag[THEN iffD1])
 apply(drule DEF_X_Bag_0[THEN iffD1])
 apply(simp)
 done


lemma Bag_sem_cases_0:
 assumes defC: ⋀ c d. ⟦ X ≠ ⊥; ⊥ ≠ X;
                        (count c ⊥ = 0); (count d ⊥ = 0);
                        Rep_Bag_0 X = ⌊c⌋;
                        X = Abs_Bag_0 ⌊d⌋ ⟧ ⟹ P X
 and    undefC: ⟦ X = ⊥ ⟧ ⟹ P X
 shows P X
 apply(rule Abs_Bag_0_cases_charn, erule undefC)
 apply(rule defC, auto)
 done

lemma Bag_sem_cases:
 assumes defC: ⋀ c d. ⟦ (X τ) ≠ ⊥; ⊥ ≠ (X τ);
                        (count c ⊥ = 0); (count d ⊥ = 0);
                        Rep_Bag_0 (X τ) = ⌊c⌋;
                        (X τ) = Abs_Bag_0 ⌊d⌋ ⟧ ⟹ P X τ
 and    undefC: ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ
 and      cpP: cp P
 shows P X τ
 apply(rule_tac P2=P in subst[OF sym[OF cp_subst]], rule cpP)
 apply(rule Bag_sem_cases_0)
 apply(rule_tac P1=P in subst[OF cp_subst], rule cpP)
 apply(rule defC) prefer 7
 apply(rule_tac P1=P in subst[OF cp_subst], rule cpP)
 apply(rule undefC, simp_all)
 done

lemma Bag_sem_cases_ext:
 assumes defC: ⋀ c d τ. ⟦ (X τ) ≠ ⊥; ⊥ ≠ (X τ);
                        (count c ⊥ = 0); (count d ⊥ = 0);
                        Rep_Bag_0 (X τ) = ⌊c⌋;
                        (X τ) = Abs_Bag_0 ⌊d⌋ ⟧ ⟹ P X τ = Q X τ
 and    undefC: ⋀ τ. ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ = Q X τ
 and      cpP: cp P
```

```
  and      cpQ: cp Q
  shows P X = Q X
  apply(rule ext)
  apply(rule_tac X=X in Bag_sem_cases)
  apply(rule defC) prefer 7
  apply(rule undefC)
  apply(simp_all add: cpP cpQ)
  done
```

These four lemmas are used by `ocl_setup_op` to reason about definedness:

```
lemma lift2_strict_is_isdef_fw_Bag_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Bag_0 ⌊g ⌈Rep_Bag_0 x⌉ y⌋)))
  and      inv_g: !!a b.( ⟦ count a ⊥ = 0; (⊥::'a::bot) ≠ b ⟧ ⟹ count (g a b) ⊥ = 0)
  shows            ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                  o_def lift0_def lift1_def lift2_def
                  strictify_def DEF_def)
  apply(rule ext)
  apply(case_tac DEF (X τ))
  apply(frule DEF_X_Bag_0[THEN iffD1])
  apply(auto simp: strictify_def DEF_def)
  apply(drule_tac b=(Y τ) in  inv_g)
  prefer 2
  apply(drule DEF_Abs_Bag, simp add: DEF_def, auto)
  done

lemma lift2_strict_is_isdef_fw_Bag_Bag:
 assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Bag_0 ⌊g ⌈Rep_Bag_0 x⌉ ⌈Rep_Bag_0
y⌉⌋)))
  and      inv_g: !!a b.( ⟦ count a ⊥ = 0; count b ⊥ = 0 ⟧ ⟹ count (g a b) ⊥ = 0)
  shows            ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                  o_def lift0_def lift1_def lift2_def
                  strictify_def DEF_def)
  apply(rule ext)
  apply(case_tac DEF (X τ))
  apply(case_tac DEF (Y τ))
  apply(drule DEF_X_Bag_0[THEN iffD1])+
  apply(auto simp: strictify_def DEF_def)
  apply(drule_tac b=ca in inv_g, assumption)
  apply(rotate_tac −1)
  apply(drule DEF_Abs_Bag, simp add: DEF_def)
  done

lemma lift2_strictify_implies_LocalValid_defined_Bag_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Bag_0 ⌊g ⌈Rep_Bag_0 x⌉ y⌋)))
  and      inv_g: !!a b.( ⟦ count a ⊥ = 0; (⊥::'a::bot) ≠ b⟧ ⟹ count (g a b) ⊥ = 0)
  shows            (τ ⊨ ∂(f X Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
  apply(insert f_def)
```

apply(*drule_tac X=X* **and** *Y=Y* **in** *lift2_strict_is_isdef_fw_Bag_Val*)
apply(*rule inv_g, assumption+*)
apply(*simp add: OclAnd_def OclTrue_def o_def*
OclIsDefined_def lift0_def lift1_def lift2_def
OclLocalValid_def)
done

lemma *lift2_strictify_implies_LocalValid_defined_Bag_Bag*:
assumes *f_def*: $f \equiv lift2(strictify(\lambda x.\ strictify(\lambda y.\ Abs\_Bag\_0 \llcorner g \ulcorner Rep\_Bag\_0\ x \urcorner \ulcorner Rep\_Bag\_0$
$y \urcorner \lrcorner)))$
and *inv_g*: $!!a\ b.(\llbracket\ count\ a\ \bot = 0;\ count\ b\ \bot = 0\ \rrbracket \Longrightarrow count\ (g\ a\ b)\ \bot = 0)$
shows $(\tau \vDash \partial(f\ X\ Y)) = ((\tau \vDash \partial\ X) \wedge (\tau \vDash\ \partial\ Y))$
apply(*insert f_def*)
apply(*drule_tac X=X* **and** *Y=Y* **in** *lift2_strict_is_isdef_fw_Bag_Bag*)
apply(*rule inv_g, assumption+*)
apply(*simp add: OclAnd_def OclTrue_def o_def*
OclIsDefined_def lift0_def lift1_def lift2_def
OclLocalValid_def)
done

The conversion operators

From bags to sequences

Informally speaking the AsSequence operation returns just some sequence that has exactly the same elements like the given bag.

Note: With the current definition it does *not* hold that `bag->seq->oset = bag->set->oset`. To get this behaviour one has to define AsSequence as some sequence that has the elements in the same order like the associated set converted to a sequence.

defs

OclAsSequence_def :
OclAsSequence $\equiv lift1\ (strictify\ (\lambda X.$
$Abs\_Sequence\_0 \llcorner list\_of\_multiset \ulcorner Rep\_Bag\_0\ X \urcorner \lrcorner))$

ocl_setup_op [*OclAsSequence*]

lemma *OCL_is_defopt_OclAsSequence*:
$(\tau \vDash \partial\ (((self::('a,'b::bot)Bag)\ {-}{>}asSequence())::(('a,\ 'b::bot)Sequence))) = (\tau \vDash \partial\ self)$
apply(*simp add: localValidDefined2sem OclAsSequence_def ss_lifting'*)
apply(*rule impI, frule DEF_X_Bag', auto*)
done

lemma *OCL_is_def_OclAsSequence*:
$(\partial\ (((self::('a,'b::bot)Bag)\ {-}{>}asSequence())::(('a,\ 'b::bot)Sequence))) = (\partial\ self)$
apply(*rule Bag_sem_cases_ext, simp_all*)
apply(*simp_all add: OclIsDefined_def OclAsSequence_def ss_lifting'*)
done

Characterisation theorems of the conversion from bags to sequences

lemma *OclAsSequence_charn_0*:
 assumes *undefCase*: $P \perp \perp \tau$
 and *defCase* : $\bigwedge xs.\ [\![\ulcorner Rep\_Bag\_0\ B \urcorner = multiset\_of\ xs;\ DEF\ B;\ \perp \notin set\ xs\ ]\!] \Longrightarrow P\ (\lambda\ s.\ (Abs\_Sequence\_0\ \llcorner xs \lrcorner))\ (\lambda\ s.\ B)\ \tau$
 shows $P\ ((OclAsSequence\ (\lambda\ s.\ (B::'b::bot\ Bag\_0)))::('a,'b::bot)Sequence)\ (\lambda\ s.\ B)\ \tau$
 apply(*insert undefCase*)
 apply(*simp add*: *localValidDefined2sem OclIsDefined_def OclAsSequence_def OclUndefined_def ss_lifting'*)
 apply(*rule impI*)
 apply(*frule DEF_X_Bag_0*[*THEN iffD1*, *simplified DEF_def*], *clarify*)
 apply(*rule list_of_multisetI*)
 apply(*rule defCase*)
 apply(*simp_all add*: *localValidDefined2sem DEF_def mem_set_multiset_eq*)
 done

lemma *OclAsSequence_charn*:
 assumes *undefCase*: $P \perp \perp \tau$
 and *defCase* : $\bigwedge xs.\ [\![\ulcorner Rep\_Bag\_0\ (B\ \tau) \urcorner = multiset\_of\ xs;\ \perp \notin set\ xs;\ \tau \vDash \partial B\ ]\!] \Longrightarrow P\ (\lambda\ s.\ (Abs\_Sequence\_0\ \llcorner xs \lrcorner))\ B\ \tau$
 and *cp1_P* : $\bigwedge y.\ cp\ (\lambda\ x.\ P\ x\ y)$
 and *cp2_P* : $\bigwedge x.\ cp\ (P\ x)$
 shows $P\ ((OclAsSequence\ (B::('a,'b::bot)Bag))::('a,'b::bot)Sequence)\ B\ \tau$
 apply(*rule subst*, *rule_tac x1=B* **in** *sym*[*OF cp_subst*])
 apply(*subst cp0_eq_cp*[*symmetric*])
 apply(*rule_tac cp0_app*) **back**
 apply(*rule_tac cp0_of_cp0*) **back**
 apply(*simp_all add*: *cp0_eq_cp cp2_P cp1_P lift0_def*)
 apply(*rule_tac P=P* **in** *OclAsSequence_charn_0*)
 apply(*insert undefCase*, *simp add*: *OclUndefined_def lift0_def*)
 apply(*drule defCase*, *simp_all add*: *localValidDefined2sem*)
 apply(*rule subst*)
 apply(*rule_tac* $P=\lambda\ x\ St.\ (P\ (\lambda\ s.\ (Abs\_Sequence\_0\ \llcorner xs \lrcorner))\ x)\ St$ **and** $A=B$ **in** *cp_charn*)

 apply(*simp_all add*: *cp2_P*)
 done

Showing the distributivity over the equivalences

Note: Actually it should be possible to develop the theory about the inverse of `multiset_of` in a more general way by combinig the results of Hilbert_Choice.thy and Multiset.thy. But I'm currently not using this approach because only few specifically tailored theorems are needed.

lemma *count_filter_inv_multiset_of*:
 $count\ M\ x = length\ (List.filter\ (\lambda\ y.\ y = x)\ (list\_of\_multiset\ M))$
 by(*auto intro*: *list_of_multisetI simp*: *count_filter*[*symmetric*])

```
lemma bag2seq_holEq:
  ((X::('a,'b::bot)Bag) = Y) = (((->asSequence() X)::('a,'b)Sequence) = (->asSequence() Y))
  apply(auto simp: OclAsSequence_def ss_lifting' expand_fun_eq)
  apply(erule_tac x=x in allE)
  apply(case_tac X x ≠ ⊥, frule DEF_X_Bag')
  apply(case_tac Y x ≠ ⊥, rotate_tac -1, frule DEF_X_Bag')
  apply(auto)
  apply(subst (asm) Abs_Sequence_0_inject_charn, auto)
  apply(subst multiset_eq_conv_count_eq)
  apply(subst count_filter_inv_multiset_of)+
  apply(simp)
  apply(case_tac Y x ≠ ⊥, rotate_tac -1, frule DEF_X_Bag')
  apply(auto)
  done

lemma bag2seq_strictEq:
  ((X::('a,'b::bot)Bag) ≐ Y) = (((->asSequence() X)::('a,'b)Sequence) ≐ (->asSequence() Y))
  apply(rule ext)
  apply(simp add: OclStrictEq_def OclAsSequence_def ss_lifting')
  apply(rule impI)+
  apply(frule DEF_X_Bag', rotate_tac 1)
  apply(frule DEF_X_Bag', clarify, simp)
  apply(subst Abs_Sequence_0_inject_charn, auto)
  apply(subst multiset_eq_conv_count_eq)
  apply(subst count_filter_inv_multiset_of)+
  apply(simp)
  done

lemma bag2seq_strongEq:
  ((X::('a,'b::bot)Bag) ≜ Y) = (((->asSequence() X)::('a,'b)Sequence) ≜ (->asSequence() Y))
  apply(rule ext)
  apply(auto simp: OclStrongEq_def OclAsSequence_def ss_lifting')
  apply(frule DEF_X_Bag', clarify, simp)
  apply(frule DEF_X_Bag', clarify, simp)
  apply(frule DEF_X_Bag', rotate_tac 1)
  apply(frule DEF_X_Bag', clarify, simp)
  apply(subst (asm) Abs_Sequence_0_inject_charn, auto)
  apply(subst multiset_eq_conv_count_eq)
  apply(subst count_filter_inv_multiset_of)+
  apply(simp)
  done

lemmas ss_bag2seq = bag2seq_strictEq bag2seq_strongEq
```

Simplifying multiple conversions

asBag is the inverse of asSequence

```
lemma seq2bag_of_bag2seq_UC:
```

$(OclAsBag\ ((OclAsSequence\ (B::('a,'b::bot)Bag))::('a,'b::bot)Sequence)) = B$
apply(*rule Bag_sem_cases_ext, simp_all*)
apply(*simp_all add: OclAsBag_def OclAsSequence_def ss_lifting'*)
done

This blowup theorem can be quite useful for conversion proofs about functions that map from a collection type to the same collection type.

lemmas *blowup_bag2seq = seq2bag_of_bag2seq_UC[symmetric]*

From bags to bags

defs
$OclAsBag\_def : OclAsBag(self::('a, 'b::bot\ Bag\_0)\ VAL) \equiv self$

lemma *bag2bag_id* [*simp*]:
$((OclAsBag\ (B::('a,'b::bot)Bag))::('a,'b)Bag) = B$
by(*simp add: OclAsBag_def*)

From bags to ordered sets

defs

$OclAsOrderedSet\_def :$
$OclAsOrderedSet \equiv lift1\ (strictify\ (\lambda X.$
$Abs\_OrderedSet\_0\ \llcorner list\_of\_set\ (set\_of\ \ulcorner Rep\_Bag\_0\ X \urcorner)\lrcorner))$

ocl_setup_op [*OclAsOrderedSet*]

From bags to sets

defs

$OclAsSet\_def :$
$OclAsSet \equiv lift1\ (strictify\ (\lambda X.\ Abs\_Set\_0\ \llcorner set\_of\ \ulcorner Rep\_Bag\_0\ X \urcorner \lrcorner))$

ocl_setup_op [*OclAsSet*]

Building a canonic representation of bags

The empty bag

constdefs
$OclMtBag :: ('a, 'b::bot\ Bag\_0)\ VAL$
$OclMtBag \equiv lift0(Abs\_Bag\_0(\llcorner\{\#\}\lrcorner))$

syntax
$\_OclMtBag\_std :: ('a,'b::bot)\ VAL \Rightarrow 'a\ Boolean$ (`Bag{}`)
syntax
$\_OclMtBag\_ascii :: ('a,'b::bot)\ VAL \Rightarrow 'a\ Boolean$ (*mtBag*)
syntax (*xsymbols*)
$\_OclMtBag\_math :: ('a,'b::bot)\ VAL \Rightarrow 'a\ Boolean$ ($\wr\wr$)
parse_translation ⟪ *flat(map (emb trans_const)* [*OclMtBag*]) ⟫
print_translation ⟪ *map emb_print* [*OclMtBag*] ⟫

Foundational Properties of MtBag

lemma *OCL_is_isdef_OclMtBag* [*simp*]:
$\models \partial \lbrace\rbrace$
by(*simp add*: *OclValid_def OclIsDefined_def OclTrue_def*
OclMtBag_def ss_lifting')

lemma *OCL_is_defopt_OclMtBag* [*simp*]:
$\tau \models \partial \lbrace\rbrace$
by(*simp add*: *valid_elim*)

Relating the empty bag to its counterpart on Sequences

lemma *bag2seq_mtSequence_mtBag_UC*:
$([] ::('a,'b::bot) Sequence) = ((OclAsSequence (\lbrace\rbrace::('a,'b::bot) Bag)))$
by(*simp add*: *OclAsSequence_def OclMtBag_def OclMtSequence_def ss_lifting'*)

lemmas *ss_bag2seq* = *ss_bag2seq bag2seq_mtSequence_mtBag_UC*[*symmetric*]

and vice versa

lemma *seq2bag_mtBag_mtSequence_UC*:
$(\lbrace\rbrace::('a,'b::bot) Bag) = ((OclAsBag ([] ::('a,'b::bot) Sequence)))$
by(*simp add*: *OCL_Sequence.OclAsBag_def OclMtBag_def OclMtSequence_def ss_lifting'*)

lemmas *ss_seq2bag* = *seq2bag_mtBag_mtSequence_UC*[*symmetric*]

And the 'cons' operation: including

defs

OclIncluding_def :
$OclIncluding \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Bag\_0$
$(\llcorner\{\#Y\#\} + \ulcorner Rep\_Bag\_0\ X\urcorner\lrcorner))))$

ocl_setup_op [*OclIncluding*]

lemma *OCL_is_def_OclIncluding*:
$\partial(OclIncluding\ (X::('a,'b::bot) Bag)\ (Y::('a,'b::bot) VAL)) = (\partial\ X \land \partial\ Y)$
by(*rule lift2_strict_is_isdef_fw_Bag_Val*[*OF OclIncluding_def*],
simp add: *neq_commute*)

lemma *OCL_is_defopt_OclIncluding*:
$(\tau \models \partial(OclIncluding\ (X::('a,'b::bot) Bag)\ (Y::('a,'b::bot) VAL))) = ((\tau \models \partial\ X) \land (\tau \models\ \partial\ Y))$
by(*rule lift2_strictify_implies_LocalValid_defined_Bag_Val*[*OF OclIncluding_def*],
simp add: *neq_commute*)

The syntax translation mkBag builds now our bags

syntax
$@OclFinBag\quad :: args \Rightarrow ('a,'b\ Bag\_0) VAL\qquad (mkBag\{(\_)\})$

translations
  mkBag{*x*, *xs*} == *OclIncluding* (*mkBag*{*xs*}) *x*
  mkBag{*x*} == *OclIncluding OclMtBag x*

lemma *mkBag*{*1*,*1*,*1*,*0*,*1*} = *?X* **oops**

Relating including on Bags to its counterpart on sequences

lemma *seq2bag_including_UC*:
  ((*OclAsBag S*)::(′*a*,′*b*::*bot*)*Bag*) ->including *x* =
  (*OclAsBag* ((*S* ::(′*a*,′*b*::*bot*)*Sequence*) ->including (*x*::(′*a*,′*b*) *VAL*)))
  apply(*rule_tac X*=*S* **in** *Sequence_sem_cases_ext*, *simp_all*)
  apply(*case_tac x* $\tau \neq \bot$, *frule neq_commute*[*THEN iffD1*])
  apply(*simp_all add*: *OCL_Sequence.OclAsBag_def OCL_Sequence.OclIncluding_def*
      OclIncluding_def ss_lifting′ *mem_set_multiset_eq union_ac*)
  done

lemmas *ss_seq2bag* = *ss_seq2bag seq2bag_including_UC*

Singletons can be converted in any direction and don't lose information

lemma *bag2seq_singleton*:
  (([]::(′*a*,′*b*::*bot*)*Sequence*) ->including *x*) =
  (->asSequence() ((⦃⦄::(′*a*,′*b*)*Bag*) ->including (*x*::(′*a*,′*b*) *VAL*)))
  apply(*rule ext*)
  apply(*simp add*: *OclMtSequence_def OclMtBag_def OclIncluding_def OclAsSequence_def*
      OCL_Sequence.OclIncluding_def ss_lifting′)
  done

lemmas *ss_bag2seq* = *ss_bag2seq bag2seq_singleton*[*symmetric*]

Further properties of including

lemma *including_ordIndep*:
  (((*B*::(′*a*,′*b*::*bot*)*Bag*) ->including *a*) ->including *b*) =
  ((*B* ->including (*b*::(′*a*,′*b*) *VAL*)) ->including (*a*::(′*a*,′*b*) *VAL*))
  apply(*rule_tac X*=*B* **in** *Bag_sem_cases_ext*, *simp_all*)
  apply(*case_tac DEF* (*a x*), *case_tac DEF* (*a x*))
  apply(*simp add*: *localValidDefined2sem DEF_def_both*, *clarify*)
  apply(*simp_all add*: *OclIncluding_def union_ac ss_lifting*′)
  done

A second bag constructor on the OCL level

defs

  OclCollectionRange_def :
  OclCollectionRange ≡ *lift2* (*strictify* (λ *x*::*Integer_0*. *strictify* (λ *y*.
      Abs_Bag_0 ⌞*multiset_of* (*map* (λ *z*::*nat*. ⌞(*int z*) + ⌜*x*⌝⌟)
        [*0*..<(*nat* (⌜*y*⌝ − ⌜*x*⌝ + *1*))])⌟)))

ocl_setup_op [*OclCollectionRange*]

lemma *OCL_is_defopt_OclCollectionRange* [*simp*]:
$\tau \models \partial$ ((*OclCollectionRange* (*a*::('*a Integer*)) *b*)::('*a*, *Integer_0*)*Bag*) =
(($\tau \models \partial$ *a*) $\wedge$ ($\tau \models \partial$ *b*))
apply(*subgoal_tac count* (*multiset_of* (*map* (λz. $\lfloor$((*int z*) + $\ulcorner$(*a* τ)$\urcorner$)$\rfloor$) [*0*..<(*nat* (($\ulcorner$$\ulcorner$(*b* τ)$\urcorner$ − $\ulcorner$(*a* τ)$\urcorner$) + *1*))])) $\perp$ = *0*)
apply(*auto simp*: *localValidDefined2sem OclCollectionRange_def ss_lifting*′ *count_filter*)
done

Relating collectionRange on Bags to its counterpart on Sequences

lemma *seq2bag_collectionRange_UC*:
((*OclCollectionRange a b*)::('*a*,*Integer_0*)*Bag*) =
OclAsBag ((*OclCollectionRange* (*a*::'*a Integer*) (*b*::'*a Integer*))::('*a*,*Integer_0*)*Sequence*)
apply(*rule ext*)
apply(*case_tac x* $\models$ $\not\partial$ *a*, *ocl_hypsubst*, *simp*)
apply(*case_tac x* $\models$ $\not\partial$ *b*, *ocl_hypsubst*, *simp*)
apply(*simp add*: *localValidDefined2sem DEF_def_both*, *clarify*)
apply(*subgoal_tac* $\perp \notin$ *set* (*map* (λz. $\lfloor$((*int z*) + $\ulcorner$(*a x*)$\urcorner$)$\rfloor$) [*0*..<(*nat* (($\ulcorner$$\ulcorner$(*b x*)$\urcorner$ − $\ulcorner$(*a x*)$\urcorner$) + *1*))]))
apply(*subgoal_tac* $\perp \notin$ *set* (*map* (λz. $\lfloor$((*int z*) + $\ulcorner$(*a x*)$\urcorner$)$\rfloor$) [*0*..<(*nat* ($\ulcorner$$\ulcorner$(*b x*)$\urcorner$ − $\ulcorner$(*a x*)$\urcorner$))]))
apply(*auto simp*: *OCL_Sequence.OclAsBag_def OclCollectionRange_def ss_lifting*′
OCL_Sequence.OclCollectionRange_def)
done

lemmas *ss_seq2bag* = *ss_seq2bag seq2bag_collectionRange_UC*

Further properties of collectionRange

Having a cp-aware rewriter would be that nice...

lemma *collectionRange_mtBag_conv*:
⟦ $\tau \models$ (*b*::'*a Integer*) < (*a*::'*a Integer*) ⟧ $\Longrightarrow$
OclCollectionRange a b τ = (⦃⦄::('*a*,*Integer_0*)*Bag*) τ
apply(*simp add*: *ss_seq2bag*)
apply(*rule trans*, *rule_tac P*=*OclAsBag* **in** *cp_charn*)
apply(*rule OCL_Sequence.collectionRange_mtSequence_conv*)
apply(*simp_all add*: *ss_seq2bag*)
done

lemma *collectionRange_singleton_UC*[*simp*]:
OclCollectionRange a a = ⦃⦄::('*a*,*Integer_0*)*Bag*`->including` (*a*::('*a*,*Integer_0*)*VAL*)
by(*simp add*: *seq2bag_collectionRange_UC seq2bag_including_UC*[*symmetric*]
seq2bag_mtBag_mtSequence_UC)

lemma *collectionRange_expand_including*:

```
⟦ τ ⊨ (a::'a Integer) ≤ b ⟧ ⟹
  ((OclCollectionRange a b)::('a,Integer_0)Bag) τ =
  (OclCollectionRange a (b − 1)) ->including b τ
apply(simp add: ss_seq2bag)
apply(rule trans, rule_tac P=OclAsBag in cp_charn)
apply(rule OCL_Sequence.collectionRange_expand_including, simp_all)
done
```

Iterate

defs

```
OclIterate_def :
OclIterate ≡ (lift3' lift_arg0 lift_arg2 lift_arg0) (strictify (λ S P A.
                    (fold_multiset (λ x y. P y x) A (⌜Rep_Bag_0 S⌝))))
```

ocl_setup_op [*OclIterate*]

Its characterisation on the empty bag

```
lemma iterate_charn_mtBag[simp]:
  (OclIterate (⦃⦄::('a,'b::bot)Bag)
              (P::(('a ⇒ 'b) ⇒ (('a ⇒ 'c::bot) ⇒ ('a ⇒ 'c))))
               A) = A
  by(rule ext, simp add: ss_lifting' OclIterate_def OclMtBag_def)

lemma iterate_charn_including_0:
  assumes ordIndep: ∀ a b c τ. (P a (lift0 (P b c τ)) τ) = (P b (lift0 (P a c τ)) τ)
  and     defB: τ ⊨ ∂(B::('a,('b::bot))Bag)
  and     defx: τ ⊨ ∂(x::('a ⇒ 'b::bot))
  shows ((OclIterate (OclIncluding B x) P A) τ) =
        (P (lift0 (x τ)) (lift0 ((OclIterate (B::(('a,'b::bot)Bag)) P A) τ)) τ)
  apply(insert defx defB ordIndep)
  apply(simp add: localValidDefined2sem)
  apply(frule DEF_X_Bag[THEN iffD1])
  apply(frule DEF_X_Bag_0[THEN iffD1])
  apply(clarify)
  apply(simp add: OclIncluding_def OclIterate_def ss_lifting'
                  fold_multiset_union_rev
             del: fold_multiset_union)
  done

lemma iterate_charn_including:
  assumes ordIndep: ∀ a b c. (P a (P b c)) = (P b (P a c))
  and     cpP_1:    ⋀ y. cp (λ x. P x y)
  and     cpP_2:    ⋀ x. cp (P x)
  and     defB: τ ⊨ ∂(B::('a,('b::bot))Bag)
  and     defx: τ ⊨ ∂(x::('a ⇒ 'b::bot))
  shows ((OclIterate (OclIncluding B x) P A) τ ) =
```

```
      (P x (OclIterate (B::('a,'b::bot)Bag) P A) τ)
  apply(rule sym[OF trans])
  apply(rule_tac x=x in cp_subst, simp add: cpP_1)
  apply(rule trans)
  apply(rule_tac x=(B ->iterate(u;ua=A | (P u ua))) in cp_subst)
  apply(insert cpP_2, simp add: cp_by_cpify lift0_def)
  apply(rule sym[OF iterate_charn_including_0])
  apply(simp add: ordIndep cp_by_cpify)
  apply(rule defB, rule defx)
  done

lemma iterate_opt_charn_including:
  assumes ordIndep: ∀ a b c. (P a (P b c)) = (P b (P a c))
  and     P_undef_1: !!y.  P ⊥ y = ⊥
  and     P_undef_2: !!x.  P x ⊥ = ⊥
  and        cpP_1: ⋀ y. cp (λ x. P x y)
  and        cpP_2: ⋀ x. cp ((P::('a,'b::bot) VAL ⇒ ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL) x)
  shows              (OclIterate (OclIncluding B x) P A) =
                    (P x (OclIterate (B::(('a,'b::bot)Bag)) P A))
  apply(insert P_undef_1 P_undef_2 cpP_1 cpP_2 ordIndep)
  apply(rule ext)
  apply(case_tac xa ⊨ ∂ B)
  apply(case_tac xa ⊨ ∂ x)
  apply(rule iterate_charn_including, simp_all)
  apply(frule isUndefined_charn_local[THEN iffD2])
  apply(rule trans, erule_tac A=x in cp_charn, simp, simp)
  apply(rule sym[OF trans], erule_tac A=x in cp_charn, simp, simp)
  apply(frule isUndefined_charn_local[THEN iffD2])
  apply(rule trans, erule_tac A=B in cp_charn, simp, simp)
  apply(rule sym[OF trans], erule_tac A=B in cp_charn)
  apply(rule_tac P=P x in cp_compose2, simp_all)
  done
```

Forward rules for functions defined by iterate

```
lemma f_by_iterate_undef_1:
  ⟦ ⋀ B. f (B::('a,'b::bot)Bag) =
     OclIterate B (P::('a,'b::bot) VAL ⇒ ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL) A ⟧
   ⟹
   f ⊥ = ⊥
  by(simp)

lemma f_by_iterate_mtBag:
  ⟦ ⋀ B. f (B::('a,'b::bot)Bag) =
     OclIterate B (P::('a,'b::bot) VAL ⇒ ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL) A;
     ∀ a b c. (P a (P b c)) = (P b (P a c));
     ⋀ x. cp (P x) ⟧
   ⟹
   f ⦃⦄ = A
```

```
  by(simp)

lemma f_by_iterate_including:
  assumes f_by_it: ⋀ B. (f (B::('a,'b::bot)Bag)) =
                     (OclIterate B (P::('a,'b::bot) VAL ⇒ ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL) A)
  and  ordIndep: ∀ a b c. (P a (P b c)) = (P b (P a c))
  and      defB: τ ⊨ ∂(B::('a,('b::bot))Bag)
  and      defx: τ ⊨ ∂(x::('a ⇒ 'b::bot))
  and      cpP_1: ⋀ y. cp (λ x. P x y)
  and      cpP_2: ⋀ x. cp (P x)
  shows (f (OclIncluding B x) τ) = (P x (f B) τ)
  by(simp add: f_by_it iterate_charn_including ordIndep defB defx cpP_1 cpP_2)


lemma f_by_iterate_opt_including:
  assumes f_by_it: ⋀ S. (f (S::('a,'b::bot)Bag)) =
                     (OclIterate S (P::('a,'b::bot) VAL ⇒ ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL) A)
  and    ordIndep: ∀ a b c. (P a (P b c)) = (P b (P a c))
  and    P_undef_1: !!y.  P ⊥ y = ⊥
  and    P_undef_2: !!x.  P x ⊥ = ⊥
  and       cpP_1:  ⋀ y. cp (λ x. P x y)
  and       cpP_2:  ⋀ x. cp (P x)
  shows f (OclIncluding S x) = P x (f S)
  by(simp add: f_by_it iterate_opt_charn_including ordIndep
             P_undef_1 P_undef_2 cpP_1 cpP_2)
```

Conversion of iterate from bags to sequences

For iterates mapping from bags to a non-sequence

```
lemma iterate_bag2seq:
  assumes ordIndep: ∀ a b c. (P a (P b c)) = (P b (P a c))
  shows             (OclIterate (B :: (('a,('b::bot))Bag))
                          (P::('a,'b::bot) VAL ⇒  ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL)
                           A) =
                   (OclIterate ((OclAsSequence B)::('a,('b::bot))Sequence) P A)
  apply(insert ordIndep, rule ext)
  apply(simp add: OclAsSequence_def OclIterate_def OCL_Sequence.OclIterate_def
               ss_lifting)
  apply(case_tac B x ≠ ⊥)
  apply(frule DEF_X_Bag')
  apply(auto simp: fold_multiset_def)
  done

lemma iterate_seq2bag:
  assumes ordIndep: ∀ a b c. (P a (P b c)) = (P b (P a c))
  and         cpP_2: ⋀ x. cp (P x)
  shows             (OclIterate (S :: (('a,'b::bot)Sequence))
                          (P::('a,'b::bot) VAL ⇒  ('a,'c::bot) VAL ⇒ ('a,'c::bot) VAL)
```

```
                              A) =
                  (OclIterate ((OclAsBag S)::('a,'b)Bag) P A)
  apply(insert ordIndep cpP_2, rule ext)
  apply(simp add: OCL_Sequence.OclAsBag_def OclIterate_def
                OCL_Sequence.OclIterate_def ss_lifting cp_by_cpify)
  apply(case_tac S x ≠ ⊥)
  apply(frule DEF_X_Sequence', clarify)
  apply(auto simp: mem_set_multiset_eq fold_multiset_def)
  apply(rule list_of_multisetI)
  apply(erule foldl_right_order_independent)
  apply(simp)
  done
```

and for iterates mapping from bags to bags. This lemma characterises the same property on lists and multisets and serves as a guide through the proof of the theorem on bags and sequences. Because there the view is a little bit obfuscated by the contextpassingness and undefinedness cases.

```
lemma foldl_multiset2list2multiset:
  assumes   mtC_conv: e = multiset_of e'
  and       stepC_conv: f = (λ M a. multiset_of (f' (SOME xs. M = multiset_of xs) a))
  and     ordInsens_f': ⋀ xs ys a. ⟦ multiset_of xs = multiset_of ys ⟧ ⟹
                        multiset_of (f' xs a) = multiset_of (f' ys a)
  shows foldl f e xs = multiset_of (foldl f' e' xs)
  apply(simp add: mtC_conv stepC_conv)
  apply(rule sym)
  apply(rule_tac xs=xs in rev_induct)
  apply(simp_all)
  apply(rule ordInsens_f', simp)
  apply(rule someI2_ex)
  apply(rule surj_multiset_of[THEN surjD], simp)
  done
```

```
lemma iterate_bag2seq2bag_0:
  assumes ordIndepP : ∀ a b c τ. (P a (lift0 (P b c τ)) τ) = (P b (lift0 (P a c τ)) τ)
  and       mtC_conv: A = OclAsBag A'
  and     stepC_conv: ⋀ B x. P x B = OclAsBag (P' x (OclAsSequence B))
  and ordInsens_P': ⋀ S S' x τ. ⟦ (OclAsBag S τ) = ((OclAsBag S' τ):: 'c::bot Bag_0) ⟧
⟹
                               OclAsBag (P' x S) τ = ((OclAsBag (P' x S') τ):: 'c::bot Bag_0)
  shows              (OclIterate (B :: ('a,'b::bot)Bag)
                               (P::('a,'b::bot) VAL ⇒  ('a,'c::bot)Bag ⇒ ('a,'c::bot)Bag)
                                A) =
                  OclAsBag (OclIterate ((OclAsSequence B)::('a,'b::bot)Sequence)
                            (P'::('a,'b::bot) VAL ⇒ ('a,'c::bot)Sequence ⇒ ('a,'c::bot)Sequence)
                             A')
  apply(rule ext)
  apply(rule trans[symmetric])
  apply(rule_tac P=OclAsBag in cp_subst, simp add: OCL_cp_OclAsBag)
```

apply(*simp add*: *OclIterate_def OCL_Sequence.OclIterate_def ordIndepP*)

apply(*simp add*: *lift3'_def lift_arg0_def lift_arg2_def strictify_def*)
apply(*rule_tac P*=λ *t*. (*?A* $\longrightarrow$ (*?B t*) $\wedge$ *?C*) $\wedge$ *?Z* **and**
t=(`->asSequence()` *B x*)::'*b*::*bot Sequence_0* **in** *subst*)
apply(*simp add*: *OclAsSequence_def ss_lifting'*)
apply(*subgoal_tac P* = (λ *x B*. *OclAsBag* (*P' x* (*OclAsSequence B*))))
apply(*simp add*: *mtC_conv*)
prefer *2*
apply(*simp add*: *stepC_conv*[*symmetric*])
apply(*thin_tac ?Q*)
apply(*rule list_of_multisetI*)

apply(*case_tac* (*B x*) = $\perp$)
apply(*simp add*: *OclAsSequence_def OCL_Sequence.OclAsBag_def*
ss_lifting')
apply(*frule DEF_X_Bag'*, *clarify*, *simp*)
oops

lemma *iterate_bag2seq2bag*:
assumes *ordIndepP* : $\forall$ *a b c*. (*P a* (*P b c*)) = (*P b* (*P a c*))
and *cpP_2* : $\bigwedge$ *x*. *cp* (*P x*)
and *mtC_conv*: *A* = *OclAsBag A'*
and *stepC_conv*: $\bigwedge$ *B x*. *P x B* = *OclAsBag* (*P' x* (*OclAsSequence B*))
and *ordInsens_P'*: $\bigwedge$ *S S' x* τ. $[\![$ (*OclAsBag S* τ) = ((*OclAsBag S'* τ):: '*c*::*bot Bag_0*) $]\!]$
$\Longrightarrow$
OclAsBag (*P' x S*) τ = ((*OclAsBag* (*P' x S'*) τ):: '*c*::*bot Bag_0*)
shows (*OclIterate* (*B* :: ('*a*,'*b*::*bot*)*Bag*)
(*P*::('*a*,'*b*::*bot*) *VAL* $\Rightarrow$ ('*a*,'*c*::*bot*)*Bag* $\Rightarrow$ ('*a*,'*c*::*bot*)*Bag*)
A) =
OclAsBag (*OclIterate* ((*OclAsSequence B*)::('*a*,'*b*::*bot*)*Sequence*)
(*P'*::('*a*,'*b*::*bot*) *VAL* $\Rightarrow$ ('*a*,'*c*::*bot*)*Sequence* $\Rightarrow$ ('*a*,'*c*::*bot*)*Sequence*)
A')
oops

Note: If one looks at the conversion theorems about union, including and so on one sees that asBag can be taken inside. The same theorem should hold on iterate. But currently I'm not sure about its use, because in all proofs about a conversion of iterate the "reformulation" theorems (the form: bag2seq2bag) are used.

I don't know either if the "distributive" version is equally or more powerful. Therefore the following lemma after the lemmas illustrating the idea on multisets is currently left unproven.

lemma *projection_of_foldl*:
assumes *projInsens_f*: $\bigwedge$ *xs ys a*. $[\![$ *P xs* = *P ys* $]\!]$ $\Longrightarrow$
P (*f xs a*) = *P* (*f ys a*)
shows *P* (*foldl f e xs*) =

```
        foldl (λ a b. P (f (SOME ys. a = P ys) b)) (P e) xs
  apply(rule_tac xs=xs in rev_induct)
  apply(simp_all)
  apply(rule projInsens_f, simp)
  apply(rule someI2, rule sym, simp_all)
  done

lemma multiset_of_foldl:
  assumes ordInsens_f: ⋀ xs ys a. ⟦ multiset_of xs = multiset_of ys ⟧ ⟹
                          multiset_of (f xs a) = multiset_of (f ys a)
  and ordIndep_f: (∀ a b c. ((f (f a b) c) = (f (f a c) b)))
  shows multiset_of (foldl f e xs) =
        foldl (λ M a. multiset_of (f (SOME ys. M = multiset_of ys) a))
              (multiset_of e)
              (SOME ys. multiset_of xs = multiset_of ys)
  oops


lemma iterate_seq2bag':
  assumes ordIndepP : ∀ a b c. (P' a (P' b c)) = (P' b (P' a c))
   and          P'_def: P' = (λ x (B::('a,'c::bot)Bag). OclAsBag (P x ((OclAsSequence
B)::('a,'c)Sequence)))
  and        cpP_2 : ⋀ x. cp (P' x)
  and    ordInsens_P: ⋀ S S' x τ. ⟦ (OclAsBag S τ) = ((OclAsBag S' τ):: 'c::bot Bag_0) ⟧
⟹
                               OclAsBag (P x S) τ = ((OclAsBag (P x S') τ):: 'c::bot Bag_0)
  shows              (OclIterate ((OclAsBag S)::('a,'b::bot)Bag)
                                          (λ x (B::('a,'c)Bag). OclAsBag (P x ((OclAsSequence
B)::('a,'c)Sequence)))
                              (OclAsBag A)) =
                     OclAsBag (OclIterate (S::('a,'b::bot)Sequence)
                            (P::('a,'b) VAL ⇒ ('a,'c::bot)Sequence ⇒ ('a,'c::bot)Sequence)
                             A)
  oops
```

Union

defs

$$OclUnion\_def \ : \ OclUnion \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Bag\_0\ (\llcorner \ulcorner Rep\_Bag\_0\ X\urcorner + \ulcorner Rep\_Bag\_0\ Y\urcorner \lrcorner))))$$

```
ocl_setup_op [OclUnion]


lemma OCL_is_def_OclUnion:
  ∂(OclUnion (X::('a,'b::bot)Bag) Y) = (∂ X ∧ ∂ Y)
  apply(rule lift2_strict_is_isdef_fw_Bag_Bag[OF OclUnion_def])
```

apply(*simp*)
done

lemma *OCL_is_defopt_OclUnion*:
$(\tau \models \partial(OclUnion\ (X::('a,'b::bot)Bag)\ \ Y)) = ((\tau \models \partial\ X) \wedge (\tau \models\ \partial\ Y))$
apply(*rule lift2_strictify_implies_LocalValid_defined_Bag_Bag[OF OclUnion_def]*)
apply(*simp*)
done

lemma *union_seq2bag*:
`->asBag()` $((X::('a,'b::bot)Sequence) \cup Y) = (($`->asBag()` $X)::('a,'b)Bag) \cup (($`->asBag()` $Y)::('a,'b)Bag)$
apply(*rule Sequence_sem_cases_ext, simp_all*)
apply(*rule_tac X=X* **in** *Sequence_sem_cases, simp_all*)
apply(*simp_all add: OCL_Sequence.OclAsBag_def OclUnion_def mem_set_multiset_eq OCL_Sequence.OclUnion_def ss_lifting'*)
done

lemma *union_bag2seq2bag*:
`->asBag()` $(((($`->asSequence()` $(X::('a,'b::bot)Bag))::('a,'b)Sequence) \cup ($`->asSequence()` $Y))$
$= (X \cup\ Y)$
by(*simp add: union_seq2bag seq2bag_of_bag2seq_UC*)

lemma *union_by_iterate*:
$((X::('a,'b::bot)Bag) \cup (Y::('a,'b::bot)Bag)) =$
$(Y$ `->iterate`$(x;y=X \mid y$ `->including` $(x::('a,'b::bot)VAL)\ ))$
apply(*simp add: union_bag2seq2bag[symmetric] union_by_iterate*)
oops

Intersection

defs

OclIntersection_def: $OclIntersection \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Bag\_0\ (\llcorner \ulcorner Rep\_Bag\_0\ X \urcorner\ \#\cap\ \ulcorner Rep\_Bag\_0\ Y \urcorner \lrcorner))))$

ocl_setup_op [*OclIntersection*]

We'll need a second definition for the mixed case of intersection between a set and a bag ... the same case holds for union.

Includes and excludes

defs

OclIncludes_def : $OclIncludes \equiv lift2(strictify\ (\lambda X.\ strictify\ (\lambda x.\ \llcorner 0 < count\ \ulcorner Rep\_Bag\_0\ X \urcorner\ x \lrcorner)))$

OclExcludes_def : *OclExcludes* ≡ *lift2* (*strictify* (λ*X*. *strictify* (λ*x*.
⌊*count* ⌈*Rep_Bag_0 X*⌉ *x* = *0*⌋)))

ocl_setup_op [*OclIncludes*, *OclExcludes*]

We prefer a normal form where excludes is written as not includes

lemma *exludes_not_includes*[*simp*]:
x ∉ *X* = ¬ (*x*::('*a*, '*b*) *VAL*) ∈ (*X*::('*a*, '*b*::*bot*) *Bag*)
apply(*rule Bag_sem_cases_ext*, *simp_all*)
apply(*simp_all add*: *OclExcludes_def OclNot_def OclIncludes_def*
ss_lifting')
done

Relating includes and excludes on Bags to their counterparts on Sequences

lemma *includes_bag2seq*:
(*a*::('*a*,'*b*) *VAL*) ∈ (*X*::('*a*,'*b*::*bot*)*Bag*) =
a ∈ ((*OclAsSequence X*)::('*a*,'*b*::*bot*)*Sequence*)
apply(*rule ext*)
apply(*rule_tac B*=*X* **in** *OclAsSequence_charn*)
apply(*simp_all add*: *localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Bag'*, *clarify*)
apply(*simp add*: *OclIncludes_def OCL_Sequence.OclIncludes_def*
OclAsSequence_def ss_lifting' mem_set_multiset_eq)
done

lemma *excludes_bag2seq*:
(*a*::('*a*,'*b*) *VAL*) ∉ (*X*::('*a*,'*b*::*bot*)*Bag*) =
a ∉ ((*OclAsSequence X*)::('*a*,'*b*::*bot*)*Sequence*)
by(*simp add*: *includes_bag2seq*)

building a simplifier set to expand all functions on bags to the ones on sets, if it is possible.

lemmas *ss_bag2seq* = *ss_bag2seq includes_bag2seq excludes_bag2seq*

and vice versa

lemma *includes_seq2bag*:
(*a*::('*a*,'*b*) *VAL*) ∈ (*X*::('*a*,'*b*::*bot*)*Sequence*) =
a ∈ ((*OclAsBag X*)::('*a*,'*b*)*Bag*)
apply(*rule Sequence_sem_cases_ext*, *simp_all*)
apply(*simp_all add*: *OclIncludes_def OCL_Sequence.OclIncludes_def*
OCL_Sequence.OclAsBag_def ss_lifting'
mem_set_multiset_eq)
done

lemma *excludes_seq2bag*:
(*a*::('*a*,'*b*) *VAL*) ∉ (*X*::('*a*,'*b*::*bot*)*Sequence*) =

```
  a ∉ ((OclAsBag X)::('a,'b)Bag)
 by(simp add: includes_seq2bag)
```

```
lemmas ss_seq2bag = ss_seq2bag includes_seq2bag excludes_seq2bag
```

Characterisation of includes on the semantic part

```
lemma includes_charn1 :
 (τ ⊨ x ∈ S) ⟹ (0 < count ⌜Rep_Bag_0 (S τ)⌝ (x τ))
 apply(frule isDefined_if_valid)
 apply(drule OCL_is_defopt_OclIncludes[THEN iffD1])
 apply(simp add: OclLocalValid_def OclIsDefined_def strictify_def
                 OclNot_def OclIncludes_def
                 o_def lift0_def lift1_def lift2_def
                 OclTrue_def OclFalse_def  OclUndefined_def DEF_def)
 done
```

```
lemma includes_charn2 :
 ⟦ DEF(S τ); 0 < count ⌜Rep_Bag_0 (S τ)⌝ (x τ) ⟧ ⟹ τ ⊨ x ∈ S
 apply(frule DEF_X_Bag [THEN iffD1], erule exE)
 apply(simp add: OclLocalValid_def OclIsDefined_def strictify_def
                 OclNot_def OclIncludes_def
                 o_def lift0_def lift1_def lift2_def
                 OclTrue_def OclFalse_def  OclUndefined_def DEF_def)
 apply(auto)
 done
```

Higher Properties of includes and excludes

```
lemma includes_of_mtBag [simp]:
 ¬ (τ ⊨ (x::('a, 'b) VAL) ∈ (⦃⦄::('a, 'b::bot) Bag))
 by(simp only: ss_bag2seq, rule includes_of_mtSequence)
```

```
lemma includes_charn1_including[simp]:
 ⟦ τ ⊨ ∂ (a::('a,'b) VAL); τ ⊨ ∂ (X::('a,'b::bot)Bag) ⟧ ⟹
  τ ⊨ a ∈ (X ->including a)
 apply(simp only: localValidDefined2sem DEF_def)
 apply(frule DEF_X_Bag', clarify)
 apply(simp add: localValid2sem OclIncludes_def OclIncluding_def
                 ss_lifting' neq_commute)
 done
```

```
lemma includes_charn2_including[simp]:
 ⟦ τ ⊨ ∂ (a::('a,'b) VAL); τ ⊨ ∂ (b::('a,'b) VAL); τ ⊨ a ∈ (X::('a,'b::bot)Bag) ⟧ ⟹
  τ ⊨ a ∈ (X ->including b)
 apply(case_tac X τ ≠ ⊥)
 apply(frule DEF_X_Bag', clarify)
```

```
  apply(simp_all only: localValidDefined2sem DEF_def)
  apply(simp_all add: localValid2sem OclIncludes_def OclIncluding_def
                ss_lifting' neq_commute)
  apply(drule neq_commute[THEN iffD1], simp)
  done
```

It would be more favourable to have this theorem proven on sets first and lift it then to all other collections. Because it doesn't work to lift it in the other direction, from sequences to bags, sets and ordered sets.

```
lemma includes_of_collectionRange:
  (x::('a Integer)) ∈ ((OclCollectionRange a b)::('a,Integer_0)Bag) =
   ((a::('a Integer)) ≤ x) ∧̇ (x ≤ (b::('a Integer)))
  apply(rule ext)
  apply(case_tac DEF (x xa), case_tac DEF (a xa), case_tac DEF (b xa))
  apply(simp add: DEF_def_both, clarify)
  apply(subgoal_tac count (multiset_of (map (λz. ⌊((int z) + ⌈(a xa)⌉)⌋) [0..<(nat ((⌈(b xa)⌉
− ⌈(a xa)⌉) + 1))])) ⊥ = 0)
  apply(auto simp: OclIncludes_def OclCollectionRange_def OclLe_def
                 OclSand_def ss_lifting' mem_set_multiset_eq[symmetric])
  apply(arith)
  apply(rule_tac x=nat(⌈(x xa)⌉ − ⌈(a xa)⌉) in image_eqI, auto)
  apply(simp add: count_filter)
  done


lemma cp_sor [simp,intro!]: ⟦ cp P; cp P' ⟧ ⟹ cp (λ X. (P X) ∨̇ (P' X))
  by(simp add: OclSor_def)


lemma includes_of_union [simp]:
  ((a::('a,'b::bot)VAL) ∈ ((X::('a,'b::bot)Bag) ∪ Y))=
   ((a ∈ X) ∨̇ (a ∈ Y))
  apply(rule_tac X=X in Bag_sem_cases_ext, simp_all)
  apply(rule_tac X=Y in Bag_sem_cases, simp_all)
  apply(case_tac τ ⊨ ∂̸ a, ocl_hypsubst, simp add: OclSor_def)
  apply(simp add: localValidDefined2sem DEF_def_both, clarify)
  apply(simp_all add: OclIncludes_def OclUnion_def OclSor_def
                    OclSand_def OclNot_def ss_lifting')
  done

lemma not_includes_of_mtBag:
  (τ ⊨ ¬ (x::('a, 'b) VAL) ∈ (⦇⦈::('a,'b::bot) Bag)) =  (τ ⊨ ∂ x)
  by(simp only: ss_bag2seq not_includes_of_mtSequence)
```

Emptiness

defs

$OclIsEmpty\_def\ :\ OclIsEmpty \equiv lift1\ (strictify\ (\lambda X.\ \lfloor\{\#\} = \ulcorner Rep\_Bag\_0\ X\urcorner\rfloor))$

$OclNotEmpty\_def:\ OclNotEmpty \equiv lift1\ (strictify\ (\lambda X.\ \lfloor\{\#\} \neq \ulcorner Rep\_Bag\_0\ X\urcorner\rfloor))$

ocl_setup_op [*OclIsEmpty*, *OclNotEmpty*]

we prefer a canical form were notEmpty is written using isEmpty

lemma *notEmpty_not_isEmpty_conv*[*simp*]:
$\neq \emptyset(X::('a,\ 'b::bot)\ Bag) = \neg\ (\doteq \emptyset\ X)$
apply(*rule ext*)
apply(*simp add*: *OclNotEmpty_def OclIsEmpty_def OclNot_def ss_lifting'*)
done

Relating isEmpty and isNotEmpty on Bags to their counterparts on Sequences

lemma *isEmpty_bag2seq*:
$\doteq \emptyset(X::('a,\ 'b::bot)\ Bag) = \doteq \emptyset((OclAsSequence\ X)::('a,('b::bot))Sequence)$
apply(*rule ext*)
apply(*rule_tac B=X* **in** *OclAsSequence_charn*, *simp_all*)
apply(*simp_all add*: *localValidDefined2sem DEF_def*)
apply(*frule DEF_X_Bag'*, *clarify*)
apply(*simp_all add*: *OclIsEmpty_def OclMtSequence_def mem_set_multiset_eq OclStrictEq_def ss_lifting'*)
apply(*subst Abs_Sequence_0_inject_charn*)
apply(*auto simp*: *mem_set_multiset_eq*)
done

lemma *isNotEmpty_bag2seq*:
$OclNotEmpty\ (X::('a,'b::bot)Bag) = OclNotEmpty\ ((OclAsSequence\ X)::('a,('b::bot))Sequence)$
by(*simp add*: *isEmpty_bag2seq*)

lemmas *ss_bag2seq* = *ss_bag2seq isEmpty_bag2seq isNotEmpty_bag2seq*

from sequences to bags

lemma *isEmpty_seq2bag*:
$\doteq \emptyset(X::('a,\ 'b::bot)\ Sequence) = \doteq \emptyset((OclAsBag\ X)::('a,'b)Bag)$
apply(*rule Sequence_sem_cases_ext*, *simp_all*)
apply(*simp_all add*: *OclIsEmpty_def OclStrictEq_def OclMtSequence_def OCL_Sequence.OclAsBag_def ss_lifting' mem_set_multiset_eq*)
apply(*subst Abs_Sequence_0_inject_charn*)

apply(*auto simp*: *mem_set_multiset_eq*)
done

lemma *isNotEmpty_seq2bag*:
$OclNotEmpty\ (X::('a,'b::bot)Sequence) = OclNotEmpty\ ((OclAsBag\ X)::('a,('b::bot))Bag)$
by(*simp only*: *isEmpty_seq2bag notEmpty_not_isEmpty_conv OCL_Sequence.notEmpty_not_isEmpty_conv*)

lemmas *ss_seq2bag* = *ss_seq2bag isEmpty_seq2bag isNotEmpty_seq2bag*

Further properties of the emptiness test

lemma *isEmpty_stricteq_mtBag_conv*[*simp*]:
$(\doteq\emptyset\ X) = ((X::('a,'b::bot)Bag) \doteq \{\!\!\{\}\!\!\})$
apply(*rule ext*)
apply(*case_tac X x* $\neq \bot$)
apply(*frule DEF_X_Bag'*, *clarify*)
apply(*simp_all add*: *OclIsEmpty_def OclMtBag_def OclStrictEq_def*
Abs_Bag_0_inject charn ss_lifting')
apply(*blast*)
done

lemma *isEmpty_of_mtBag*:
$\tau \vDash \doteq\emptyset\ \{\!\!\{\}\!\!\}$
by (*simp*)

lemma *isEmpty_of_including*:
$[\![\ \tau \vDash \partial\ (a::('a,'b)VAL);\ \tau \vDash \partial\ (X::('a,'b::bot)Bag)\ ]\!] \Longrightarrow$
$\tau \vDash \neg\ (\doteq\emptyset(X \text{->}\texttt{including}\ a))$
oops

lemma *notEmpty_of_mtBag*:
$\tau \vDash \neg\ (\not\doteq\emptyset\ \{\!\!\{\}\!\!\})$
by(*simp*)

lemma *notEmpty_of_including*:
$[\![\ \tau \vDash \partial\ (a::('a,'b)VAL);\ \tau \vDash \partial\ (X::('a,'b::bot)Bag)\ ]\!] \Longrightarrow$
$\tau \vDash \not\doteq\emptyset\ (X \text{->}\texttt{including}\ a)$
oops

Size

defs

OclSize_def :$(OclSize::('a,\ 'b::bot\ Bag\_0)\ VAL => 'a\ Integer) \equiv$
$lift1(strictify(\lambda X.\ \llcorner int\ (size(\ulcorner Rep\_Bag\_0\ X\urcorner))\lrcorner\ ))$

ocl_setup_op [*OclSize*]

Relating size on bags to its counterpart on sequences

lemma *size_multiset_of*: (*size* (*multiset_of xs*)) = (*length xs*)
  by(*induct xs*, *auto*)

lemma *size_bag2seq*:
  $(\|(X :: (('a,('b::bot))Bag))\|) =$
   $(\|((OclAsSequence\ X)::('a,('b::bot))Sequence)\|)$
  apply(*rule ext*)
  apply(*rule_tac B=X* **in** *OclAsSequence_charn*, *simp_all*)
  apply(*simp_all add*: *localValidDefined2sem DEF_def*)
  apply(*frule DEF_X_Bag′*, *clarify*)
  apply(*auto simp*: *OclSize_def OCL_Sequence.OclSize_def*
          OclAsSequence_def mem_set_multiset_eq
          ss_lifting′ size_multiset_of)
  done

lemmas *ss_bag2seq* = *ss_bag2seq size_bag2seq*

lemma *size_seq2bag*:
  $(\|(X :: (('a,('b::bot))Sequence))\|) =$
   $(\|((OclAsBag\ X)::('a,('b::bot))Bag)\|)$
  apply(*rule Sequence_sem_cases_ext*, *simp_all*)
  apply(*auto simp*: *OclSize_def OCL_Sequence.OclSize_def*
          OCL_Sequence.OclAsBag_def size_multiset_of
          ss_lifting′ mem_set_multiset_eq)
  done

lemmas *ss_seq2bag* = *ss_seq2bag size_seq2bag*

Further properties of size

The real power of the proofs by conversion

lemma *size_by_iterate*:
  $(\|(X::('a,'b::bot)Bag)\|) =$
   $OclIterate\ X\ (\lambda\ (x::('a,'b::bot)VAL)\ (y::'a\ Integer).\ y + 1)\ 0$
  by(*simp add*: *ss_bag2seq OCL_Sequence.size_by_iterate iterate_bag2seq*)

Count

defs

$OclCount\_def \quad : OclCount \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda x.\ \llcorner int\ (count\ \ulcorner Rep\_Bag\_0\ X \urcorner\ x)\lrcorner)))$

ocl_setup_op [*OclCount*]

Relating count on bags to its counterpart on sequences

lemma *count_bag2seq*:
  $(X{::}('a,'b{::}bot)\,Bag)$ ->count $(a{::}('a,'b)\,VAL)$ =
  $(($->asSequence() $X){::}('a,'b{::}bot)\,Sequence)$ ->count a
  apply(*rule ext*)
  apply(*rule_tac B=X* **in** *OclAsSequence_charn, simp_all*)
  apply(*simp_all add: localValidDefined2sem DEF_def*)
  apply(*frule DEF_X_Bag′, clarify*)
  apply(*auto simp: OclCount_def OCL_Sequence.OclCount_def*
    OclAsSequence_def DEF_def mem_set_multiset_eq
    ss_lifting′)
  apply(*induct_tac xs, simp_all*)
  done

lemmas *ss_bag2seq = ss_bag2seq count_bag2seq*

lemma *count_seq2bag*:
  $(X{::}('a,'b{::}bot)\,Sequence)$ ->count $(a{::}('a,'b)\,VAL)$ =
  $(($->asBag() $X){::}('a,'b{::}bot)\,Bag)$ ->count a
  apply(*rule Sequence_sem_cases_ext, simp_all*)
  apply(*auto simp: OclCount_def OCL_Sequence.OclCount_def*
    OCL_Sequence.OclAsBag_def
    ss_lifting′ mem_set_multiset_eq)
  apply(*induct_tac c, simp_all*)
  done

lemmas *ss_seq2bag = ss_seq2bag count_seq2bag*

Further properties of count

lemma *count_by_iterate*:
  $((X{::}('a,'b{::}bot)\,Bag)$ ->count $((a{::}('a,'b)\,VAL)))$ =
  $(OclIterate\ X$
    $(\lambda\ (x{::}('a,'b{::}bot)\,VAL)\ (y{::}'a\ Integer).$*if* $(x \doteq a)$*then* $(y{+}1)$ *else* (y) *endif* $)$
    $($*if* $(\partial\ a)$ *then* 0 *else*$\perp$ *endif* $)\ )$
  apply(*simp add: ss_bag2seq count_by_iterate*)
  apply(*subst iterate_bag2seq, simp_all*)
  apply((*rule allI*)+, *rule ext*)
  apply(*simp add: OclIf_def plus_def One_ocl_int_def ss_lifting′*)
  done

lemma *includes_by_count_UC*[*simp*]:
  $(X$ ->count $a) > 0 = (a{::}('a,'b)\,VAL) \in (X{::}('a,'b{::}bot)\,Bag)$
  by(*simp only: ss_bag2seq OCL_Sequence.includes_by_count_UC*)

lemma *excludes_by_count_UC*[*simp*]:
  $OclCount\ S\ x \doteq 0 = \neg\ ((x{::}('a,'b)\,VAL) \in (S{::}('a,'b{::}bot)\,Sequence))$
  by(*simp only: ss_bag2seq OCL_Sequence.excludes_by_count_UC*)

Forall

defs

$OclForAll\_def$: $OclForAll \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify(\ \lambda\ S.\ (\lambda\ P.$
$\quad if\ \ \forall\, x \in set\_of\ \ulcorner Rep\_Bag\_0\ S\urcorner.\ \ P\ x = \llcorner True\lrcorner$
$\quad then\ \llcorner True\lrcorner$
$\quad else\ if\ \exists\, x \in set\_of\ \ulcorner Rep\_Bag\_0\ S\urcorner.\ P\ x = \llcorner False\lrcorner$
$\qquad then\ \llcorner False\lrcorner$
$\qquad else\ \bot)))$

ocl_setup_op [*OclForAll*]

Relating forall on bags to its counterpart on sequences

lemma *forall_bag2seq*:
$(\forall\ x \in (X :: (('a,('b::bot))Bag))\ \bullet\ P\ (x::'a \Rightarrow 'b)) =$
$(\forall\ x \in ((OclAsSequence\ X)::('a,('b::bot))Sequence)\ \ \bullet\ P\ x)$
apply(*rule ext*)
apply(*rule_tac B=X* **in** *OclAsSequence_charn, simp_all*)
apply(*simp_all add: OclForAll_def OCL_Sequence.OclForAll_def OclUndefined_def OclAsSequence_def mem_set_multiset_eq cp_by_cpify ss_lifting' localValidDefined2sem*)
done

lemmas *ss_bag2seq* = *ss_bag2seq forall_bag2seq*

Further properties of forall

lemma *forall_by_iterate*:
$(\forall\ x \in (B::('a,'b::bot)Bag)\ \bullet\ (P::('a,'b::bot)VAL \Rightarrow\ 'a\ Boolean)\ x) =$
$(B\ \text{->iterate}(x;y = \mathsf{T}\ |\ (P\ x) \wedge y))$
apply(*simp add: forall_by_iterate ss_bag2seq*)
apply(*subst iterate_bag2seq, simp_all add: and_assoc*)
apply(*simp add: and_commute*)
done

Exists

Relating exists on bags to its counterpart on sequences

lemma *exists_bag2seq*:
$(\exists\ x \in (B :: (('a,('b::bot))Bag))\ \bullet\ P\ (x::'a \Rightarrow 'b)) =$
$(\exists\ x \in ((OclAsSequence\ B)::('a,('b::bot))Sequence)\ \ \bullet\ P\ x)$
by(*simp add: OclExists_def forall_bag2seq*)

lemmas *ss_bag2seq* = *ss_bag2seq exists_bag2seq*

Further properties of exists

lemma *exists_by_iterate*:
$(\exists\ x \in (B::('a,'b::bot)Bag)\ \bullet\ (P::('a,'b::bot)VAL \Rightarrow\ 'a\ Boolean)\ x) =$
$(B\ \text{->iterate}(x;y = \mathsf{F}\ |\ (P\ x) \vee y))$
apply(*simp add: exists_by_iterate ss_bag2seq*)

```
apply(subst iterate_bag2seq, simp_all)
apply(subst or_commute)
apply(simp add: or_assoc)
apply(simp add: or_commute)
done
```

IncludesAll

defs

```
  OclIncludesAll_def: OclIncludesAll ≡ lift2 (strictify (λX. strictify (λY.
                                  ⌊set_of ⌈Rep_Bag_0 Y⌉ ⊆ set_of ⌈Rep_Bag_0 X⌉⌋)))
```

```
ocl_setup_op [OclIncludesAll]
```

Relating includesAll on bags to its counterpart ond sequences

```
lemma includesAll_bag2seq:
  OclIncludesAll (X::('a,'b::bot)Bag) (Y::('a,'b)Bag) =
   OclIncludesAll ((OclAsSequence X)::('a,'b)Sequence) ((OclAsSequence Y)::('a,'b)Sequence)
  apply(rule ext)
  apply(rule_tac B=X in OclAsSequence_charn, simp_all)
  apply(rule_tac B=Y in OclAsSequence_charn, simp_all)
  apply(simp add: OclIncludesAll_def OCL_Sequence.OclIncludesAll_def
                  OclAsSequence_def mem_set_multiset_eq
                  localValidDefined2sem ss_lifting')
  done
```

```
lemmas ss_bag2seq = ss_bag2seq includesAll_bag2seq
```

Further properties of includesAll

```
lemma includesAll_by_iterate:
  OclIncludesAll X (Y::('a,'b::bot)Bag) =
   (Y->iterate(x;y= if (∂ X) then ⊤ else ⊥ endif | ((x::('a,'b)VAL) ∈ X) ∧̇ y))
  apply(simp add: includesAll_by_iterate ss_bag2seq)
  apply(subst iterate_bag2seq)
  apply(simp_all add: OCL_is_def_OclAsSequence)
  apply(auto, rule ext, auto simp: OclSand_def ss_lifting')
  done
```

ExcludesAll

defs

```
  OclExcludesAll_def: OclExcludesAll ≡ lift2 (strictify (λX. strictify (λY.
                              ⌊set_of ⌈Rep_Bag_0 X⌉ ∩ set_of ⌈Rep_Bag_0 Y⌉ = {}⌋)))
```

```
ocl_setup_op [OclExcludesAll]
```

Relating excludesAll on bags to its counterpart ond sequences

lemma *excludesAll_bag2seq*:
  $OclExcludesAll\ (X::('a,'b::bot)Bag)\ (Y::('a,'b)Bag) =$
  $OclExcludesAll\ ((OclAsSequence\ X)::('a,'b)Sequence)\ ((OclAsSequence\ Y)::('a,'b)Sequence)$
  apply(*rule ext*)
  apply(*rule_tac B=X* **in** *OclAsSequence_charn, simp_all*)
  apply(*rule_tac B=Y* **in** *OclAsSequence_charn, simp_all*)
  apply(*simp add: OclExcludesAll_def OCL_Sequence.OclExcludesAll_def*
  OclAsSequence_def mem_set_multiset_eq
  localValidDefined2sem ss_lifting')
  done

lemmas *ss_bag2seq = ss_bag2seq excludesAll_bag2seq*

Further properties of excludesAll

lemma *excludesAll_by_iterate*:
  $OclExcludesAll\ X\ (Y::('a,'b::bot)Bag) =$
  $(Y\text{->iterate}(x;y= \mathit{if}\ (\partial\ X)\ \mathit{then}\ \top\ \mathit{else}\ \bot\ \mathit{endif} \mid ((x::('a,'b)VAL) \notin X)\ \dot{\wedge}\ y))$
  apply(*simp add: excludesAll_by_iterate ss_bag2seq*)
  apply(*subst iterate_bag2seq*)
  apply(*simp_all add: OCL_is_def_OclAsSequence*)
  apply(*auto, rule ext, auto simp: OclSand_def ss_lifting'*)
  done

Select

defs

$$
\begin{aligned}
OclSelect\_def:\ OclSelect \equiv\ & (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.\\
& if\quad \forall\ x \in set\_of\ \ulcorner Rep\_Bag\_0\ S \urcorner.\ DEF\ (P\ x)\\
& then\ \ Abs\_Bag\_0(\llcorner MCollect\ \ulcorner Rep\_Bag\_0\ S \urcorner\ (\lambda\ x.\ P\ x = \llcorner True \lrcorner)\lrcorner)\\
& else\ \ \bot))
\end{aligned}
$$

ocl_setup_op [*OclSelect*]

Relating select on bags to its counterpart on sequences

lemma *select_bag2seq2bag*:
  $(OclSelect\ (X::('a,'b::bot)Bag)\ (P::('a,'b)VAL \Rightarrow 'a\ Boolean)) =$
  $(OclAsBag\ (OclSelect\ ((OclAsSequence\ X)::('a,'b)Sequence)\ P))$
  apply(*rule ext*)
  apply(*case_tac X x* $\neq \bot$)
  apply(*frule DEF_X_Bag', clarify*)
  apply(*auto simp: OclAsSequence_def OCL_Sequence.OclAsBag_def*
  OclSelect_def OCL_Sequence.OclSelect_def
  ss_lifting')
  done

lemmas *ss_bag2seq2bag = select_bag2seq2bag*

Further properties of select

lemma *select_by_iterate*:
$(\{x : (B::('a,'b::bot) Bag) \mid ((P::('a,'b::bot) VAL \Rightarrow 'a\ Boolean)\ x)\}) =$
$(B \text{->iterate}(x;\ y = \{\} \mid$
$\quad if\ (P\ x) then\ (y \text{->including}\ x)\ else\ y\ endif\ ))$

oops

Reject

defs

$OclReject\_def$: $OclReject \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
$if\ \ \forall\ x \in set\_of\ \ulcorner Rep\_Bag\_0\ S \urcorner.\ DEF\ (P\ x)$
$then\ \ Abs\_Bag\_0(\llcorner MCollect\ \ulcorner Rep\_Bag\_0\ S \urcorner\ (\lambda\ x.\ P\ x = \llcorner False \lrcorner) \lrcorner)$
$else\ \ \bot))$

ocl_setup_op [*OclReject*]

Excluding

defs

$OclExcluding\_def$:
$OclExcluding \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Bag\_0$
$(\llcorner\ MCollect\ \ulcorner Rep\_Bag\_0\ X \urcorner\ (\lambda\ a.\ a \neq Y)\ \lrcorner))))$

ocl_setup_op [*OclExcluding*]

Collect

Note currently this collect corresponds to a collectNested. This will be fixed later.

defs

$OclCollect\_def$: $OclCollect\ S\ P \equiv\ \lambda\ \tau.$
$if\ DEF\ (S\ \tau)$
$then\ if\ \forall\ x \in set\_of\ \ulcorner Rep\_Bag\_0\ (S\ \tau) \urcorner.\ DEF\ (P\ (\lambda \tau.\ x)\ \tau)$
$then\ Abs\_Bag\_0\ \llcorner Abs\_multiset\ ((count\ \ulcorner Rep\_Bag\_0\ (S\ \tau) \urcorner)\ o\ (inv\ (\lambda$
$x.\ P\ (lift0\ x)\ \tau))) \lrcorner$
$else\ \bot$
$else\ \bot$

ocl_setup_op [*OclCollect*]

One

defs

$OclOne\_def$: $OclOne\ (S::('a,('b::bot)\ Bag\_0) VAL)\ P\ \equiv$

$\|((\!|x : S \mid P\ (x{::}'a{::}type \Rightarrow {}'b{::}bot)|\!))\| \doteq 1$

ocl_setup_op [*OclOne*]

Sum

Like collect sum is specified as mapping to other ocl functions ...this is because for every type supporting *plus*, *zero* the sum function must be available and it make no sense to add a new definition for every one of these

defs

$$\begin{aligned}&OclSum\_def &&:\\ &OclSum\ S &&\equiv OclIterate\ (S{::}('a,'b{::}\{bot,plus,zero\})Bag)\ (\lambda\ x\ y.\ x + y)\ 0\end{aligned}$$

Complement

defs

$$OclComplement\_def{:}OclComplement \equiv lift1\ (strictify(\lambda X.\ Abs\_Bag\_0\ (\llcorner -(\ulcorner Rep\_Bag\_0\ X\urcorner - \{\#\bot\#\})\lrcorner)))$$

ocl_setup_op[*OclComplement*]

Stuff that should be moved into the right theory

end

B.4.12. OCL OrderedSet

theory *OCL_OrderedSet*
imports
$HOLOCL_HOME/src/library/collection/smashed/OCL_Sequence
begin

Properties of the Congruence inside the Datatype Adaption

The following rules are transformed versions of the automatically generated rules for datatype conversions.

lemma *Abs_OrderedSet_inject_charn*:
$[\![\ (\bot{::}'a{::}bot) \notin set\ x;\ (\bot{::}'a{::}bot) \notin set\ y;\ distinct\ x;\ distinct\ y\ ]\!] \Longrightarrow$
$(((Abs\_OrderedSet\_0\ \llcorner x\lrcorner) = (Abs\_OrderedSet\_0\ \llcorner y\lrcorner)) = (x = y))$
by(*subst Abs_OrderedSet_0_inject,*
simp_all add: OrderedSet_0_def smash_def)

```
lemma Abs_OrderedSet_0_inject_absurd11 [simp]:
  ⟦ ⊥ ∉ set x; distinct x ⟧ ⟹ ((Abs_OrderedSet_0 down) = (Abs_OrderedSet_0 ⌊x⌋)) =
False
  by(subst Abs_OrderedSet_0_inject,
     simp_all add: OrderedSet_0_def smash_def)

lemma Abs_OrderedSet_0_inject_absurd12 [simp]:
  ⟦ ⊥ ∉ set x; distinct x ⟧ ⟹ ((Abs_OrderedSet_0 ⌊x⌋) = (Abs_OrderedSet_0 down)) =
False
  by(subst Abs_OrderedSet_0_inject,
     simp_all add: OrderedSet_0_def smash_def)

lemma Abs_OrderedSet_0_inject_absurd21 [simp]:
  ⟦ ⊥ ∉ set x; distinct x ⟧ ⟹ ( ⊥ = (Abs_OrderedSet_0 ⌊x⌋)) = False
  by(simp add: UU_OrderedSet_def)

lemma Abs_OrderedSet_0_inject_absurd22 [simp]:
  ⟦ ⊥ ∉ set x; distinct x ⟧ ⟹ ((Abs_OrderedSet_0 ⌊x⌋) = ⊥) = False
  by(simp add: UU_OrderedSet_def)

lemma Abs_OrderedSet_0_inject_charn:
  ⟦ (⊥::'a::bot) ∉ set x; (⊥::'a::bot) ∉ set y; distinct x; distinct y ⟧ ⟹
   (((Abs_OrderedSet_0 ⌊x⌋) = (Abs_OrderedSet_0 ⌊y⌋)) = (x = y))
  by(subst Abs_OrderedSet_0_inject,
     simp_all add: OrderedSet_0_def smash_def)

lemma Abs_OrderedSet_0_inverse_charn1 [simp]:
  ⟦ (⊥ ∉ set x); distinct x ⟧ ⟹ Rep_OrderedSet_0 (Abs_OrderedSet_0 ⌊x⌋) = ⌊x⌋
  by (subst Abs_OrderedSet_0_inverse,
      simp_all add: OrderedSet_0_def smash_def)

lemma Abs_OrderedSet_0_inverse_charn2 [simp]:
  ⟦ (⊥ ∉ set x); distinct x ⟧ ⟹ ⌈Rep_OrderedSet_0 (Abs_OrderedSet_0 ⌊x⌋)⌉ = x
  by (subst Abs_OrderedSet_0_inverse,
      simp_all add: OrderedSet_0_def smash_def)

lemma Abs_OrderedSet_0_cases_charn:
  assumes bottomCase : ⟦ x = ⊥ ⟧ ⟹ P
  assumes listCase   : ⋀y. (⟦ x = Abs_OrderedSet_0 ⌊y⌋; ⊥ ∉ set y; distinct y ⟧ ⟹ P)
  shows                P
  apply(rule_tac x=x in Abs_OrderedSet_0_cases)
  apply(case_tac y = ⊥)
  apply(rule bottomCase, simp add: UU_OrderedSet_def)
  apply(frule not_down_exists_lift2[THEN iffD1])
  apply(rule_tac y=⌈y⌉ in listCase)
  apply(auto simp: OrderedSet_0_def smash_def)
  apply(case_tac distinct ya, simp_all)
  done
```

```
lemma Abs_OrderedSet_0_induct_charn:
  assumes bottomCase : P ⊥
  assumes stepCase   : ⋀y. ([[ ⊥ ∉ set y; distinct y ]] ⟹ P (Abs_OrderedSet_0 ⌊y⌋))
  shows                P x
  apply(rule_tac x=x in Abs_OrderedSet_0_induct)
  apply(rule_tac  x=Abs_OrderedSet_0 y in Abs_OrderedSet_0_cases_charn)
  apply(simp_all add: bottomCase stepCase)
  done

lemma inj_on_Abs_OrderedSet: inj_on Abs_OrderedSet_0 OrderedSet_0
  by(rule inj_on_inverseI,rule OrderedSet_0.Abs_OrderedSet_0_inverse,assumption)

lemma inj_Rep_OrderedSet : inj Rep_OrderedSet_0
  by(rule inj_on_inverseI, rule OrderedSet_0.Rep_OrderedSet_0_inverse)

lemma smashed_orderedSet_charn:
  (⊥ ∉ set X ∧ distinct X) = ((⌊X⌋) ∈ OrderedSet_0)
  by(unfold smash_def OrderedSet_0_def UU_OrderedSet_def,auto)

lemma UU_in_smashed_orderedSet [simp]:
  ⊥ ∈ OrderedSet_0
  by(unfold smash_def OrderedSet_0_def UU_OrderedSet_def,auto)

lemma down_in_smashed_orderedSet [simp]:
  down ∈ OrderedSet_0
  by(unfold smash_def OrderedSet_0_def UU_OrderedSet_def,auto)

lemma mt_in_smashed_orderedSet[simp]:
  (⌊[]⌋) ∈ OrderedSet_0
  by (unfold smash_def OrderedSet_0_def,auto)

lemma DEF_Abs_OrderedSet: ⋀X. [[⊥ ∉ set X; distinct X ]] ⟹ DEF (Abs_OrderedSet_0
(⌊X⌋))
  apply(unfold DEF_def UU_OrderedSet_def, simp)
  done

lemma DEF_Rep_OrderedSet:
  ⋀X. DEF X ⟹ DEF (Rep_OrderedSet_0 X)
  apply(unfold DEF_def UU_OrderedSet_def,auto)
  apply(drule_tac f = Abs_OrderedSet_0 in arg_cong)
  apply(simp add: Rep_OrderedSet_0_inverse)
  done

lemma not_DEF_Rep_OrderedSet:
  ⋀X. ¬ DEF X ⟹ ¬ DEF (Rep_OrderedSet_0 X)
  apply(unfold DEF_def UU_OrderedSet_def,auto)
  apply(simp add: Abs_OrderedSet_0_inverse)
  done
```

lemma *exists_lift_OrderedSet*:
$\bigwedge X.\ DEF\ X \Longrightarrow \exists\, c.\ Rep\_OrderedSet\_0\ X = (\llcorner c \lrcorner)$
by (*drule DEF_Rep_OrderedSet,simp add: DEF_X_up*)

lemma *exists_lift_OrderedSet2*:
$\bigwedge X.\ DEF\ X \Longrightarrow \exists\, c.\ \bot \notin set\ c \wedge distinct\ c \wedge Rep\_OrderedSet\_0\ X = (\llcorner c \lrcorner)$
apply(*frule exists_lift_OrderedSet*)
apply(*clarify, simp*)
apply(*subgoal_tac* $\llcorner c \lrcorner \in OrderedSet\_0$)
apply(*erule smashed_orderedSet_charn[THEN iffD2]*)
apply(*drule sym, simp add: Rep_OrderedSet_0*)
done

lemma *Rep_OrderedSet_cases*:
$Rep\_OrderedSet\_0\ X = \bot \vee (\exists\, c.\ \bot \notin set\ c \wedge distinct\ c \wedge Rep\_OrderedSet\_0\ X = (\llcorner c \lrcorner))$
apply(*case_tac DEF X*)
apply(*drule exists_lift_OrderedSet2*)
apply(*drule_tac [2] not_DEF_Rep_OrderedSet*)
apply(*simp_all add: DEF_def*)
done

lemma *DEF_X_OrderedSet_0* : $DEF(X) = (\exists\, c.\ \bot \notin set\ c \wedge distinct\ c \wedge Rep\_OrderedSet\_0\ X = (\llcorner c \lrcorner))$
apply(*insert Rep_OrderedSet_cases [of X], auto*)
apply(*drule exists_lift_OrderedSet,auto*)
apply(*rule swap*)
prefer *2*
apply(*rule not_DEF_Rep_OrderedSet, auto*)
done

The following lemma is crucial for the theory morpher:

lemma *DEF_X_OrderedSet*: $DEF\ X = (\exists\, c.\ \bot \notin set\ c \wedge distinct\ c \wedge X = Abs\_OrderedSet\_0\ (\llcorner c \lrcorner))$
apply (*auto simp: DEF_Abs_OrderedSet*)
apply (*simp add:DEF_X_OrderedSet_0*)
apply (*erule exE, rule exI, auto*)
apply (*rule injD [OF inj_Rep_OrderedSet]*)
apply (*auto simp: smashed_orderedSet_charn Abs_OrderedSet_0_inverse*)
done

This lemma is very convenient in simplifing unfolded definitions

lemma *DEF_X_OrderedSet'*:
$[\![\ X \neq \bot\ ]\!] \Longrightarrow$
$(\exists\, c.\ (\bot \notin set\ c) \wedge (distinct\ c) \wedge (Rep\_OrderedSet\_0\ X = \llcorner c \lrcorner)) \wedge$
$(\exists\, c.\ (\bot \notin set\ c) \wedge (distinct\ c) \wedge (X = (Abs\_OrderedSet\_0\ \llcorner c \lrcorner)))$
apply(*fold DEF_def*)

```
  apply(frule DEF_X_OrderedSet[THEN iffD1])
  apply(drule DEF_X_OrderedSet_0[THEN iffD1])
  apply(simp)
  done


lemma OrderedSet_sem_cases_0:
  assumes defC: ⋀ c d. ⟦ X ≠ ⊥; ⊥ ≠ X;
                        (⊥ ∉ set c); (⊥ ∉ set d);
                        distinct c; distinct d;
                        Rep_OrderedSet_0 X = ⌊c⌋;
                        X = Abs_OrderedSet_0 ⌊d⌋ ⟧ ⟹ P X
  and   undefC: ⟦ X = ⊥ ⟧ ⟹ P X
  shows P X
  apply(rule Abs_OrderedSet_0_cases_charn, erule undefC)
  apply(rule defC, auto)
  done

lemma OrderedSet_sem_cases:
  assumes defC: ⋀ c d. ⟦ (X τ) ≠ ⊥; ⊥ ≠ (X τ);
                        (⊥ ∉ set c); (⊥ ∉ set d);
                        distinct c; distinct d;
                        Rep_OrderedSet_0 (X τ) = ⌊c⌋;
                        (X τ) = Abs_OrderedSet_0 ⌊d⌋ ⟧ ⟹ P X τ
  and   undefC: ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ
  and     cpP: cp P
  shows P X τ
  apply(rule_tac P2=P in subst[OF sym[OF cp_subst]], rule cpP)
  apply(rule OrderedSet_sem_cases_0)
  apply(rule_tac P1=P in subst[OF cp_subst], rule cpP)
  apply(rule defC) prefer 9
  apply(rule_tac P1=P in subst[OF cp_subst], rule cpP)
  apply(rule undefC, simp_all)
  done

lemma OrderedSet_sem_cases_ext:
  assumes defC: ⋀ c d τ. ⟦ (X τ) ≠ ⊥; ⊥ ≠ (X τ);
                        (⊥ ∉ set c); (⊥ ∉ set d);
                        distinct c; distinct d;
                        Rep_OrderedSet_0 (X τ) = ⌊c⌋;
                        (X τ) = Abs_OrderedSet_0 ⌊d⌋ ⟧ ⟹ P X τ = Q X τ
  and   undefC: ⋀ τ. ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ = Q X τ
  and     cpP: cp P
  and     cpQ: cp Q
  shows P X = Q X
  apply(rule ext)
  apply(rule_tac X=X in OrderedSet_sem_cases)
  apply(rule defC) prefer 9
  apply(rule undefC)
```

```
  apply(simp_all add: cpP cpQ)
  done
```

These four lemmas are used by ocl_setup_op to reason about definedness:

```
lemma lift2_strict_is_isdef_fw_OrderedSet_Val:
 assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_OrderedSet_0 ⌊g ⌈Rep_OrderedSet_0
x⌉ y⌋)))
  and     inv_g1: !!a b.( ⟦ ⊥ ∉ set a; distinct a; (⊥::'a::bot) ≠ b ⟧ ⟹ ⊥ ∉ set (g a b))
  and     inv_g2: !!a b.( ⟦ ⊥ ∉ set a; distinct a; (⊥::'a::bot) ≠ b ⟧ ⟹ distinct (g a b))
  shows           ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(rule ext)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                o_def ss_lifting')
  apply(rule_tac X=X x in OrderedSet_sem_cases_0)
  apply(rule impI, frule_tac b=(Y x) in  inv_g1)
  apply(drule_tac[3] b=(Y x) in inv_g2)
  apply(simp_all add: neq_commute)
  done

lemma lift2_strict_is_isdef_fw_OrderedSet_OrderedSet:
 assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_OrderedSet_0 ⌊g ⌈Rep_OrderedSet_0
x⌉ ⌈Rep_OrderedSet_0 y⌉⌋)))
  and     inv_g1: !!a b. ⟦ ⊥ ∉ set a; distinct a; ⊥ ∉ set b; distinct b ⟧ ⟹ ⊥ ∉ set (g a b)
  and     inv_g2: !!a b. ⟦ ⊥ ∉ set a; distinct a; ⊥ ∉ set b; distinct b ⟧ ⟹ distinct (g a b)
  shows           ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(rule ext)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                o_def ss_lifting')
  apply(rule_tac X=X x in OrderedSet_sem_cases_0)
  apply(rule_tac X=Y x in OrderedSet_sem_cases_0)
  apply(rule impI, frule_tac b=ca in  inv_g1)
  apply(drule_tac[4] b=ca in inv_g2)
  apply(simp_all add: neq_commute)
  done

lemma lift2_strictify_implies_LocalValid_defined_OrderedSet_Val:
 assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_OrderedSet_0 ⌊g ⌈Rep_OrderedSet_0
x⌉ y⌋)))
  and     inv_g1: !!a b.( ⟦ ⊥ ∉ set a; (⊥::'a::bot) ≠ b⟧ ⟹ ⊥ ∉ set (g a b))
  and     inv_g2: !!a b.( ⟦ ⊥ ∉ set a; (⊥::'a::bot) ≠ b⟧ ⟹ distinct (g a b))
  shows           (τ ⊨ ∂(f X Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
  apply(insert f_def)
  apply(drule_tac X=X and  Y=Y in lift2_strict_is_isdef_fw_OrderedSet_Val)
  apply(assumption)+
  apply(simp_all add: OclAnd_def  OclTrue_def o_def
              OclIsDefined_def lift0_def lift1_def lift2_def
              OclLocalValid_def)
  done
```

lemma *lift2_strictify_implies_LocalValid_defined_OrderedSet_OrderedSet*:
 assumes *f_def*: $f \equiv lift2(strictify(\lambda x.\ strictify(\lambda y.\ Abs\_OrderedSet\_0\ \llcorner g \ulcorner Rep\_OrderedSet\_0\ x \urcorner\ \ulcorner Rep\_OrderedSet\_0\ y \urcorner \lrcorner)))$
 and *inv_g1*: $!!a\ b.(\ [\![\ \bot \notin set\ a;\ distinct\ a;\ \bot \notin set\ b;\ distinct\ b\ ]\!] \Longrightarrow \bot \notin set\ (g\ a\ b))$
 and *inv_g2*: $!!a\ b.(\ [\![\ \bot \notin set\ a;\ distinct\ a;\ \bot \notin set\ b;\ distinct\ b\ ]\!] \Longrightarrow distinct\ (g\ a\ b))$
 shows $(\tau \vDash \partial(f\ X\ Y)) = ((\tau \vDash \partial\ X) \wedge (\tau \vDash\ \partial\ Y))$
 apply(*insert f_def*)
 apply(*drule_tac X=X* **and** *Y=Y* **in** *lift2_strict_is_isdef_fw_OrderedSet_OrderedSet*)
 apply(*assumption*)+
 apply(*simp add: OclAnd_def OclTrue_def o_def*
   OclIsDefined_def lift0_def lift1_def lift2_def
   OclLocalValid_def)
 done

Building a canonic representation of orderedSets

The empty orderedSet

constdefs
  OclMtOrderedSet :: ('a, 'b::bot OrderedSet_0) VAL
  OclMtOrderedSet ≡ lift0(Abs_OrderedSet_0(⌞[]⌟))

  syntax
  _OclMtOrderedSet_std :: ('a,'b::bot) VAL ⇒ 'a Boolean   (`OrderedSet{}`)
  syntax
  _OclMtOrderedSet_ascii :: ('a,'b::bot) VAL ⇒ 'a Boolean   (*mtOrderedSet*)
  syntax (*xsymbols*)
  _OclMtOrderedSet_math :: ('a,'b::bot) VAL ⇒ 'a Boolean   (⟨⟩)
  parse_translation ⟪ flat(map (emb trans_const) [OclMtOrderedSet]) ⟫
  print_translation ⟪ map emb_print [OclMtOrderedSet] ⟫

lemma *OCL_is_isdef_OclMtOrderedSet* [*simp*]:
 ⊨ ∂ ⟨⟩
 by(*simp add: OclValid_def OclIsDefined_def OclTrue_def*
   OclMtOrderedSet_def ss_lifting')

lemma *OCL_is_defopt_OclMtOrderedSet* [*simp*]:
 τ ⊨ ∂ ⟨⟩
 by(*simp add: valid_elim*)

And the 'cons' operation: including

defs

  OclIncluding_def :
  OclIncluding ≡ $lift2(strictify(\lambda S.\ strictify\ (\lambda e.\ Abs\_OrderedSet\_0\ (\llcorner remdups\ (concat\ [\ulcorner Rep\_OrderedSet\_0\ S \urcorner,\ [e]\ ])\lrcorner))))$

ocl_setup_op [*OclIncluding*]

lemma *OCL_is_def_OclIncluding*:
∂(*OclIncluding* (*X*::('*a*,'*b*::*bot*) *OrderedSet*) (*Y*::('*a*,'*b*::*bot*) *VAL*)) = (∂ *X* ∧ ∂ *Y*)
by(*rule lift2_strict_is_isdef_fw_OrderedSet_Val*[*OF OclIncluding_def*], *simp_all*)

lemma *OCL_is_defopt_OclIncluding*:
(τ ⊨ ∂(*OclIncluding* (*X*::('*a*,'*b*::*bot*) *OrderedSet*) (*Y*::('*a*,'*b*::*bot*) *VAL*))) = ((τ ⊨ ∂ *X*) ∧ (τ ⊨ ∂ *Y*))
by(*rule lift2_strictify_implies_LocalValid_defined_OrderedSet_Val*[*OF OclIncluding_def*], *simp_all*)

The syntax translation mkOrderedSet builds now our orderedSets

syntax
@*OclFinOrderedSet* :: *args* ⇒ ('*a*,'*b* *OrderedSet_0*) *VAL* (*mkOrderedSet*[(_)])

translations
mkOrderedSet[*x*, *xs*] == *OclIncluding* (*mkOrderedSet*[*xs*]) *x*
mkOrderedSet[*x*] == *OclIncluding OclMtOrderedSet x*

lemma *test* : *mkOrderedSet*[*1*,*2*,*3*,*4*,*5*] = *?X* **oops**

Relation of mtOrderedSet and including

These two theorems should actually suffice to decide inequality of orderedSets

lemma *including_notstrongeq_mtOrderedSet*[*simp*]:
(¬ ((*X*::('*a*,'*b*::*bot*) *OrderedSet*) ->including (*a*::('*a*,'*b*) *VAL*) ≜ ⟨⟩)) = T
apply(*rule_tac X*=*X* **in** *OrderedSet_sem_cases_ext*)
apply(*simp_all add*: *cp_strongEq*)
apply(*simp_all add*: *OclIncluding_def OclMtOrderedSet_def OclStrongEq_def*
OclTrue_def OclNot_def localValid2sem ss_lifting'
Abs_OrderedSet_inject_charn neq_commute)
done

lemma *including_notstricteq_mtOrderedSet*[*simp*]:
⟦ τ ⊨ ∂ (*a*::('*a*,'*b*) *VAL*); τ ⊨ ∂ (*X*::('*a*,'*b*::*bot*) *OrderedSet*) ⟧ ⟹
τ ⊨ (*X* ->including *a*) ‘<>‘ ⟨⟩
apply(*rule_tac X*=*X* **in** *OrderedSet_sem_cases*)
apply(*simp_all add*: *cp_strongEq cp_strictEq localValidDefined2sem*)
apply(*simp_all add*: *OclIncluding_def OclMtOrderedSet_def OclStrictEq_def*
OclTrue_def OclNot_def localValid2sem ss_lifting'
Abs_OrderedSet_inject_charn neq_commute)
done

A second orderedSet constructor on the OCL level

defs

$OclCollectionRange\_def$:
$OclCollectionRange \equiv lift2\ (strictify\ (\lambda\ x{::}Integer\_0.\ strictify\ (\lambda\ y.$
$\quad Abs\_OrderedSet\_0\ \llcorner map\ (\lambda\ z{::}nat.\ \llcorner(int\ z) + \ulcorner x \urcorner \lrcorner)$
$\quad [0..<(nat\ (\ulcorner y \urcorner - \ulcorner x \urcorner + 1))] \lrcorner)))$

ocl_setup_op [$OclCollectionRange$]

lemma $OCL\_is\_defopt\_OclCollectionRange$ [$simp$]:
$\tau \vDash \partial\ ((OclCollectionRange\ (a{::}('a\ Integer))\ b){::}('a,\ Integer\_0)OrderedSet) =$
$((\tau \vDash \partial\ a) \land (\tau \vDash \partial\ b))$
oops

lemma $collectionRange\_mtOrderedSet\_conv$:
$[\![\ \tau \vDash (b{::}'a\ Integer) < (a{::}'a\ Integer)\ ]\!] \Longrightarrow$
$OclCollectionRange\ a\ b\ \tau = (\langle\rangle{::}('a,Integer\_0)OrderedSet)\ \tau$
apply($case\_tac\ \tau \vDash \not\partial\ a,\ ocl\_hypsubst,\ simp$)
apply($case\_tac\ \tau \vDash \not\partial\ b,\ ocl\_hypsubst,\ simp$)
apply($simp\ add{:}\ localValidDefined2sem\ DEF\_def\_both,\ clarify$)
apply($simp\_all\ add{:}\ DEF\_def\ OclCollectionRange\_def\ OclMtOrderedSet\_def$
$\quad OclLess\_def\ localValid2sem\ ss\_lifting'$)
done

lemma $collectionRange\_singleton\_UC$[$simp$]:
$OclCollectionRange\ a\ a = \langle\rangle{::}('a,Integer\_0)OrderedSet$ `->including` $(a{::}('a,Integer\_0)\ VAL)$
apply($rule\ ext,\ case\_tac\ x \vDash \not\partial(a),\ ocl\_hypsubst,\ simp$)
apply($simp\ add{:}\ OclCollectionRange\_def\ OclMtOrderedSet\_def\ OclIncluding\_def$
$\quad localValidDefined2sem\ ss\_lifting'$)
done

lemma $collectionRange\_expand\_including$:
$[\![\ \tau \vDash (a{::}'a\ Integer) \leq b\ ]\!] \Longrightarrow$
$((OclCollectionRange\ a\ b){::}('a,Integer\_0)OrderedSet)\ \tau =$
$(OclCollectionRange\ a\ (b - 1))$ `->including` $b\ \tau$
oops

The conversion operators

OrderedSet to sequence

defs

$OclAsSequence\_def$:
$OclAsSequence \equiv lift1\ (strictify\ (\lambda X.$

```
                      Abs_Sequence_0 (Rep_OrderedSet_0 X)))
```

ocl_setup_op [*OclAsSequence*]

OrderedSet to bag

defs

```
  OclAsBag_def  :
  OclAsBag     ≡ lift1 (strictify (λX.
                    Abs_Bag_0 ⌊multiset_of ⌈Rep_OrderedSet_0 X⌉⌋))
```

ocl_setup_op [*OclAsBag*]

OrderedSet to orderedSet

defs

```
  OclAsOrderedSet_def  :
  OclAsOrderedSet     ≡ id
```

lemma *asOrderedSet_identity*[*simp*]:
  OclAsOrderedSet (*S*::(′*a*,′*b*::*bot*) *OrderedSet*) = *S*
  by(*simp add*: *OclAsOrderedSet_def*)

OrderedSet to set

defs

```
  OclAsSet_def :
  OclAsSet    ≡ lift1 (strictify (λX. Abs_Set_0 ⌊set ⌈Rep_OrderedSet_0 X⌉⌋))
```

ocl_setup_op [*OclAsSet*]

end

B.4.13. OCL Set

theory *OCL_Set*
imports
  $*HOLOCL_HOME/src/library/collection/smashed/OCL_Bag*
  $*HOLOCL_HOME/src/library/collection/smashed/OCL_OrderedSet*
begin

Is OCL Set Faithful to the OCL Standard?

Unlike the basic library theories, for this first and foundational “Collection”-type theory the question of faithfulness is not merely a syntactic matter (like fusing various semantic interpretation functions into one). In fact, it is not even trivial to answer. This is mainly for two reasons:

1. We generalize the concept of Set as required by OCL 2.0 to *infinite sets*.
2. We interprete the standard with respect to the question, if `OclUndefind` can be included in a set or not: we deny this question in this version of a set theory.

The first decision has the advantage, that general Sets can be used to represent the syntactic category of "types" in the standard as "characteristic sets" in HOL and enables the possibility to reason over them. From the pragmatics point of view, this also allows for quantifications of sets of "values" in the sense of OCL 2.0. For example, it is possible in HOL-OCL to express the commutativity law on Integers *inside* OCL.

The second decision is due to the fact that the standard avoids a clear statement to the nature of undefinedness in OCL. This led to two forms of interpretations by users and tool-developers: is `OclUndefined` just "null", so the null-reference, or is it a semantic construct denoting non-termination or exceptional behaviour? In the former sense, it is possible to include `OclUndefined` in a set, in the latter not. Admitting undefinedness in sets also has consequenses in the inclusion operation and the elementhood test (`->including` and `->includes`). The operational behaviour of Java-implementations reflects the latter view, which we chose therefore as default. Semantically, this leads to a quotient construction (called "smashing" in the literature) which identifies $\{x, \bot\}$ with $\bot$.

With respect to the question of faithfulness of our OCL semantics we have the following answers:

1. the required *properties* of OCL 2.0 should hold in our setting for finite sets, and
2. the subtle issue of $\bot$s in sets is not mentioned at all in them.

Therefore, we show compliance in this sense by providing the individual proof for these "required properties". However, we will only use them in rare cases in definitions and provide a number of other properties for them that are more suited for automated reasoning.

Foundational Properties of the smash-construction

Global OCL type constructor for OCL Set types:

types $('\tau,'\alpha)$ *Set* $= ('\tau,\ '\alpha$ *Set_0*$)$ *VAL*

Properties of the Congruence inside the Datatype Adaption

lemma *Abs_Set_inject_absurd11* [*simp*]:
 $[\![\ \bot \notin x\ ]\!] \Longrightarrow ((Abs\_Set\_0\ down) = (Abs\_Set\_0\ \llcorner x \lrcorner)) = False$
 by(*subst Abs_Set_0_inject*,
 simp_all add: *OCL_Set_type.Set_0_def smash_def*)

lemma *Abs_Set_0_inject_absurd12* [*simp*]:

$\llbracket \perp \notin x \rrbracket \Longrightarrow ((Abs\_Set\_0 \lfloor x \rfloor) = (Abs\_Set\_0\ down)) = False$
by(*subst Abs_Set_0_inject,*
simp_all add: OCL_Set_type.Set_0_def smash_def)

lemma *Abs_Set_0_inject_absurd21* [*simp*]:
$\llbracket \perp \notin x \rrbracket \Longrightarrow (\perp = (Abs\_Set\_0 \lfloor x \rfloor)) = False$
by(*simp add: UU_Set_def*)

lemma *Abs_Set_0_inject_absurd22* [*simp*]:
$\llbracket \perp \notin x \rrbracket \Longrightarrow ((Abs\_Set\_0 \lfloor x \rfloor) = \perp) = False$
by(*simp add: UU_Set_def*)

lemma *Abs_Set_0_inject_charn*:
$\llbracket (\perp::'\alpha::bot) \notin x; (\perp::'\alpha::bot) \notin y \rrbracket \Longrightarrow$
$(((Abs\_Set\_0 \lfloor x \rfloor) = (Abs\_Set\_0 \lfloor y \rfloor)) = (x = y))$
by(*subst Abs_Set_0_inject,*
simp_all add: OCL_Set_type.Set_0_def smash_def)

lemma *Abs_Set_0_inverse_charn1* [*simp*]:
$(\perp \notin x) \Longrightarrow Rep\_Set\_0\ (Abs\_Set\_0 \lfloor x \rfloor) = \lfloor x \rfloor$
by (*subst Abs_Set_0_inverse,*
simp_all add: OCL_Set_type.Set_0_def smash_def)

lemma *Abs_Set_0_inverse_charn2* [*simp*]:
$(\perp \notin x) \Longrightarrow \ulcorner Rep\_Set\_0\ (Abs\_Set\_0 \lfloor x \rfloor)\urcorner = x$
by (*subst Abs_Set_0_inverse,*
simp_all add: OCL_Set_type.Set_0_def smash_def)

lemma *Abs_Set_0_cases_charn*:
assumes *bottomCase* : $\llbracket x = \perp \rrbracket \Longrightarrow P$
assumes *listCase* : $\bigwedge y. (\llbracket x = Abs\_Set\_0 \lfloor y \rfloor; \perp \notin y \rrbracket \Longrightarrow P)$
shows P
apply(*rule_tac x=x* **in** *Abs_Set_0_cases*)
apply(*case_tac* $y = \perp$)
apply(*rule bottomCase, simp add: UU_Set_def*)
apply(*frule not_down_exists_lift2*[*THEN iffD1*])
apply(*rule_tac* $y=\ulcorner y \urcorner$ **in** *listCase*)
apply(*auto simp: OCL_Set_type.Set_0_def smash_def*)
done

lemma *Abs_Set_0_induct_charn*:
assumes *bottomCase* : $P \perp$
assumes *stepCase* : $\bigwedge y. (\llbracket \perp \notin y \rrbracket \Longrightarrow P\ (Abs\_Set\_0 \lfloor y \rfloor))$
shows $P\ x$
apply(*rule_tac x=x* **in** *Abs_Set_0_induct*)
apply(*rule_tac x=Abs_Set_0 y* **in** *Abs_Set_0_cases_charn*)
apply(*auto intro*!: *bottomCase elim: stepCase*)
done

the rest

lemma *inj_on_Abs_Set*: $inj\_on\ Abs\_Set\_0\ OCL\_Set\_type.Set\_0$
by(*rule inj_on_inverseI,rule Set_0.Abs_Set_0_inverse,assumption*)

lemma *inj_Rep_Set* : $inj\ Rep\_Set\_0$
by(*rule inj_on_inverseI, rule Set_0.Rep_Set_0_inverse*)

lemma *smashed_set_charn*: $(\bot \notin X) = ((\llcorner X \lrcorner) \in OCL\_Set\_type.Set\_0)$
by(*unfold smash_def OCL_Set_type.Set_0_def UU_Set_def,auto*)

lemma *UU_in_smashed_set*[*simp*]: $\bot : OCL\_Set\_type.Set\_0$
by(*unfold smash_def OCL_Set_type.Set_0_def UU_Set_def,auto*)

lemma *down_in_smashed_set*[*simp*]: $down : OCL\_Set\_type.Set\_0$
by(*unfold smash_def OCL_Set_type.Set_0_def UU_Set_def,auto*)

lemma *mt_in_smashed_set*: $\llcorner \{\} \lrcorner \in OCL\_Set\_type.Set\_0$
by (*unfold smash_def OCL_Set_type.Set_0_def,auto*)

Universal sets are never in *OCL_Set_type.Set_0*, since they must contain bottom elements (HOL's type discipline forces *UNIV* to be of type $'\alpha{::}bot\ set$) and therefore their smashed value is $\bot$.

lemma *UNIV_notin_smashed_set*: $\llcorner UNIV \lrcorner \notin OCL\_Set\_type.Set\_0$
by (*simp add*: *smash_def OCL_Set_type.Set_0_def*)

In contrast, the set consisting of all 1-lifted elements is in *OCL_Set_type.Set_0*. This justifies the construction of characteristic sets for Boolean, Integer, Real, and String.

lemma *liftUNIV_in_smashed_set*: $\llcorner lift\ `\ UNIV \lrcorner \in OCL\_Set\_type.Set\_0$
by (*auto simp*: *smash_def OCL_Set_type.Set_0_def*)

lemma *DEF_Abs_Set*: $\bot \notin X \Longrightarrow DEF\ (Abs\_Set\_0\ (\llcorner X \lrcorner))$
by(*simp add*: *DEF_def*)

lemma *DEF_Rep_Set*: $DEF\ X \Longrightarrow DEF\ (Rep\_Set\_0\ X)$
apply(*unfold DEF_def UU_Set_def,auto*)
apply(*drule_tac f* = *Abs_Set_0* **in** *arg_cong*)
apply(*simp add*: *Rep_Set_0_inverse*)
done

lemma *not_DEF_Rep_Set*: $\neg\ DEF\ X \Longrightarrow \neg\ DEF\ (Rep\_Set\_0\ X)$
apply(*unfold DEF_def UU_Set_def,auto*)
apply(*simp add*: *Abs_Set_0_inverse*)
done

lemma *exists_lift_Set*: $DEF\ X \Longrightarrow \exists c.\ Rep\_Set\_0\ X = (\llcorner c \lrcorner)$
by (*drule DEF_Rep_Set,simp add: DEF_X_up*)

lemma *exists_lift_Set2*: $DEF\ X \Longrightarrow \exists c.\ \bot \notin c \wedge Rep\_Set\_0\ X = (\llcorner c \lrcorner)$
apply(*frule exists_lift_Set*)
apply(*auto simp: smashed_set_charn*)
apply(*drule sym*)
apply(*simp add: Rep_Set_0*)
done

lemma *Rep_Set_cases*:
$Rep\_Set\_0\ X = \bot \vee (\exists c.\ \bot \notin c \wedge Rep\_Set\_0\ X = (\llcorner c \lrcorner))$
apply(*case_tac DEF X*)
apply (*drule exists_lift_Set2*)
apply (*drule_tac [2] not_DEF_Rep_Set*)
apply(*simp_all add: DEF_def*)
done

lemma *DEF_X_Set_0* : $DEF\ X = (\exists c.\ \bot \notin c \wedge Rep\_Set\_0\ X = (\llcorner c \lrcorner))$
apply (*insert Rep_Set_cases [of X], auto simp: smashed_set_charn*)
apply (*drule exists_lift_Set,auto*)
apply (*rule swap*)
prefer *2*
apply (*rule not_DEF_Rep_Set, auto*)
done

The following lemma is crucial for the theory morpher:

lemma *DEF_X_Set*: $DEF\ X = (\exists c.\ \bot \notin c \wedge X = Abs\_Set\_0\ (\llcorner c \lrcorner))$
apply (*auto simp: DEF_Abs_Set smashed_set_charn*)
apply (*simp add:DEF_X_Set_0*)
apply (*erule exE, rule exI, auto simp: smashed_set_charn*)
apply (*rule injD [OF inj_Rep_Set]*)
apply (*auto simp: Abs_Set_0_inverse*)
done

lemma [*simp*]: $\bot \notin ca \Longrightarrow Rep\_Set\_0\ (Abs\_Set\_0\ \llcorner ca \lrcorner) = \llcorner ca \lrcorner$
apply (*rule Abs_Set_0_inverse*)
apply (*simp add: smash_def OCL_Set_type.Set_0_def*)
done

lemma *RepAbs_UNIV*[*simp*]:
$Rep\_Set\_0\ (Abs\_Set\_0\ \llcorner lift\ \text{`}\ UNIV \lrcorner) = \llcorner lift\ \text{`}\ UNIV \lrcorner$
apply (*subst Abs_Set_0_inverse*)
apply (*simp_all add: Abs_Set_0_inverse liftUNIV_in_smashed_set*)
done

lemma *Abs_Set_rangelift_DEF*[*simp*]:
DEF(*Abs_Set_0* $\llcorner$*range lift*$\lrcorner$)
by(*auto simp*: *DEF_X_Set_0*)

lemma *DEF_X_Set'*:
  $[\![X \neq \bot]\!] \Longrightarrow$
   $(\exists\, c.\ (\bot \notin c) \wedge (Rep\_Set\_0\ X = \llcorner c \lrcorner)) \wedge$
   $(\exists\, c.\ (\bot \notin c) \wedge (X = (Abs\_Set\_0\ \llcorner c \lrcorner)))$
  apply(*fold DEF_def*)
  apply(*frule DEF_X_Set*[*THEN iffD1*])
  apply(*drule DEF_X_Set_0*[*THEN iffD1*])
  apply(*simp*)
  done

lemma *Set_sem_cases_0*:
  assumes *defC*: $\bigwedge$ *c d*. $[\![X \neq \bot;\ \bot \neq X;$
   $\bot \notin c;\ \bot \notin d;$
   $Rep\_Set\_0\ X = \llcorner c \lrcorner;$
   $X = Abs\_Set\_0\ \llcorner d \lrcorner\ ]\!] \Longrightarrow P\ X$
  and   *undefC*: $[\![X = \bot]\!] \Longrightarrow P\ X$
  shows $P\ X$
  apply(*rule Abs_Set_0_cases_charn*, *erule undefC*)
  apply(*rule defC*, *auto*)
  done

lemma *Set_sem_cases*:
  assumes *defC*: $\bigwedge$ *c d*. $[\![(X\ \tau) \neq \bot;\ \bot \neq (X\ \tau);$
   $\bot \notin c;\ \bot \notin d;$
   $Rep\_Set\_0\ (X\ \tau) = \llcorner c \lrcorner;$
   $(X\ \tau) = Abs\_Set\_0\ \llcorner d \lrcorner\ ]\!] \Longrightarrow P\ X\ \tau$
  and   *undefC*: $[\![(X\ \tau) = \bot]\!] \Longrightarrow P\ X\ \tau$
  and   *cpP*: *cp P*
  shows $P\ X\ \tau$
  apply(*rule_tac P2=P* **in** *subst*[*OF sym*[*OF cp_subst*]], *rule cpP*)
  apply(*rule Set_sem_cases_0*)
  apply(*rule_tac P1=P* **in** *subst*[*OF cp_subst*], *rule cpP*)
  apply(*rule defC*) **prefer** *7*
  apply(*rule_tac P1=P* **in** *subst*[*OF cp_subst*], *rule cpP*)
  apply(*rule undefC*, *simp_all*)
  done

lemma *Set_sem_cases_ext*:
  assumes *defC*: $\bigwedge$ *c d* τ. $[\![(X\ \tau) \neq \bot;\ \bot \neq (X\ \tau);$
   $(\bot \notin c);\ (\bot \notin d);$
   $Rep\_Set\_0\ (X\ \tau) = \llcorner c \lrcorner;$
   $(X\ \tau) = Abs\_Set\_0\ \llcorner d \lrcorner\ ]\!] \Longrightarrow P\ X\ \tau = Q\ X\ \tau$

```
  and   undefC: ⋀ τ. ⟦ (X τ) = ⊥ ⟧ ⟹ P X τ = Q X τ
  and      cpP: cp P
  and      cpQ: cp Q
  shows P X = Q X
  apply(rule ext)
  apply(rule_tac X=X in Set_sem_cases)
  apply(rule defC) prefer 7
  apply(rule undefC)
  apply(simp_all add: cpP cpQ)
  done
```

These four lemmas are used by ocl_setup_op to reason about definedness:

```
lemma lift2_strict_is_isdef_fw_Set_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Set_0 ⌊g ⌈Rep_Set_0 x⌉ y⌋)))
  and      inv_g: !!a b.( ⟦ ⊥ ∉ a; (⊥::'α::bot) ≠ b ⟧ ⟹ ⊥ ∉ (g a b))
  shows            ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                  o_def lift0_def lift1_def lift2_def
                  strictify_def DEF_def)
  apply(rule ext)
  apply(case_tac DEF (X τ))
  apply(frule DEF_X_Set_0[THEN iffD1])
  apply(auto simp: strictify_def DEF_def)
  apply(drule_tac b=(Y τ) in  inv_g)
  prefer 2
  apply(drule DEF_Abs_Set, simp add: DEF_def, auto)
  done

lemma lift2_strict_is_isdef_fw_Set_Set:
 assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Set_0 ⌊g ⌈Rep_Set_0 x⌉ ⌈Rep_Set_0
y⌉⌋)))
  and      inv_g: !!a b.( ⟦ ⊥ ∉ a; ⊥ ∉ b ⟧ ⟹ ⊥ ∉ (g a b))
  shows            ∂(f X Y) = (∂ X ∧ ∂ Y)
  apply(simp add: f_def OclIsDefined_def OclAnd_def
                  o_def lift0_def lift1_def lift2_def
                  strictify_def DEF_def)
  apply(rule ext)
  apply(case_tac DEF (X τ))
  apply(case_tac DEF (Y τ))
  apply(drule DEF_X_Set_0[THEN iffD1])+
  apply(auto simp: strictify_def DEF_def)
  apply(drule_tac b=ca in inv_g, assumption)
  apply(rotate_tac −1)
  apply(drule DEF_Abs_Set, simp add: DEF_def)
  done

lemma lift2_strictify_implies_LocalValid_defined_Set_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Set_0 ⌊g ⌈Rep_Set_0 x⌉ y⌋)))
  and      inv_g: !!a b.( ⟦ ⊥ ∉ a; (⊥::'α::bot) ≠ b⟧ ⟹ ⊥ ∉ (g a b))
```

```
  shows        (τ ⊨ ∂(f X Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
  apply(insert f_def)
  apply(drule_tac X=X and Y=Y in lift2_strict_is_isdef_fw_Set_Val)
  apply(rule inv_g, assumption+)
  apply(simp add: OclAnd_def  OclTrue_def o_def
            OclIsDefined_def lift0_def lift1_def lift2_def
            OclLocalValid_def)
  done

lemma lift2_strictify_implies_LocalValid_defined_Set_Set:
 assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Set_0 ⌊g ⌈Rep_Set_0 x⌉ ⌈Rep_Set_0
y⌉⌋)))
  and     inv_g: !!a b.( ⟦ ⊥ ∉ a; ⊥ ∉ b ⟧ ⟹ ⊥ ∉ (g a b))
  shows        (τ ⊨ ∂(f X Y)) = ((τ ⊨ ∂ X) ∧ (τ ⊨  ∂ Y))
  apply(insert f_def)
  apply(drule_tac X=X and Y=Y in lift2_strict_is_isdef_fw_Set_Set)
  apply(rule inv_g, assumption+)
  apply(simp add: OclAnd_def  OclTrue_def o_def
            OclIsDefined_def lift0_def lift1_def lift2_def
            OclLocalValid_def)
  done
```

The conversion operators

From sets to sequences

Because there is no ordering on the elements of the sets the *AsSequence* operator choses some sequence which has exactly the same elements as the set.

Because this is just some sequence the proof about statements containing the *AsSequence* operator, have to show the same properties as one has to show for the general Hilbert Choice operator.

```
defs

  OclAsSequence_def  :
  OclAsSequence      ≡ lift1 (strictify (λX.
                        Abs_Sequence_0 ⌊list_of_set ⌈Rep_Set_0 X⌉⌋))
ocl_setup_op [OclAsSequence]
```

From sets to bags

```
defs

  OclAsBag_def  :
  OclAsBag      ≡ lift1 (strictify (λX.
                   Abs_Bag_0 ⌊multiset_of_set ⌈Rep_Set_0 X⌉⌋))
ocl_setup_op [OclAsBag]
```

From sets to ordered sets

```
defs
```

$OclAsOrderedSet\_def$:
$OclAsOrderedSet \equiv lift1\ (strictify\ (\lambda X.$
$Abs\_OrderedSet\_0\ \lfloor list\_of\_set\ \ulcorner Rep\_Set\_0\ X \urcorner \rfloor))$

ocl_setup_op [*OclAsOrderedSet*]

lemma *bag2seq2oset_eq_bag2set2oset*:
->asOrderedSet() ((->asSequence() $(B{::}('\tau,'\alpha{::}bot)Bag)){::}('\tau,'\alpha)Sequence) =$
((->asOrderedSet() ((->asSet() $B){::}('\tau,'\alpha)Set)){::}('\tau,'\alpha)OrderedSet)$
apply(*rule ext*)
apply(*case_tac* $x \vDash \emptyset$ *B*, *ocl_hypsubst*, *simp*)
apply(*simp add*: *localValidDefined2sem DEF_def_both*, *clarify*)
apply(*frule DEF_X_Bag'*, *clarify*)
apply(*simp add*: *OCL_Bag.OclAsSequence_def OCL_Bag.OclAsSet_def*
OCL_Sequence.OclAsOrderedSet_def
OCL_Set.OclAsOrderedSet_def ss_lifting')
apply(*subst Abs_OrderedSet_0_inject_charn*)
apply(*simp_all add*: *rev_remdups rev_remstutter*[*symmetric*])
done

lemma *oset2seq2bag_eq_oset2set2bag*:
->asBag() ((->asSequence() $(X{::}('\tau,'\alpha{::}bot)OrderedSet)){::}('\tau,'\alpha)Sequence) =$
((->asBag() ((->asSet() $X){::}('\tau,'\alpha)Set)){::}('\tau,'\alpha)Bag)$
apply(*rule OrderedSet_sem_cases_ext*, *simp_all*)
apply(*simp_all add*: *OCL_OrderedSet.OclAsSequence_def OCL_OrderedSet.OclAsSet_def*
OCL_Sequence.OclAsBag_def
OCL_Set.OclAsBag_def ss_lifting')
done

From sets to sets

defs

$OclAsSet\_def$:
$OclAsSet \equiv id$

lemma *set2set_id* [*simp*]:
$((OclAsSet\ (B{::}('\tau,'\alpha{::}bot)Set)){::}('\tau,'\alpha)Set) = B$
by(*simp add*: *OclAsSet_def*)

The Definition of the Set Operators

The Powerset Operator

constdefs
$Set :: ('st, ('\alpha{::}bot)\ Set\_0)\ VAL \Rightarrow ('st, '\alpha\ Set\_0\ Set\_0)\ VAL$
$Set \equiv lift1(strictify\ (\lambda X.$
$Abs\_Set\_0\lfloor(Abs\_Set\_0\ o\ lift)\ `\ \{x.\ x \subseteq \ulcorner Rep\_Set\_0\ X \urcorner\}\rfloor))$**ocl_setup_op**
[*Set*]

lemma *Set_defined* :
$\tau \models \partial\ X \Longrightarrow\ \tau \models \partial(Set\ X)$
apply(*simp add*: *Set_def localValidDefined2sem*)
apply (*auto simp*: *strictify_def*
o_def lift0_def lift1_def lift2_def image_def
OclTrue_def OclFalse_def OclUndefined_def)
apply(*rule DEF_Abs_Set*, *simp*)
apply(*rule allI*, *rule impI*)
apply(*rule neq_commute*[*THEN iffD1*])
apply(*simp only*:*DEF_def*[*symmetric*])
apply(*rule DEF_Abs_Set*, *simp*)
thm *exists_lift_Set2*
apply(*drule exists_lift_Set2*, *auto*)
done

lemma *Set_definedR* :
$(\tau \models \partial(Set\ X)) =\ (\tau \models \partial\ X)$
apply (*auto simp*: *Set_defined*)
oops

The Empty Set

constdefs
$OclMtSet\ ::\ ('\tau,\ '\alpha{::}bot\ Set\_0)\ VAL$
$OclMtSet\ \equiv\ lift0(Abs\_Set\_0(\lfloor\{\}\rfloor))$

syntax
$\_OclMtSet\_std\ ::\ ('\tau,'\alpha{::}bot)\ VAL \Rightarrow\ '\tau\ Boolean\qquad (\{\})$
syntax
$\_OclMtSet\_ascii :: ('\tau,'\alpha{::}bot)\ VAL \Rightarrow\ '\tau\ Boolean\qquad (mtSet)$
syntax (*xsymbols*)
$\_OclMtSet\_math\ ::\ ('\tau,'\alpha{::}bot)\ VAL \Rightarrow\ '\tau\ Boolean\qquad (\emptyset)$

Foundational Properties of MtSet

lemma *Abs_Set_Mt_DEF*[*simp*]:
$DEF(Abs\_Set\_0\ \lfloor\{\}\rfloor)$
by(*simp add*: *DEF_X_Set_0*)

lemma *mtSet_Defined* [*simp*]:
$\partial\ \emptyset = \mathsf{T}$
by(*simp add*: *localValidDefined2sem Abs_Set_Mt_DEF OclMtSet_def*
OclIsDefined_def lift1_def lift0_def OclTrue_def)

Foundational Properties of includes and excludes

defs

$OclIncludes\_def :\ \ OclIncludes \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda x.$

$\llcorner x \in \ulcorner Rep\_Set\_0\ X \urcorner \lrcorner)))$

$OclExcludes\_def : OclExcludes \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda x.$
$\llcorner x \notin \ulcorner Rep\_Set\_0\ X \urcorner \lrcorner)))$

ocl_setup_op [*OclIncludes*, *OclExcludes*]

lemma *includes_charn1* : $(\tau \models x \in S) \Longrightarrow (x\ \tau \in \ulcorner Rep\_Set\_0\ (S\ \tau) \urcorner)$
apply (*frule isDefined_if_valid,*
drule OCL_is_defopt_OclIncludes[*THEN iffD1*])
apply (*simp add: OclLocalValid_def OclIsDefined_def strictify_def*
OclNot_def OclIncludes_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma *includes_charn2* : $[\![\ DEF(S\ \tau);\ x\ \tau \in \ulcorner Rep\_Set\_0\ (S\ \tau) \urcorner\ ]\!] \Longrightarrow \tau \models x \in S$
apply (*frule DEF_X_Set* [*THEN iffD1*], *erule exE*)
apply (*simp add: OclLocalValid_def OclIsDefined_def strictify_def*
OclNot_def OclIncludes_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
apply *auto*
done

Higher Properties of includes and excludes

lemma *exludes_not_includes*[*simp*]:
$x \notin X = \neg\ (x{::}('\tau,\ '\alpha)\ VAL) \in (X{::}('\tau,\ '\alpha{::}bot)\ Set)$
apply (*rule ext*)
apply (*simp add: OclExcludes_def OclIncludes_def*)
apply (*simp only: OclNot_def*)
apply (*simp only: OclIncludes_def OclExcludes_def lift1_def lift2_def*
strictify_def o_def not_def)
apply (*simp_all* (*no_asm_use*) *add: UU_fun_def DEF_def strictify_def*
split add: split_if up.split)
done

lemma *mtSet_Includes_charn* :
$\tau \models \partial\ x \Longrightarrow$
$\tau \models (x{::}('\tau,\ '\alpha)\ VAL) \notin (\emptyset{::}('\tau,\ '\alpha{::}bot)\ Set)$
apply(*simp add: localValidDefined2sem*)
apply(*simp only: exludes_not_includes OclMtSet_def OclLocalValid_def*
OclIncludes_def lift0_def lift1_def lift2_def

```
              strictify_def o_def OclNot_def OclTrue_def DEF_def)
apply(simp add: Abs_Set_Mt_DEF [simplified DEF_def] strictify_def)
done


lemma X_in_Set_X:
 τ ⊨ ∂ (S::('τ, 'α::bot)Set) ⟹
  τ ⊨ S ∈ (Set S)
oops

lemma mtSet_in_Set:
 τ ⊨ ∂ (S::('τ, 'α::bot)Set) ⟹
  τ ⊨ (∅::('τ,'α::bot)Set) ∈ (Set S)
oops
```

Consequences of Smashedness of Sets

This is the characteristic property of smashed sets: if `X ->includes(x)` is valid, we know that x must be defined.

```
lemma smashed_sets_charn:
τ ⊨ ((x::('τ,'α) VAL) ∈ (X::('τ,'α::bot)Set)) ⟹ τ ⊨ ∂ x
apply (simp add: OclIncludes_def lift2_def lift1_def lift0_def
               strictify_def OclTrue_def OclIsDefined_def o_def OclLocalValid_def)
apply(case_tac X τ = ⊥, auto)
apply(simp add: strictify_def DEF_def)
apply(case_tac x τ = ⊥, auto)
done
```

Fundamental Theorem on (infinite) Sets

```
lemma Set_Ext0 :
(∀  x::(('τ,'α) VAL). τ ⊨ ∂ x ⟶ (x ∈ (S::('τ,'α::bot)Set))τ = (x ∈ T)τ)
 ⟹ S τ = T τ
apply (simp add: OclIncludes_def lift2_def lift1_def lift0_def
               strictify_def OclTrue_def OclIsDefined_def o_def OclLocalValid_def)
apply(case_tac S τ = ⊥, auto)
apply(case_tac T τ = ⊥, auto)
apply(insert exists_DEF, auto)
apply(case_tac T τ = ⊥, auto)
apply(simp add: DEF_def[symmetric])
apply (frule DEF_Rep_Set)
apply (frule DEF_Rep_Set) back
apply (simp add: DEF_X_Set)
apply auto
apply (rule_tac f = λ c. Abs_Set_0 ⌞c⌟ in arg_cong)
apply (rule set_ext)
apply (erule_tac x=λ r. xa in allE)
apply(case_tac x=⊥, simp_all add: DEF_def[symmetric])
```

apply (*auto simp*: *DEF_def*)
done

lemma *Set_Ext* :
assumes $H : \bigwedge x::(('\tau,'\alpha)\, VAL).\ \tau \models \partial\ x \Longrightarrow (x \in (S::('\tau,'\alpha::bot)Set))\tau = (x \in T)\tau$
shows $S\ \tau = T\ \tau$
apply(*rule Set_Ext0*)**back**
by(*auto intro*: *H*)

Semantic Definitions of the Standard Set Operations

defs

$$
\begin{aligned}
&\mathit{OclSize\_def} \quad :(\mathit{OclSize}::('\tau,\ '\alpha::bot\ Set\_0)\ \ VAL => '\tau\ Integer) \equiv \\
&\qquad (\mathit{lift1}\ (\mathit{strictify}\ (\lambda X. \\
&\qquad\qquad \mathit{if\ infinite}\ \ulcorner Rep\_Set\_0\ X\urcorner\ \mathit{then}\ \bot \\
&\qquad\qquad \mathit{else}\ (\llcorner(\mathit{int}(\mathit{card}(\ulcorner Rep\_Set\_0\ X\urcorner)))\lrcorner))))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{OclCount\_def} \quad : \mathit{OclCount} \equiv \mathit{lift2}\ (\mathit{strictify}\ (\lambda X.\ \mathit{strictify}\ (\lambda x. \\
&\qquad\qquad \llcorner \mathit{if}\ x \in \ulcorner Rep\_Set\_0\ X\urcorner \\
&\qquad\qquad \mathit{then\ int\ 1} \\
&\qquad\qquad \mathit{else\ int\ 0}\lrcorner)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{OclIncludesAll\_def}: \mathit{OclIncludesAll} \equiv \mathit{lift2}\ (\mathit{strictify}\ (\lambda X.\ \mathit{strictify}\ (\lambda Y. \\
&\qquad\qquad \llcorner\ulcorner Rep\_Set\_0\ Y\urcorner \subseteq \ulcorner Rep\_Set\_0\ X\urcorner\lrcorner)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{OclExcludesAll\_def}: \mathit{OclExcludesAll} \equiv \mathit{lift2}\ (\mathit{strictify}\ (\lambda X.\ \mathit{strictify}\ (\lambda Y. \\
&\qquad\qquad \llcorner\ulcorner Rep\_Set\_0\ X\urcorner \cap \ulcorner Rep\_Set\_0\ Y\urcorner = \{\}\lrcorner)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{OclIsEmpty\_def}\ \ : \mathit{OclIsEmpty} \equiv \mathit{lift1}\ (\mathit{strictify}\ (\lambda X. \\
&\qquad\qquad \llcorner\{\} = \ulcorner Rep\_Set\_0\ X\urcorner\lrcorner)) \\
&\mathit{OclNotEmpty\_def} : \mathit{OclNotEmpty} \equiv \mathit{lift1}\ (\mathit{strictify}\ (\lambda X. \\
&\qquad\qquad \llcorner\{\} \neq \ulcorner Rep\_Set\_0\ X\urcorner\lrcorner))
\end{aligned}
$$

defs

$$
\mathit{OclSum\_def} \quad : \mathit{OclSum}\ (\mathit{self}::('\tau,\ '\alpha::bot\ Set\_0)\ \ VAL)\ \ \equiv \bot
$$

ocl_setup_op [*OclSize*, *OclCount*, *OclIncludesAll*, *OclExcludesAll*,
OclIsEmpty, *OclNotEmpty*]

`sum` *wrongly defined in the standard: idempotence*

defs
$OclFlatten\_def$: $(OclFlatten :: ('\tau,'\alpha::bot\ Set\_0\ Set\_0)\ VAL \Rightarrow ('\tau,'\alpha\ Set\_0)\ VAL) \equiv$
$lift1\ (strictify\ (\lambda X.\ Abs\_Set\_0$
$(\llcorner\bigcup\ (Lifting.drop \circ Rep\_Set\_0)\ `\ \ulcorner Rep\_Set\_0\ X\urcorner\lrcorner)))$

ocl_setup_op [*OclFlatten*]

defs

$OclIncluding\_def$: $OclIncluding \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Set\_0$
$(\llcorner insert\ Y\ \ulcorner Rep\_Set\_0\ X\urcorner\lrcorner))))$

$OclExcluding\_def$: $OclExcluding \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Set\_0$
$(\llcorner\ulcorner Rep\_Set\_0\ X\urcorner - \{Y\}\lrcorner))))$

$OclComplement\_def$: $OclComplement \equiv lift1\ (strictify(\lambda X.\ Abs\_Set\_0$
$(\llcorner - (\ulcorner Rep\_Set\_0\ X\urcorner - \{\bot\})\lrcorner)))$

$OclUnion\_def$: $OclUnion \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Set\_0$
$(\llcorner\ulcorner Rep\_Set\_0\ X\urcorner \cup \ulcorner Rep\_Set\_0\ Y\urcorner\lrcorner))))$

$OclIntersection\_def$: $OclIntersection \equiv lift2\ (strictify\ (\lambda X.\ strictify\ (\lambda Y.\ Abs\_Set\_0$
$(\llcorner\ulcorner Rep\_Set\_0\ X\urcorner \cap \ulcorner Rep\_Set\_0\ Y\urcorner\lrcorner))))$

ocl_setup_op [*OclIncluding*, *OclExcluding*,
OclComplement, *OclUnion*]

ocl_setup_op [*OclIntersection*]

declare *OCL_cp_OclIntersection*[*simp*,*intro*!]

Note that also sequencing of declarations are possible.

syntax
$@OclFinSet\ \ :: args \Rightarrow ('\tau,'\alpha\ Set\_0)\ VAL\ \ \ (mkSet\{(\_)\})$

lemma $mkSet\{1,1,1,0,1\} = ?X$ **oops**

Properties of the Operations

Useful Lemmas (necessary in this theory)

```
lemma OclIntersection_idem[simp]: ((S::('τ,'α::bot)Set) ∩ S)  = S
apply (rule ext)
apply (case_tac x ⊨  ∂(S))
apply (simp_all, ocl_hypsubst [2], simp_all)
apply (simp only: OclIntersection_def
                  OclLocalValid_def OclIsDefined_def strictify_def
                  OclNot_def
                  o_def lift0_def lift1_def lift2_def
                  OclTrue_def DEF_def)
apply (simp_all add: strictify_def lift_drop_idem DEF_def[symmetric]
                    DEF_Rep_Set Rep_Set_0_inverse)
done

lemma not_empty: OclNotEmpty(X::('τ, 'α::bot) Set) =  ¬ (OclIsEmpty(X))
apply (rule ext)
apply (simp add:  OclNotEmpty_def OclIsEmpty_def OclNot_def)
apply (simp only: lift1_def lift2_def
                  strictify_def o_def not_def)
apply (simp_all (no_asm_use) add: UU_fun_def DEF_def strictify_def
                              split add: split_if up.split)
done


lemma OclIsEmpty_charn : OclIsEmpty(X) = (X ≐ ∅)

apply (rule ext)
apply (case_tac x ⊨  ∂(X))
apply (simp_all)
apply (ocl_hypsubst [2], simp_all)

apply (rule sym, rule trans)
apply (rule strictEq_is_strongEq_LE)
apply (simp_all)

apply (simp only: OclIsDefined_def OclTrue_def OclLocalValid_def OclMtSet_def
                  OclIsEmpty_def OclStrongEq_def
                  lift0_def lift1_def lift2_def
                  strictify_def o_def not_def)
apply (auto simp: DEF_def)
apply (rule_tac t = {} in ssubst, assumption)
apply (subst lift_drop_idem)
apply (rule DEF_Rep_Set)
apply (auto simp: DEF_def Rep_Set_0_inverse)
done
```

```
lemma OclIsNotEmpty_local_charn_LJE :
   τ ⊨ ∂(X::('τ,'α::bot)Set) ⟹
    (τ ⊨ (≠ ∅(X))) =
    (∃ a::('τ,'α::bot) VAL. τ ⊨ ∂a ∧ τ ⊨ a ∈ X)
apply (simp only: OclIsDefined_def OclTrue_def OclLocalValid_def
                  OclNotEmpty_def OclIncludes_def
                  OclIsEmpty_def OclStrongEq_def
                  lift0_def lift1_def lift2_def
                  strictify_def o_def not_def DEF_def)
apply (simp)
apply (subst neq_commute, subst ex_in_conv[symmetric])
apply auto
apply (rule_tac x=λ c. x in exI)
apply (simp add: strictify_def DEF_def[symmetric] DEF_X_Set)
apply auto
apply (subgoal_tac Rep_Set_0 (Abs_Set_0 ⌊c⌋) = ⌊c⌋)
apply (auto simp:DEF_def)
done
```

Sets with defined size must be finite.

```
lemma finite_Sets: τ ⊨ ∂((self::('τ,'α::bot)Set)->size()) ⟹
                    finite ⌈Rep_Set_0 (self τ)⌉
apply (simp add:  OclIsDefined_def OclTrue_def lift0_def lift1_def lift2_def
                  strictify_def o_def not_def)
apply (simp only: OclSize_def)
apply (simp add:  lift0_def lift1_def lift2_def strictify_def DEF_def UU_up_def)
apply (cases self τ  = ⊥)
apply (auto  simp: OclLocalValid_def OclTrue_def lift0_def)
apply (cases finite (Lifting.drop (Rep_Set_0 (self τ))))
apply (auto  simp: OclLocalValid_def)
done
```

Sets with defined size must be defined.

```
lemma def_if_size_def : τ ⊨ ∂ (self::('τ,'α::bot)Set) ->size() ⟹
                        τ ⊨ ∂ self
apply (simp add:  OclIsDefined_def OclTrue_def lift0_def lift1_def lift2_def
                  strictify_def o_def not_def)
apply (simp only: OclSize_def)
apply (simp add:  lift0_def lift1_def lift2_def OclLocalValid_def
                  strictify_def DEF_def UU_up_def)
apply (cases self τ = ⊥)
apply (auto  simp: OclTrue_def lift0_def)
done

lemma def_if_size_def_global:
     ⊨ ∂(self::('τ, 'α::bot)Set) ->size() ⟹  ⊨ ∂ self
apply (rule valid_intro)
apply (drule_tac τ=τ in valid_elim)
```

```
apply (simp add: def_if_size_def)
done
```

```
lemma REQ_11_7_1_4_alt2:
  (self::('τ,'α::bot) Set) ->count((obj::('τ, 'α) VAL)) =
   (if obj ∈ self then 1 else 0 endif)
apply (rule ext)
apply (simp add:  OclCount_def OclIncludes_def OclIf_def OclStrictEq_def
                  OCL_Integer.Zero_ocl_int_def OCL_Integer.One_ocl_int_def)
apply (simp only: lift0_def lift1_def lift2_def lift3_def
                  strictify_def o_def not_def OclIsDefined_def OclTrue_def)
apply (simp_all (no_asm_use) add: UU_fun_def DEF_def strictify_def split
                            add: split_if up.split)
done
```

Semantic Definitions of the Iterators

With "iterator", the standard attepts to subsume several vaguely related higher-order concepts such as fold, map, filter and comprehensions. The quantifier $ForAll$ is in fact be defined via fold. Since this approach only works for finite sets, we do not follow the standard here and give a more general definition, but show that this coincides with the standards requirements for finite sets.

Universal Quantifiers

```
defs

  OclForAll_def: OclForAll ≡ (lift2' lift_arg0 lift_arg1) (strictify( λ S. (λ P.
                      if ∀ x ∈ ⌜Rep_Set_0 S⌝.  P x = ⌞True⌟
                      then ⌞True⌟
                      else if ∃ x ∈ ⌜Rep_Set_0 S⌝. P x = ⌞False⌟
                           then ⌞False⌟
                           else ⊥)))

ocl_setup_op[OclForAll]
```

```
lemma OclExists_UU1 [simp]:
     (∃ x ∈ (⊥::('τ, 'α::bot)Set) . P (x::('τ, 'α) VAL)) = ⊥
  by(simp add: OclExists_def)
```

```
lemma OclForAllMt [simp]:
     (∀ x ∈ (∅::('τ, 'α::bot)Set)  . P (x::('τ, 'α::bot) VAL)) = T
  by(simp add: OclForAll_def OclTrue_def OclMtSet_def ss_lifting')
```

lemma *OclExistsMt* [*simp*]:
    $(\exists\ x \in (\emptyset{::}('\tau,\ '\alpha{::}bot)Set)\ \ \bullet\ P\ (x{::}('\tau,\ '\alpha{::}bot)\,VAL)) = \mathsf{F}$
  by(*simp add*: *OclExists_def*)

lemma *ForAllI*:
assumes *defS* : $\tau \vDash \partial(S{::}('\tau,('\alpha{::}bot))Set)$
and *allP* : $\bigwedge\ x.\ \tau \vDash x \in S \Longrightarrow \tau \vDash P\ x$
shows $\tau \vDash (\forall\ x \in S\ \bullet\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))$
  apply(*insert defS*)
  apply(*rule_tac X=S* **in** *Set_sem_cases*)
  apply(*simp_all add*: *localValidDefined2sem DEF_def*)
  apply(*subgoal_tac* $(\forall\, x \in\ c.\ P\ (\lambda\tau.\ x)\ \tau = \llcorner True\lrcorner)$)
  apply(*auto simp*: *OclForAll_def localValid2sem ss_lifting'*)
  apply(*rule_tac* $\tau 1 = \tau$ **in** *localValid2sem*[*THEN iffD1*], *rule allP*)
  apply(*auto simp*: *localValid2sem OclIncludes_def ss_lifting'*)
  done

lemma *notForAllI*:
  assumes *isin* : $\tau \vDash x \in (S{::}('\tau,('\alpha{::}bot))Set)$
  and *not* : $\tau \vDash \neg\ (P\ x)$
  and *cpP* : *cp P*
  shows $\tau \vDash \neg\ (\forall\ x \in S\ \bullet\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))$
  apply(*insert isin not cpP*)
  apply(*rule_tac X=S* **in** *Set_sem_cases*)
  apply(*simp_all add*: *localValidDefined2sem DEF_def*)
  apply(*subgoal_tac* $(P\ x\ \tau) = (P\ (\lambda s.\ (x\ \tau))\ \tau)$)
  apply(*case_tac* $(P\ x)\ \tau \neq \bot$, *frule neq_commute*[*THEN iffD1*])
  apply(*case_tac* $x\ \tau \neq \bot$, *rotate_tac* -1, *frule neq_commute*[*THEN iffD1*])
  apply(*subgoal_tac* $(\exists\, x{\in}\ c.\ ((P\ (\lambda s.\ x)\ \tau) = \llcorner False\lrcorner))$)
  apply(*simp_all add*: *OclForAll_def OclIncludes_def OclNot_def*
        ss_lifting' localValid2sem o_def)
  apply(*rule_tac* $x{=}x\ \tau$ **in** *bexI*)
  apply(*rule_tac* $x{=}P\ (\lambda s.\ (x\ \tau))\ \tau$ **in** *boolean_cases_sem*)
  apply(*simp_all*)
  apply(*rule_tac* $x{=}x\ \tau$ **in** *bexI*)
  apply(*rule_tac* $x{=}P\ (\lambda s.\ (x\ \tau))\ \tau$ **in** *boolean_cases_sem*)
  apply(*simp_all*)
  apply(*rule trans*, *rule_tac x=x* **in** *cp_subst*)
  apply(*simp_all add*: *lift0_def*)
  done

lemma *ForAllD*:
  assumes *allP* : $\tau \vDash (\forall\ x \in S\ \bullet\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))$
  and *cpP* : *cp P*
  shows $\bigwedge\ x.\ \tau \vDash x \in (S{::}('\tau,('\alpha{::}bot))Set) \Longrightarrow \tau \vDash P\ x$
  apply (*insert allP cpP*)

```
  apply (simp add: localValid2sem OclForAll_def OclNot_def OclIncludes_def
                 ss_lifting')
  apply (simp add:   strictify_def
              split: split_if_asm)
  apply (erule_tac x=x τ in ballE)
  apply (simp_all add: cp_by_cpify lift0_def)
  done

lemma localValidForall2forall :
  assumes defS : τ ⊨ ∂(S::('τ,('α::bot))Set)
  and     cpP  : cp P
  shows   (τ ⊨ ∀ x ∈ S • P(x::'τ ⇒ 'α::bot)) =
          (∀ x. τ ⊨ x ∈ S ⟶ τ ⊨ P x)
  apply (safe)
  apply (drule  ForAllD, simp_all add: cpP)
  apply (rule   ForAllI, simp_all add: defS)
  done


lemma undefForAllI1:
  ⟦  τ ⊨ ∂̸ (S::('τ,('α::bot))Set)⟧
  ⟹ τ ⊨ ∂̸ (∀ x ∈ S • P(x::'τ ⇒ 'α))
  by(erule_tac P = λ X. (∂̸ (∀ x ∈ X • P x)) in subst_LJ_undef, simp_all)

lemma undefForAllI2:
  assumes exUndef1: τ ⊨ x ∈ (S::('τ,('α::bot))Set)
  and     exUndef2: τ ⊨ ∂̸(P x)
  and     allP     : ⋀ x. τ ⊨ x ∈ S ⟹ τ ⊨ (P x) ∨ ∂̸(P x)
  and     cpP      : cp P
  shows              τ ⊨ ∂̸(∀ x ∈ S • P(x::'τ ⇒ 'α::bot))
  apply(insert exUndef1 exUndef2)
  apply(frule isDefined_if_valid)
  apply(drule OCL_is_defopt_OclIncludes[THEN iffD1])
  apply(simp add: OclLocalValid_def OclIsDefined_def OclTrue_def
                 OclForAll_def OclNot_def OclIncludes_def
                 ss_lifting')
  apply(erule conjE)
  apply(frule DEF_X_Set', clarify)
  apply(simp, safe)
  apply(rule_tac x = xa in bexI, simp_all) prefer 2
  apply(rule_tac x = x τ in bexI, simp_all)
  apply(subgoal_tac P (λτa. x τ) τ = P x τ, simp)
  apply(rule_tac τ = τ and P = P in cp_charn, simp add: cpP, rule cpP)
  apply(erule_tac Q = P (λτ. xa) τ = ⌞False⌟ in contrapos_pp)
  apply(rule_tac τ1 = τ in localValidToNotFalse2sem [THEN iffD1]) back
  apply(rule allP)
  apply(simp add: includes_charn2 DEF_def)
  done
```

lemma *isdefForAllI2*:
  assumes *exNot1*: $\tau \vDash x \in (S{::}('\tau,('\alpha{::}bot))Set)$
  and *exNot2*: $\tau \vDash \neg\ (P\ x)$
  and *cpP* : *cp P*
  shows $\tau \vDash \partial(\forall\ x \in S\ .\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))$
  apply(*insert exNot1 exNot2 cpP*)
  apply(*drule_tac P=P* **in** *notForAllI*)
  apply(*simp_all add*: *isDefined_if_invalid*)
  done

lemma *isDefinedForall_UC*:
assumes *cpP* : *cp P*
shows $(\partial(\forall\ x \in S\ .\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))) =$
$((\ \partial\ (S{::}('\tau,'\alpha)Set)) \wedge$
$((\exists\ x \in S\ .\ (\neg\ (P\ x))) \vee$
$(\forall\ x \in S\ .\ (\partial\ (P\ x)))))$
oops

lemma *isDefinedForall_LJE*:
assumes *cpP* : *cp P*
shows $\tau \vDash \partial(\forall\ x \in S\ .\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot)) =$
$((\tau \vDash \partial\ S) \wedge$
$((\exists\ x.\ (\tau \vDash x \in (S{::}('\tau,'\alpha)Set)) \wedge \neg\ (\tau \vDash\ P\ x)) \vee$
$(\forall\ x.\ \ (\tau \vDash \partial\ x \wedge \tau \vDash x \in S)\ \ \longrightarrow (\tau \vDash \partial\ (P\ x)))))$
oops

lemmas *isdefForAllI3* = *ForAllI*[*THEN isDefined_if_valid*]

lemma *ExistsI*:
assumes *isin* : $\tau \vDash x \in (S{::}('\tau,('\alpha{::}bot))Set)$
and *not* : $\tau \vDash (P\ x)$
and *cpP* : *cp P*
shows $\tau \vDash (\exists\ x \in S\ .\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))$
by (*auto simp*:*OclExists_def intro*!: *notForAllI isin not cpP*)

lemma *NotExistsI*:
assumes *defS* : $\tau \vDash \partial(S{::}('\tau,('\alpha{::}bot))Set)$
and *allP* : $\bigwedge\ x.\ \tau \vDash x \in S \Longrightarrow \tau \vDash\ \neg\ P\ x$
shows $\tau \vDash\ \neg(\exists\ x \in S\ .\ P(x{::}'\tau \Rightarrow\ '\alpha{::}bot))$
by (*auto simp*:*OclExists_def intro*!: *ForAllI defS allP*)

lemma *ExistsD*:

```
  assumes ExP : τ ⊨ (∃ x ∈ S • P(x::'τ ⇒ 'α::bot))
  shows        ∃ x. τ ⊨ x ∈ (S::('τ,('α::bot))Set) ∧ τ ⊨ (P x)
  apply (insert ExP)
  apply (simp add: localValid2sem OclForAll_def OclNot_def OclIncludes_def
                   OclExists_def o_def ss_lifting)
  apply (simp add:   strictify_def
              split: split_if_asm)
  apply(frule DEF_X_Set', clarify)
  apply(rule_tac x=lift0 x in exI)
  apply(auto simp: lift0_def not_down_exists_lift)
  done

lemma   localValidExists2exists:
  assumes cpP  : cp P
  assumes defS : τ ⊨ ∂(S::('τ,'α::bot)Set)
  shows         (τ ⊨ (∃ x ∈ S • P(x::'τ ⇒ 'α::bot))) =
               (∃ x. τ ⊨ x ∈ S ∧ τ ⊨ (P x))
  apply (safe)
  apply (drule ExistsD, simp)
  apply (auto intro!:  ExistsI cpP)
  done

lemmas weak_pred_LJE = weak_prop_LJE
                       localValidForall2forall
                       localValidExists2exists

thm weak_pred_LJE
```

Select and Collect

```
defs

   OclSelect_def:  OclSelect ≡  (lift2' lift_arg0 lift_arg1) (strictify (λ S P.
                   if    ∀ x ∈ ⌜Rep_Set_0 S⌝. DEF (P x)
                   then  Abs_Set_0(⌞{x∈⌜Rep_Set_0 S⌝. P x = ⌞True⌟}⌟)
                   else  ⊥))
ocl_setup_op [OclSelect]

lemma SelectPartialUndef :
assumes ainS : τ ⊨ a ∈ (S::('τ,('α::bot))Set)
and     allP : τ ⊨ ∂̸ (P a)
and     cpP  : cp P
shows         ((|x : S | P (x::'τ⇒'α::bot)|)) τ = ⊥ τ
apply (insert ainS allP)
apply (frule isDefined_if_valid,
       drule OCL_is_defopt_OclIncludes[THEN iffD1])
apply (simp add: OclSelect_def OclIncluding_def OclUndefined_def
                 OclLocalValid_def OclIsDefined_def OclTrue_def
                 OclForAll_def OclNot_def OclIncludes_def
```

```
                 ss_lifting′)
apply auto
apply (erule_tac x=a τ in ballE, simp_all)
apply (subgoal_tac P (λτ′. a τ) τ = P a τ, simp)
apply (rule_tac τ = τ and P = P in cp_charn, simp add: cpP,rule cpP)
done

lemma SelectPartialUndef2[simp] :
assumes ainS : τ ⊨ (a::′τ⇒′α::bot) ∈ (S::(′τ,(′α::bot))Set)
shows           (|x : S | (λx. ⊥)((x::′τ⇒′α::bot))|) τ = ⊥ τ
by(rule SelectPartialUndef, auto intro!: ainS)


lemma mem_OclSelect_eq:
assumes defS : τ ⊨ ∂(S::(′τ,(′α::bot))Set)
and      defa : τ ⊨ ∂(a::(′τ⇒′α::bot))
and      ainS : τ ⊨ a ∈ S
and      allP : ⋀ x. τ ⊨ x ∈ S ⟹ τ ⊨  ∂(P x)
and      cpP  : cp P
shows           (τ ⊨ a ∈ ((|x : S | P (x::′τ⇒′α::bot)|)))
              = (τ ⊨ P a)
apply (insert defS defa ainS)
apply (simp add: OclSelect_def OclIncluding_def OclUndefined_def
                 OclLocalValid_def OclIsDefined_def OclTrue_def
                 OclForAll_def OclNot_def OclIncludes_def
                 ss_lifting′)
apply (frule DEF_X_Set′, clarify)
apply(simp)
apply(rule_tac t=P a τ in subst)
apply(rule sym, rule_tac P=P and x=a in cp_subst, rule cpP)
apply(rule_tac x=P (lift0 (a τ)) τ in boolean_cases_sem)
apply(auto simp: lift0_def)
apply(rotate_tac −1, erule contrapos_pp)
apply(rule allP[simplified
                 OclLocalValid_def OclIsDefined_def OclTrue_def
                 OclForAll_def OclNot_def OclIncludes_def
                 o_def ss_lifting′, simplified])
apply(auto simp: strictify_def)
done


lemma nonmem_OclSelect:
assumes defS : τ ⊨ ∂(S::(′τ,(′α::bot))Set)
and      defa : τ ⊨ ∂(a::(′τ⇒′α::bot))
and      aninS: ¬ (τ ⊨ a ∈ S)
and      cpP  : cp P
shows           ¬ (τ ⊨ a ∈ ((|x : S | P (x::′τ⇒′α::bot)|)))
apply (insert defS defa aninS)
apply (simp add: OclSelect_def OclIncluding_def
```

```
                    OclLocalValid_def OclIsDefined_def OclTrue_def
                    OclForAll_def OclNot_def OclIncludes_def
                    o_def ss_lifting')
apply (frule DEF_X_Set', clarify, simp)
done


lemma mem_Select_charn:
assumes defS : τ ⊨ ∂(S::('τ,('α::bot))Set)
and     defa : τ ⊨ ∂(a::('τ⇒'α::bot))
and     allP : ⋀ x. τ ⊨ x ∈ S ⟹ τ ⊨  ∂(P x)
and     cpP  : cp P
shows          (τ ⊨ a ∈ (⦇x:S | P(x::'τ⇒'α::bot)⦈))=
             (τ ⊨ P a ∧ τ ⊨ a ∈ S)
apply(case_tac τ ⊨ a ∈ S, simp_all)
by   (intro mem_OclSelect_eq nonmem_OclSelect,
      simp_all add: allP defS defa cpP)+


lemma select_mem_charn:
assumes defT : τ ⊨ ∂(T::('τ,('α::bot))Set)
shows          (⦇x : S | ((x::('τ⇒'α::bot)) ∈ T)⦈) τ =
             ((S::('τ,'α)Set)  ∩ T) τ
apply(insert defT)
apply(simp add: localValidDefined2sem DEF_def)
apply(frule DEF_X_Set', clarify)
apply(rule_tac X=S in Set_sem_cases, simp_all)
apply(simp_all add: OclSelect_def OclIncludes_def
                 OclIntersection_def ss_lifting')
apply(safe)
apply(subst Abs_Set_0_inject_charn)
apply(auto)
done


lemma select_mem_charn_global[simp]:
assumes defT : ⊨ ∂(T::('τ,('α::bot))Set)
shows          (⦇x : S | ((x::('τ⇒'α::bot)) ∈ T)⦈) =
             ((S::('τ,'α)Set)  ∩ T)
apply (rule ext, insert defT)
apply (rule select_mem_charn)
apply (auto simp: global_vs_local_validity[symmetric])
done



lemma select_mem_charn_universal[simp]:
shows   (⦇x : S | ((x::('τ⇒'α::bot)) ∈ S)⦈)  =
        ((S::('τ,'α)Set))
```

apply (*rule ext*)
apply (*case_tac x* $\models$ $\partial(S)$)
apply (*simp_all, ocl_hypsubst* [*2*],*simp_all*)
apply (*rule trans*)
apply (*rule select_mem_charn*)
apply (*simp_all*)
done

defs
OclCollect_def: $OclCollect\ S\ P \equiv \lambda\,\tau.$
$if\ DEF\ (S\ \tau)$
$then\ if\ \forall\, x{\in}\ulcorner Rep\_Set\_0\ (S\ \tau)\urcorner.\ DEF\ (P\ (\lambda\tau.\ x)\ \tau)$
$then\ Abs\_Set\_0\ \llcorner(\lambda x.\ P\ (\lambda\tau.\ x)\ \tau)\ `\ \ulcorner Rep\_Set\_0\ (S\ \tau)\urcorner\lrcorner\ else\ \bot$
$else\ \bot$

Any and One

defs
OclAny_def :
$OclAny \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
$if\ \ (\forall\ x \in \ulcorner Rep\_Set\_0\ S\urcorner.\ DEF\ (P\ x)) \wedge$
$(\exists\ x \in \ulcorner Rep\_Set\_0\ S\urcorner.\ P\ x = \llcorner True\lrcorner)$
$then\ SOME\ x.(x \in \ulcorner Rep\_Set\_0\ S\urcorner \wedge P\ x = \llcorner True\lrcorner)$
$else\ \bot))$

ocl_setup_op [*OclAny*]

defs

OclOne_def :
$OclOne \equiv (lift2'\ lift\_arg0\ lift\_arg1)\ (strictify\ (\lambda\ S\ P.$
$if\ \forall\, x \in \ulcorner Rep\_Set\_0\ S\urcorner.\ DEF\ (P\ x)$
$then\ \llcorner card\ \{x \in \ulcorner Rep\_Set\_0\ S\urcorner.\ P\ x = \llcorner True\lrcorner\} = 1\lrcorner$
$else\ \bot))$

ocl_setup_op [*OclOne*]

The iterator for Sets is not adequately defined in the standard (idempotence). This can be overcome by Paulsons theory of the finite fold operator of sets: it is always defined, but only for bodies P that are ACI, there is an unfold theorem.

This definition essentially captures the intention of the standard.

defs

OclIterate_def :
$OclIterate \equiv (lift3'\ lift\_arg0\ lift\_arg2\ lift\_arg0)\ (strictify\ (\lambda\ S\ P\ A.$
$if\ \ (finite\ \ulcorner Rep\_Set\_0\ S\urcorner)$
$then\ (fold\_set\ (\lambda\ x\ y.\ P\ y\ x)\ A\ (\ulcorner Rep\_Set\_0\ S\urcorner))$

```
                  else ⊥))
ocl_setup_op[OclIterate]
```

Properties of iterate on sets

```
lemma iterate_infinite:
  τ ⊨ ∄ (‖ (X::('τ,'α::bot)Set) ‖) ⟹
   τ ⊨ ∄ (OclIterate X (P::('τ,'α) VAL ⇒ ('τ,'β::bot) VAL ⇒ ('τ,'β) VAL) A)
  apply(rule_tac X=X in Set_sem_cases, simp_all)
  apply(case_tac finite c)
  apply(simp_all add: localValidUnDefined2sem OclSize_def OclIterate_def
                      ss_lifting')
  done
```

Using this theorem all requirements proven about iterate on bags will carry over to iterate on *finite* sets quite easily. Because they are actually equivalent in this case.

```
lemma iterate_set2bag:
  τ ⊨ ∂ (‖ (X::('τ,'α::bot)Set) ‖) ⟹
   τ ⊨ (OclIterate X (P::('τ,'α) VAL ⇒ ('τ,'β::bot) VAL ⇒ ('τ,'β) VAL) A) ≜
     (OclIterate ((->asBag() X)::('τ,'α::bot)Bag) P A)
  apply(rule_tac X=X in Set_sem_cases, simp_all)
  apply(simp_all add: localValidDefined2sem)
  apply(case_tac finite c)
  apply(simp_all add: OclSize_def OclIterate_def OclLocalValid_def
                      OclStrongEq_def OCL_Bag.OclIterate_def
                      fold_set_fold_multiset_conv
                      OclAsBag_def OclTrue_def ss_lifting')
  done
end
```

B.4.14. OclAny

```
theory OCL_OclAny
imports
    $HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_Set
begin
```

Level 0

```
setup ⟪ [Theory.add_path UML_OCL.level0] ⟫

constdefs
  mk_OclAny        :: 'α OclAny_0 ⇒ 'α U
  mk_OclAny        ≡ Inl
  get_OclAny       :: 'α U ⇒ 'α OclAny_0
  get_OclAny       ≡  sum_case (λx. x) (λx. ϵ x. True)
  is_OclAny        ::  'α OclAny_0 ⇒ bool
  is_OclAny obj  ≡ DEF(obj)
  is_OclAny_univ  :: 'α U ⇒ bool
```

```
  is_OclAny_univ ≡  sum_case (λx. True) (λx. False)

lemma get_mk_OclAny_id: get_OclAny(mk_OclAny x) = x
  by (simp add: get_OclAny_def mk_OclAny_def)

lemma mk_get_OclAny_id: is_OclAny_univ x ⟹ mk_OclAny(get_OclAny x) = x
   apply (simp add: get_OclAny_def mk_OclAny_def is_OclAny_univ_def)
  apply (case_tac x, simp, simp)
  done

lemma is_OclAny_univ_implies_is_get: ⟦DEF(get_OclAny x); is_OclAny_univ x ⟧
  ⟹ is_OclAny(get_OclAny x)
 apply (simp add: get_OclAny_def mk_OclAny_def is_OclAny_univ_def is_OclAny_def)
  done


lemma is_mk_OclAny:
  ⟦ is_OclAny obj ⟧ ⟹ is_OclAny_univ (mk_OclAny obj)
  apply (simp add: is_OclAny_univ_def  get_OclAny_def is_OclAny_def
                  mk_OclAny_def)
  done

setup ⟪ [Theory.add_path /] ⟫
```

Level 1

```
setup ⟪ [Theory.add_path UML_OCL.level1] ⟫

constdefs
  mk_OclAny       :: ('τ , 'α OclAny_0) VAL ⇒ ('τ , 'α U) VAL
  mk_OclAny       ≡ lift1 level0.mk_OclAny
  get_OclAny      :: ('τ , 'α U) VAL ⇒ ('τ , 'α OclAny_0) VAL
  get_OclAny      ≡ lift₁ level0.get_OclAny
  is_OclAny       ::   ('τ ,'α OclAny_0) VAL   ⇒  ('τ)Boolean
  is_OclAny  ≡ lift₁(strictify (lift ∘ level0.is_OclAny))
  is_OclAny_univ  :: ('τ ,'α U ) VAL   ⇒  ('τ)Boolean
  is_OclAny_univ ≡  lift₁(lift ∘ level0.is_OclAny_univ)
    OclAny  :: ('τ, 'α OclAny_0 Set_0) VAL
  OclAny  ≡ lift0(Abs_Set_0 ⌊lift ‘ UNIV⌋)

lemma get_mk_OclAny_id :
  τ⊨ level1.is_OclAny obj ⟹  level1.get_OclAny (level1.mk_OclAny obj) = obj
  apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def
    level1.mk_OclAny_def OclLocalValid_def)
  apply(rule ext)
  apply(simp add: lift1_def level0.get_mk_OclAny_id)
done

lemma mk_get_OclAny_id :
```

```
  τ⊨ level1.is_OclAny_univ univ
   ⟹ level1.mk_OclAny (level1.get_OclAny univ) τ= univ τ
  apply (frule isDefined_if_valid)
  apply (frule localValidDefined2sem [THEN iffD1])
  apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def
              level1.mk_OclAny_def OclLocalValid_def)
 apply(simp add: OclLocalValid_def lift1_def lift0_def level0.mk_get_OclAny_id OclTrue_def)
done

lemma is_OclAny_univ_implies_is_get:
  ⟦ DEF(level1.get_OclAny univ τ); (τ⊨ level1.is_OclAny_univ univ) ⟧
   ⟹ (τ⊨ level1.is_OclAny (level1.get_OclAny univ))
  apply (frule isDefined_if_valid)
  apply (frule localValidDefined2sem [THEN iffD1])
  apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def
    level1.mk_OclAny_def OclLocalValid_def)
 apply(simp add: OclLocalValid_def lift1_def lift0_def level0.is_OclAny_univ_implies_is_get

                  OclTrue_def strictify_def DEF_def )
done

lemma is_mk_OclAny:
  τ⊨ level1.is_OclAny obj
   ⟹ (τ⊨ level1.is_OclAny_univ (level1.mk_OclAny obj))
  apply (frule isDefined_if_valid)
  apply (frule localValidDefined2sem [THEN iffD1])
  apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def
    level1.mk_OclAny_def OclLocalValid_def)
 apply(simp add: OclLocalValid_def lift1_def lift0_def OclTrue_def strictify_def DEF_def
)
  apply(case_tac obj τ= down)
  apply(auto simp: level0.is_mk_OclAny)
done


setup ⟪ [Theory.add_path /] ⟫
```

Level 2

```
setup ⟪ [Theory.add_path UML_OCL] ⟫

constdefs
  mk_OclAny       :: ('τ , 'α OclAny_0) VAL ⇒ ('τ , 'α U) VAL
  mk_OclAny       ≡ level1.mk_OclAny
  get_OclAny      :: ('τ , 'α U) VAL ⇒ ('τ , 'α OclAny_0) VAL
  get_OclAny      ≡ level1.get_OclAny
  is_OclAny       ::  ('τ ,'α OclAny_0) VAL  ⇒  ('τ)Boolean
  is_OclAny  ≡  level1.is_OclAny
  is_OclAny_univ  :: ('τ ,'β U ) VAL  ⇒  ('τ) Boolean
```

is_OclAny_univ ≡ *level1.is_OclAny_univ*
OclAny :: ($'\tau$, $'\alpha$ *OclAny_0 Set_0*) *VAL*
OclAny ≡ *level1.OclAny*

lemma *get_mk_OclAny_id* :
$\tau \models$ *UML_OCL.is_OclAny obj*
$\Longrightarrow$ *UML_OCL.get_OclAny* (*UML_OCL.mk_OclAny obj*) = *obj*
apply (*unfold UML_OCL.is_OclAny_def UML_OCL.get_OclAny_def UML_OCL.mk_OclAny_def*)
apply(*simp add*: *level1.get_mk_OclAny_id*)
done

lemma *mk_get_OclAny_id* :
$\tau \models$ *UML_OCL.is_OclAny_univ univ*
$\Longrightarrow$ *UML_OCL.mk_OclAny* (*UML_OCL.get_OclAny univ*) τ= *univ* τ
apply (*unfold UML_OCL.is_OclAny_def UML_OCL.get_OclAny_def UML_OCL.mk_OclAny_def*

UML_OCL.is_OclAny_univ_def)
apply(*simp add*: *level1.mk_get_OclAny_id*)
done

lemma *is_OclAny_univ_implies_is_get*:
⟦ *DEF*(*UML_OCL.get_OclAny univ* τ); ($\tau \models$ *UML_OCL.is_OclAny_univ univ*) ⟧
$\Longrightarrow$ ($\tau \models$ *UML_OCL.is_OclAny* (*UML_OCL.get_OclAny univ*))
apply (*unfold UML_OCL.is_OclAny_def UML_OCL.get_OclAny_def UML_OCL.mk_OclAny_def*

UML_OCL.is_OclAny_univ_def)
apply(*simp add*: *level1.is_OclAny_univ_implies_is_get*)
done

lemma *is_mk_OclAny*:
$\tau \models$ *UML_OCL.is_OclAny obj*
$\Longrightarrow$ ($\tau \models$ *UML_OCL.is_OclAny_univ* (*UML_OCL.mk_OclAny obj*))
apply (*unfold UML_OCL.is_OclAny_def UML_OCL.get_OclAny_def UML_OCL.mk_OclAny_def*

UML_OCL.is_OclAny_univ_def)
apply(*simp add*: *level1.is_mk_OclAny*)
done

setup ⟪ [*Theory.add_path* /] ⟫

Re-typing or Casting.

The standard describes an operation that "casts" an object to one of its subtypes. Note that our OCL-encoder provides for each new class an own definition for this polymorphic scheme.

consts
OclAsType ::[($'\tau$,$'\alpha$::*bot*) *VAL*,($'\tau$,$'\beta$ *Set_0*) *VAL*] ⇒ ($'\tau$,$'\beta$::*bot*) *VAL*
(_ −>*oclAsType*′(_ ′) [66,65]65)

Checking Dynamic Types of "Kinds"

As a consequence of the difference between dynamic and static typing in object-oriented languages, a test for the "kind" of an object is provided. It checks for the dynamic type

Note that this operator is in fact defined by our OCL encoder and uses the internally defined test functions.

constdefs
OclIsKindOf :: $[('\tau,'\alpha::bot)\ VAL,\ ('\tau,'\beta::collection)\ VAL] \Rightarrow '\tau\ Boolean$
$(\_ \ {-}{>}oclIsKindOf'(\ \_\ ')\ [66,65]65)$
OclIsKindOf self CharSet ≡ *self* ∈ *CharSet*

This simple fundamental definition is the fruit of our generalization of the OCL set theory.

constdefs
OclIsTypeOf :: $[('\tau,\ '\alpha::bot)\ VAL,\ ('\tau,\ '\beta::collection)\ VAL] => '\tau\ Boolean$
$(\_ \ {-}{>}oclIsTypeOf'(\ \_\ ')\ [66,65]65)$
OclIsTypeOf self NarrowCharSet ≡ *self* ∈ *NarrowCharSet*

end

B.4.15. CharactersticSets

theory *OCL_CharactersticSet*
imports
$HOLOCL_HOME/src/library/basic/OCL_Real
$HOLOCL_HOME/src/library/basic/OCL_String
$HOLOCL_HOME/src/library/OclAny/OCL_OclAny
begin

Embedding Characteristic Sets into OCL

lemma *Integer_0_Defined* [*simp*]:
$\tau \models \partial$ *Integer_0*
by(*simp add: localValidDefined2sem Abs_Set_Mt_DEF Integer_0_def lift0_def*)

lemma *in_Integer_0*:
$\tau \models \partial\ x \Longrightarrow$
$\tau \models (x::'a\ Integer) \in (Integer\_0::('a,Integer\_0)Set)$
apply (*simp add: OCL_Set.OclIncludes_def Integer_0_def localValidDefined2sem*
OclLocalValid_def OclIsDefined_def OclTrue_def
lift0_def lift1_def lift2_def strictify_def
Abs_Set_rangelift_DEF[simplified DEF_def])
by (*auto simp:DEF_def not_down_exists_lift*)

lemma *in_Integer_0R*[*simp*]:

$(\tau \vDash (x::'a\ Integer) \in (Integer\_0::('a,Integer\_0)Set)) =$
$(\tau \vDash \partial\ x)$
by (*auto simp*: *in_ Integer_ 0*
OCL_ Logic_ core.isDefined_ if_ valid
[THEN OCL_ Set.OCL_ is_ defopt_ OclIncludes
[THEN iffD1]])

The relevance of the latter type of rules become apparent if we look at the following example:

lemma *test*:
$(\forall\ x \in Integer\_0\ .\ (x + (OclNegative\ x) \doteq 0)) = \mathsf{T}$
oops

First of all, such an expression is syntactically illegal in OCL. In Standard syntax, this is presented as

```
Integer_0 ->forAll( x | x + (-x) = 0)
```

`Integer_0` is considered a type name which is not part of the expression language; therefore, no quantification over them is possible in standard OCL. Since we interpret OCL types as sets in the sense of the OCL_ Set theory, we can quantify and reason over it in HOL-OCL.

In particular, rule *forAllI* eliminates the quantifier and reduces this to

$$\tau \models x \in Integer_0 \Longrightarrow (x + (OclNegativex) \doteq 0)) = \mathsf{T} \tag{B.15}$$

which can be further simplified by *in_ Integer_ 0R* to

$$\tau \models \partial x \Longrightarrow (x + (-x) \doteq 0)) = \mathsf{T} \tag{B.16}$$

This is a valid statement in OCL (although we ommit the proof here). In contrast,

$$(x + (-x) \doteq 0)) = \mathsf{T} \tag{B.17}$$

in itself is not valid since x may be undefined. Thus, quantifiers range over defined values only (as one would expect). These theorems also implicitly state that the characteristic sets of the basic types are non-empty.

lemma *Real_ Defined* [*simp*]:
$\tau \vDash \partial\ Real\_0$
by(*simp add*: *localValidDefined2sem Real_ 0_ def lift0_ def*)

lemma *in_ Real_ 0*:
$\tau \vDash \partial\ x \Longrightarrow$
$\tau \vDash (x::'a\ Real) \in (Real\_0::('a,Real\_0)Set)$
apply (*simp add*: *OCL_ Set.OclIncludes_ def Real_ 0_ def localValidDefined2sem*
OclLocalValid_ def OclIsDefined_ def OclTrue_ def
lift0_ def lift1_ def lift2_ def strictify_ def
Abs_ Set_ rangelift_ DEF[simplified DEF_ def])

```
by     (auto simp:DEF_def not_down_exists_lift)

lemma   in_RealR_0[simp]:
(τ ⊨ (x::'a Real) ∈ (Real_0::('a,Real_0)Set)) =
 (τ ⊨ ∂ x)
by (auto simp: in_Real_0
              isDefined_if_valid
                 [THEN OCL_Set.OCL_is_defopt_OclIncludes
                        [THEN iffD1]])



lemma Boolean_Defined [simp]:
τ ⊨ ∂ Boolean_0
by(simp add: localValidDefined2sem Boolean_0_def lift0_def)

lemma   in_Boolean:
τ ⊨ ∂ x ⟹
 τ ⊨ (x::'a Boolean) ∈ (Boolean_0::('a,Boolean_0)Set)
apply (simp add: OCL_Set.OclIncludes_def Boolean_0_def localValidDefined2sem
               OclLocalValid_def OclIsDefined_def OclTrue_def
               lift0_def lift1_def lift2_def strictify_def
               Abs_Set_rangelift_DEF[simplified DEF_def])
by     (auto simp:DEF_def not_down_exists_lift)

lemma   in_BooleanR[simp]:
(τ ⊨ (x::'a Boolean) ∈ (Boolean_0::('a,Boolean_0)Set)) =
 (τ ⊨ ∂ x)
by (auto simp: in_Boolean
              isDefined_if_valid
                 [THEN OCL_Set.OCL_is_defopt_OclIncludes
                        [THEN iffD1]])


lemma String_0_Defined [simp]:
τ ⊨ ∂ String_0
by(simp add: localValidDefined2sem String_0_def lift0_def)

lemma   in_String_0:
τ ⊨ ∂ x ⟹
 τ ⊨ (x::'a String) ∈ (String_0::('a,String_0)Set)
apply (simp add: OCL_Set.OclIncludes_def String_0_def localValidDefined2sem
               OclLocalValid_def OclIsDefined_def OclTrue_def
               lift0_def lift1_def lift2_def strictify_def
               Abs_Set_rangelift_DEF[simplified DEF_def])
by     (auto simp:DEF_def not_down_exists_lift)

lemma   in_StringR[simp]:
(τ ⊨ (x::'a String) ∈ (String_0::('a,String_0)Set)) =
```

```
 (τ ⊨ ∂ x)
by (auto simp: in_String_0
          isDefined_if_valid
            [THEN OCL_Set.OCL_is_defopt_OclIncludes
                  [THEN iffD1]])


lemma OclAny_Defined [simp]:
τ ⊨ ∂ OclAny
by(simp add: localValidDefined2sem OclAny_def level1.OclAny_def lift0_def)

lemma  in_OclAny:
τ ⊨ ∂ x ⟹
 τ ⊨ (x::('a,'b) OclAny) ∈ (OclAny::('a,'b OclAny_0)Set)
apply (simp add: OCL_Set.OclIncludes_def OclAny_def level1.OclAny_def
            localValidDefined2sem
            OclLocalValid_def OclIsDefined_def OclTrue_def
            lift0_def lift1_def lift2_def strictify_def
            Abs_Set_rangelift_DEF[simplified DEF_def])
by   (auto simp:DEF_def not_down_exists_lift)


lemma  in_OclAnyR[simp]:
 τ ⊨ (x::('a,'b) OclAny) ∈ (OclAny::('a,'b OclAny_0)Set) =
 (τ ⊨ ∂ x)
by (auto simp: in_OclAny
          isDefined_if_valid
            [THEN OCL_Set.OCL_is_defopt_OclIncludes
                  [THEN iffD1]])
end
```

B.4.16. The OCL Library

```
theory OCL_Library
imports
 $HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_CharactersticSet
begin
```

Datatype Conversions

```
consts
  HolOclToReal    :: 'a Integer => 'a Real       (_ ->toReal'( ') [66])
defs
  HolOclToReal_def:
  HolOclToReal      ≡ lift₁(strictify(λ x::Integer_0.
                      ⌊ real ( ⌈x⌉ ) ⌋ ))
```

end

B.5. State

B.5.1. OCL State

theory *OCL_State*
imports
 $HOLOCL_HOME/src/library/OclAny/OCL_OclAny
begin types
$'a\ state\ \ = oid \rightharpoonup 'a$
$'s\ St\ \ \ \ = 's\ state \times 's\ state$
$('a,'t)\ V = ('a\ St, 't)\ VAL$

translations
$a\ state\ <= (type)\ oid \Rightarrow a\ option$
$a\ St\ \ <= (type)\ (a\ state) \times (s\ state)$
$(a,t)\ V\ <= (type)\ a\ St \Rightarrow t$

To sum up, the HOL types assigned to the key OCL types of the OCL standard are a context (st) indexed family of types. Recall: the contexts are a state relation or a Kripke-Structure over states over the universe.

Provided that we have a representation of class types generated from an UML class diagram, we can therefore translate any OCL type into an HOL-type achieving type soundness on the HOL-level.

Checking Freshness of Object Instances in the State.

constdefs
$OclIsNew\ :: ('a\ U\ St,'b::bot)\ VAL\ \Rightarrow ('a\ U\ St)\ Boolean$
$OclIsNew\ obj \equiv (\lambda(s,s').\ \llcorner level0.mk\_OclAny\ (((obj)\ {-}{>}oclAsType(OclAny))\ (s,s')) \notin ran\ s$
$\wedge\ level0.mk\_OclAny\ (((obj)\ {-}{>}oclAsType(OclAny)\ )\ (s,s')) \in ran\ s' \lrcorner)$

The OCL encoder must generate a coercion in order to make this operation applicable to any object of some class-type

Selecting the Objects living in a State.

consts
$OclAllInstances\ \ \ :: ('a\ U\ St,'b)\ VAL \Rightarrow ('a\ U\ St,'b\ Set\_0)\ VAL$
$OclAllInstancesAtpre :: ('a\ U\ St,'b)\ VAL \Rightarrow ('a\ U\ St,'b\ Set\_0)\ VAL$

defs
OclAllInstancesString_def[simp]:
$OclAllInstances\ (self::('a\ U\ St,String\_0)\ VAL) \equiv OclUndefined$

OclAllInstancesPreString_def[simp]:

OclAllInstancesAtpre $(self::('a\ U\ St,String\_0)\ VAL) \equiv$ *OclUndefined*

defs
OclAllInstancesInteger_def[simp]:
OclAllInstances $(self::('a\ U\ St,Integer\_0)\ VAL) \equiv$ *OclUndefined*

OclAllInstancesPreInteger_def[simp]:
OclAllInstancesAtpre $(self::('a\ U\ St,Integer\_0)\ VAL) \equiv$ *OclUndefined*

defs
OclAllInstancesReal_def[simp]:
OclAllInstances $(self::('a\ U\ St,Real\_0)\ VAL) \equiv$ *OclUndefined*

OclAllInstancesPreReal_def[simp]:
OclAllInstancesAtpre $(self::('a\ U\ St,Real\_0)\ VAL) \equiv$ *OclUndefined*

defs
OclAllInstancesBoolean_def[simp]:
OclAllInstances $(self::('a\ U\ St,Boolean\_0)\ VAL) \equiv Boolean\_0$

OclAllInstancesPreBoolean_def[simp]:
OclAllInstancesAtpre $(self::('a\ U\ St,Boolean\_0)\ VAL) \equiv Boolean\_0$

And now the general case for all "objects", i.e. all instances of subclass OclAny:

defs
OclAllInstancesOclAny_def:
OclAllInstances $(self::('a\ U\ St,\ 'a\ OclAny\_0)\ VAL) \equiv$
$\lambda(s,s').\ Abs\_Set\_0\ (\llcorner level0.get\_OclAny\ `\ ran\ s'\lrcorner)$

OclAllInstancesPreOclAny_def:
OclAllInstancesAtpre $(self::('a\ U\ St,\ 'a\ OclAny\_0)\ VAL) \equiv$
$\lambda(s,s').\ Abs\_Set\_0\ (\llcorner level0.get\_OclAny\ `\ ran\ s\lrcorner)$

The `modifiedOnly()`-clause and its Infrastructure

constdefs *OclOidOf* $::\ 'a\ U\ state \Rightarrow\ 'a\ OclAny\_0 \Rightarrow oid\ set$
$OclOidOf\ s\ X \equiv (\{x\ .\ (s\ x) = Some(OCL\_OclAny.mk\_OclAny\ X)\})$

The oid `Set` is a tribute to non-referential universes where there may be different object identifiers in the store for the same object. The redtriction to $'\alpha\ OclAny\_0$ assures that the projection is only applicable to objects.

constdefs *OclModifiedOnly* $::\ (('a\ U\ St,'a\ OclAny\_0)\ Set) \Rightarrow ('a\ U\ St)\ Boolean$
$OclModifiedOnly\ X \equiv (\lambda(s,s').$
$\llcorner\forall\ id \in -(\bigcup\ (OclOidOf\ s)\ `\ulcorner Rep\_Set\_0(X(s,s'))\urcorner).\ s\ id = s'\ id\lrcorner)$

end

B.5.2. The OCL Calculi

theory *OCL_Calculi*
imports
  $*HOLOCL_HOME/src/OCL_Library*
begin

This theory develops some more experimental calculi for OCL such as the local judgement tableaux calculus (**lct!**) or a particular congruence rewriting calculus. It is the basis of some more experimental proof procedures so far.

Some core tableaux calculus

The conceptual basis of this calculus is drawn from [28] where "labeled formulas" where introduced. These are:

- A^{t} (for $A\,s = \mathsf{T}\,s$),
- A^{f} (for $A\,s = \mathsf{F}\,s$)
- A^{u} (for $A\,s = \bot\,s$)

A "labeled clause" is then a set of labelled formulas (considered as disjunctions). In their notation, the crucial rules for labeled clauses are:

$$\frac{C, (A \wedge B)^{\mathsf{t}}}{C, A^{\mathsf{t}} \quad B^{\mathsf{t}}} \qquad \frac{C, (A \wedge B)^{\mathsf{f}}}{C, A^{\mathsf{f}} \quad B^{\mathsf{f}}} \frac{C, (A \wedge B)^{\mathsf{u}}}{C, A^{\mathsf{u}} \quad B^{\mathsf{u}} \quad C, A^{\mathsf{u}}, B^{\mathsf{u}}} \tag{B.18}$$

$$\frac{C, (\neg A)^{\mathsf{t}}}{C, A^{\mathsf{f}}} \qquad \frac{C, (\neg A)^{\mathsf{f}}}{C, A^{\mathsf{t}}} \qquad \frac{C, (\neg A)^{\mathsf{u}}}{C, A^{\mathsf{u}}} \tag{B.19}$$

Tableau-calculi were implemented in Isabelle not on the basis of clauses, but on Horn formula $[\![A_1; \ldots; A_n]\!] \Longrightarrow B$ corresponding to $\neg A_1, \ldots A_n, B$ in clause representation. The conclusion in the Horn-format is used with preference during proof search.

As a consequence, it is necessary to duplicate the set of rules into introduction- and elimination rules. Further, we use the validity normal form to represent negation; thus the handling of the not is different and has to be encoded into the rules (leading to further duplications of the rule sets), this is necessary in Isabelle's representation of tableaux calculi anyway.

lemma *and_TRUE_I*:
  $[\![\ \tau \vDash A;\ \tau \vDash B\ ]\!] \Longrightarrow\ \tau \vDash (A \wedge B)$
  by (*erule subst_LJ_TRUE*, *erule subst_LJ_TRUE*, *auto*)

lemma *and_D1*:
  $[\![\ \tau \vDash A \wedge B\ ]\!] \Longrightarrow\ \tau \vDash A$
  apply (*simp_all add*: *OclAnd_def OclLocalValid_def*

```
                        DEF_def lift2_def split add: split_if)
  apply (case_tac A τ = down)
  apply (simp_all add: OclTrue_def lift0_def)
  apply (simp add: split_if_eq1)
  apply (case_tac B τ = down)
  apply (simp_all add: split_if_eq1 not_down_exists_lift)
  apply (erule exE)
  apply (erule exE)
  apply (simp_all add: drop_lift)
  done



lemma and_D2:
  ⟦ τ ⊨ A ∧ B ⟧ ⟹ τ ⊨ B
  by (rule and_D1, subst and_commute, assumption)



lemma and_TRUE_E:
  ⟦ τ ⊨ (A ∧ B); ⟦ τ ⊨ A; τ ⊨ B ⟧ ⟹ R ⟧ ⟹ R
  by(frule and_D1, drule and_D2, auto)



lemma and_congr':
  (τ ⊨ (A ∧ B)) = ((τ ⊨ A) ∧ (τ ⊨ B))
by(auto elim: and_TRUE_E and_D1 and_D2 intro: and_TRUE_I)

lemma not_and_undef_congr1[simp]:
  (τ ⊨ ¬ (⊥ ∧ B)) = (τ ⊨ ¬ B)
  apply (cut_tac τ = τ and A = B in non_quatrium_datur)
  apply auto
  apply (drule subst_LJ_TRUE_fw, assumption, auto)
  apply (erule_tac subst_LJ_FALSE, auto)
  by (drule_tac subst_LJ_undef_fw,assumption, auto)

lemma or_undef_congr1[simp]:
  (τ ⊨ ⊥ ∨ B) = (τ ⊨ B)
  by (simp add: OclOr_def)

lemma not_and_undef_congr2[simp]:
  (τ ⊨ ¬ (B ∧ ⊥)) = (τ ⊨ ¬ B)
  by(subst and_commute,simp add: not_and_undef_congr1)

lemma or_undef_congr2[simp]:
  (τ ⊨ B ∨ ⊥) = (τ ⊨ B)
  by (simp add: OclOr_def)
```

lemma *and_undef_congr3*[*simp*]:
 $\sim(\tau \models \bot \wedge B)$
 apply (*cut_tac* $\tau = \tau$ **and** $A = B$ **in** *non_quatrium_datur*)
 apply *auto*
 apply (*drule_tac* $P = \lambda B.\ (\bot \wedge B)$ **in** *subst_LJ_TRUE_fw_rev*)
 apply *simp_all*
 by (*drule_tac subst_LJ_undef_fw,assumption, auto*)

lemma *and_undef_congr4*[*simp*]:
 $\sim(\tau \models B \wedge \bot)$
 by(*subst and_commute,simp add*: *not_and_undef_congr1*)

lemma *and_FALSE_I_core*:
 $[\![(\tau \models \neg A) \vee (\tau \models \neg B)]\!] \Longrightarrow \tau \models \neg(A \wedge B)$
 by (*auto elim*: *subst_LJ_FALSE*)

lemmas *and_FALSE_I* = *disjCI* [*THEN and_FALSE_I_core, standard*]

lemma *and_FALSE_I1*:
 $[\![(\tau \models B) \vee (\tau \models (\not\partial\ B)) \Longrightarrow (\tau \models \neg A)]\!] \Longrightarrow \tau \models \neg(A \wedge B)$
 by (*rule and_FALSE_I, simp only*: *not_invalid*)
lemma *and_FALSE_E*:
 $[\![\tau \models \neg(A \wedge B); \tau \models \neg A \Longrightarrow R; \tau \models \neg B \Longrightarrow R]\!] \Longrightarrow R$
 apply (*cut_tac* $\tau = \tau$ **and** $A = A$ **in** *non_quatrium_datur*)
 apply *auto*
 apply (*rotate_tac* [*1*] −*1*)
 apply (*rotate_tac* [*2*] −*1*)
 apply (*drule_tac* [*2*] *subst_LJ_undef_fw*)
 apply (*drule subst_LJ_TRUE_fw*)
 apply *assumption*
 prefer *3*
 apply *assumption*
 apply *auto*
 done

lemma *or_2_classic_disj*:
 $[\![\tau \models \partial\ X; \tau \models \partial\ Y]\!] \Longrightarrow (\tau \models X \vee Y) = ((\tau \models X) \vee (\tau \models Y))$
 apply *auto*
 defer *1* **apply** (*erule_tac* $P = \lambda\ X\ St.\ (X \vee Y)\ St$ **in** *subst_LJ_TRUE*)
 prefer *3* **apply** (*erule_tac* $P = \lambda\ Y\ St.\ (X \vee Y)\ St$ **in** *subst_LJ_TRUE*)
 apply (*auto simp*: *not_valid*)
 apply (*rotate_tac* −*1*)
 apply (*drule_tac* $P = \lambda\ Y\ St.\ (X \vee Y)\ St$ **in** *subst_LJ_FALSE_fw*)
 apply *auto*
 done

```
lemma isdef_and_congr:
  (τ ⊨ ∂ (X ∧ Y)) = ((τ ⊨ ∂ X ∧ ∂ Y) ∨
                      (τ ⊨ ∂ X ∧ ¬ X)   ∨
                      (τ ⊨ ∂ Y ∧ ¬ Y))
  apply (auto simp: isDefined_andD0 or_2_classic_disj or_2_classic_disj)
  apply (simp_all add: or_2_classic_disj)
  done

lemma solution0:
  ⟦ ((τ ⊨ ∂ X)  ∧ (τ ⊨  ∂ Y)) ∨
    ((τ ⊨ ∂ X)  ∧ (τ ⊨ ¬ X)) ∨
    ((τ ⊨ ∂ Y)  ∧ (τ ⊨ ¬ Y)) ⟧
 ⟹  τ ⊨ ∂(X ∧ Y)
 by (simp add: isdef_and_congr and_congr')


lemma solution1:
  ⟦ ⟦(∂ B ∧ ¬ B)τ ~= T τ;(∂ A ∧ ¬ A)τ ~= T τ ⟧
    ⟹ (τ ⊨ ∂ A) ∧ (τ ⊨ ∂ B) ⟧
 ⟹  τ ⊨ ∂(A ∧ B)
 apply (simp add: OclLocalValid_def[symmetric])
 apply (rule solution0)
 apply (cases ¬ τ ⊨ ∂ B  ∧ ¬ B, simp_all)
 apply (cases ¬ τ ⊨ ∂ A  ∧ ¬ A, simp_all)
 apply(auto elim!: and_TRUE_E)
 done


lemma solution2:
 ⟦  ⟦  τ ⊨ ¬(∂ B ∧ ¬ B); τ ⊨  ¬(∂ A ∧ ¬ A) ⟧
    ⟹ (τ ⊨ ∂ A) ∧ (τ ⊨ ∂ B) ⟧
 ⟹  τ ⊨ ∂(A ∧ B)
 apply (rule solution1)
 apply (simp only: OclLocalValid_def[symmetric] not_valid)
 apply auto
 done


lemma isDefined_and_TRUE_I:
 ⟦ ⟦ (τ ⊨ ∅ B) ∨ (τ ⊨ B);(τ ⊨ ∅ A) ∨ (τ ⊨ A) ⟧
    ⟹ (τ ⊨ ∂ A) ∧(τ ⊨ ∂ B) ⟧
  ⟹  τ ⊨ ∂(A ∧ B)
 apply (rule solution2)
 apply (auto elim!: and_FALSE_E and_TRUE_E simp add: isDefined_andD0 or_def)
 done


lemma solution3:
```

$\llbracket$ $\llbracket$~$(\tau \vDash \neg\ A)$; ~$(\tau \vDash \neg\ B)$ $\rrbracket \Longrightarrow (\tau \vDash \partial\ A)$;
$\llbracket$~$(\tau \vDash \neg\ A)$; ~$(\tau \vDash \neg\ B)$ $\rrbracket \Longrightarrow (\tau \vDash \partial\ B)$ $\rrbracket$
$\Longrightarrow\ \tau \vDash \partial(A \wedge B)$
apply (*rule isDefined_ and_ TRUE_ I*)
apply (*simp only: not_ invalid*)
apply *auto*
done

lemma *isDefined_ and_ TRUE_ E*:
$\llbracket\ \tau \vDash \partial(A \wedge B)$;
$\llbracket\ \tau \vDash \partial\ A;\ \tau \vDash \partial\ B \rrbracket \Longrightarrow R$;
$\tau \vDash \neg\ A \Longrightarrow R$;
$\tau \vDash \neg\ B \Longrightarrow R$
$\rrbracket \Longrightarrow R$
by (*auto elim!: and_ FALSE_ E and_ TRUE_ E*
simp add: isDefined_ andD0 OclOr_ def)

The following rule follows the principal equality:

$$\tau \vDash \not\partial(A \wedge B) = \tau \vDash ((\not\partial A \vee \not\partial B) \wedge (\not\partial A \vee A) \wedge (\not\partial B \vee B)) \qquad \text{(B.20)}$$

Via *isDefined_ andD0*, *de_ morgan1*, and *de_ morgan2* one gets:

$$= \tau \vDash (\neg(\not\partial A \wedge \not\partial B) \wedge \neg(\not\partial A \wedge \neg A) \wedge \neg(\not\partial B \wedge \neg B)) \qquad \text{(B.21)}$$

lemma *isDefined_ and_ FALSE_ I*:
$\llbracket\ \ \tau \vDash \neg\ (\partial\ A \wedge \partial\ B)$;
$\tau \vDash \neg\ (\partial\ A \wedge \neg\ A)$;
$\tau \vDash \neg\ (\partial\ B \wedge \neg\ B)\quad \rrbracket$
$\Longrightarrow\ \tau \vDash \not\partial(A \wedge B)$
by (*auto intro!: and_ TRUE_ I and_ FALSE_ I elim!: and_ FALSE_ E and_ TRUE_ E*
simp add: isDefined_ andD0 OclOr_ def)

lemma *isDefined_ and_ FALSE_ I1*:
$\llbracket\ \ ((\tau\ \vDash \not\partial\ A) \wedge$ ~$(\tau \vDash \neg\ B)) \vee$
(~$(\tau \vDash \neg\ A) \wedge (\tau \vDash \not\partial\ B))\quad \rrbracket$
$\Longrightarrow\ \tau \vDash \not\partial(A \wedge B)$
apply (*rule isDefined_ and_ FALSE_ I*)
apply (*simp_ all add: not_ valid*)
apply (*auto elim!: subst_ LJ_ undef subst_ LJ_ TRUE subst_ LJ_ FALSE*)
done

lemma *isDefined_ and_ FALSE_ I2*:
$\llbracket\ \ ((\tau \vDash \neg\ A) \vee \neg(\tau \vDash \not\partial\ B)) \Longrightarrow \tau \vDash \not\partial\ A$;
$((\tau \vDash \neg\ A) \vee \neg(\tau \vDash \not\partial\ B)) \Longrightarrow \neg(\tau \vDash \neg\ B)\ \rrbracket$
$\Longrightarrow\ \tau \vDash \not\partial(A \wedge B)$
apply (*rule isDefined_ and_ FALSE_ I1*)

apply(*auto simp del*: *isDefined_if_valid isDefined_if_invalid*)
done

lemma *isDefined_and_FALSE_I3*:
assumes A: $((\tau \vDash \partial\!\!\!/\ B) \Longrightarrow (\tau \vDash \neg\ A)) \Longrightarrow \tau \vDash \partial\!\!\!/\ A$
and B: $((\tau \vDash \partial\!\!\!/\ B) \Longrightarrow (\tau \vDash \neg\ A)) \Longrightarrow {}^{\sim}(\tau \vDash \neg\ B)$
shows $\tau \vDash \partial\!\!\!/(A \wedge B)$
apply (*rule isDefined_and_FALSE_I2*)
apply (*rule A*)
apply (*rule_tac* [*2*] *B*)
apply *auto*
done

lemma *isDefined_and_FALSE_I4*:
$[\![\ \tau \vDash (\partial\ B) \Longrightarrow (\tau \vDash \partial\!\!\!/\ A);$
$\tau \vDash (\partial\ A) \Longrightarrow (\tau \vDash A);$
$\tau \vDash (\partial\ B) \Longrightarrow (\tau \vDash B)\ ]\!]$
$\Longrightarrow \tau \vDash \partial\!\!\!/(A \wedge B)$
apply (*rule isDefined_and_FALSE_I*)
apply (*rule and_FALSE_I*)
prefer *2*
apply (*subst and_commute*)
apply (*rule_tac and_FALSE_I*)
prefer *3*
apply (*subst and_commute*)
apply (*rule_tac and_FALSE_I*)
apply(*auto simp del*: *isDefined_if_valid isDefined_if_invalid*)
done

lemma *isDefined_and_FALSE_I5*:
$[\![\ (\tau \vDash \partial\!\!\!/\ A) \vee (\tau \vDash \partial\!\!\!/\ B);$
$(\tau \vDash \partial\!\!\!/\ A) \vee (\tau \vDash A);$
$(\tau \vDash \partial\!\!\!/\ B) \vee (\tau \vDash B)\ ]\!]$
$\Longrightarrow \tau \vDash \partial\!\!\!/(A \wedge B)$
apply(*simp add*: *isDefined_andD0 de_morgan1 de_morgan2*
and_congr′ or_2_classic_disj)
apply(*auto elim!*: *subst_LJ_undef subst_LJ_TRUE subst_LJ_FALSE*
simp del: *isDefined_if_valid*
not_isUndefined_eq_isDefined)
done

lemma *isDefined_and_FALSE_E*:
$[\![\ \tau \vDash \partial\!\!\!/(A \wedge B);$
$[\![\ \tau \vDash \partial\!\!\!/\ A; \tau \vDash \partial\!\!\!/\ B]\!] \Longrightarrow R;$
$[\![\ \tau \vDash \partial\!\!\!/\ A; \tau \vDash B\ ]\!] \Longrightarrow R;$
$[\![\ \tau \vDash A; \tau \vDash \partial\!\!\!/\ B\ ]\!] \Longrightarrow R\ ]\!]$

```
  ⟹ R
  by (auto elim!: and_TRUE_E and_FALSE_E simp add: isDefined_andD0 OclOr_def)



  lemmas isDefined_TRUE_E = non_tertium_datur_if_isDefined

  lemma isDefined_TRUE_I:
    ⟦ τ ⊨ A ∨ (τ ⊨ ¬ A) ⟧ ⟹ τ ⊨ ∂ A
    apply (erule disjE)
    apply (drule isDefined_if_valid)
    apply (drule_tac [2] isDefined_if_invalid)
    apply auto
    done

lemma isDefined_TRUE_I1:
⟦ (τ ⊨ A) ∨ (τ ⊨ (∅(A))) ⟹ τ ⊨ A ⟧ ⟹ τ ⊨ (∂ A)
  apply (rule isDefined_TRUE_I)
  apply (auto simp del: isDefined_if_valid isDefined_if_invalid)
  apply (simp only: not_valid not_not isDefined_notD0)
  done

  lemma not_not_E:
    ⟦ τ ⊨ ¬(¬ A); τ ⊨ A ⟹ R⟧ ⟹ R
    by auto

  lemma not_not_D:
    τ ⊨ ¬(¬ A) ⟹ τ ⊨ A
    by auto

  lemma not_not_I:
    ⟦ τ ⊨ A ⟧ ⟹ τ ⊨ ¬(¬ A)
    by auto

  lemma isDefined_not_TRUE_E:
    ⟦ τ ⊨ ∂(¬ A)⟧ ⟹ τ ⊨ ∂ A
    by auto

  lemma isDefined_not_TRUE_I:
    ⟦ τ ⊨ ∂ A ⟧ ⟹ τ ⊨ ∂(¬ A)
    by auto

  lemma isDefined_not_FALSE_E:
    ⟦ τ ⊨ ∅(¬ A)⟧ ⟹ τ ⊨ ∅(A)
    by auto

  lemma isDefined_not_FALSE_I:
    ⟦ τ ⊨ ∅ A ⟧ ⟹ τ ⊨ ∅(¬ A)
    by auto
```

Drived rules for the or

lemma *or_TRUE_I_core*:
$\llbracket\ (\tau \vDash A) \vee (\tau \vDash B)\ \rrbracket \Longrightarrow\ \tau \vDash A \vee B$
apply (*unfold OclOr_def*)
apply (*rule and_FALSE_I_core*)
apply *auto*
done

lemmas *or_TRUE_I* = *disjCI* [*THEN or_TRUE_I_core*, *standard*]

lemma *or_FALSE_I*:
$\llbracket\ \tau \vDash \neg\ A;\ \tau \vDash \neg\ B\ \rrbracket \Longrightarrow\ \tau \vDash \neg(A \vee B)$

apply (*unfold OclOr_def*)
apply (*rule not_not_I*)
apply (*rule and_TRUE_I*)
apply *auto*
done

lemma *or_TRUE_E*:
$\llbracket\ \ \tau \vDash A \vee B;$
$\quad \tau \vDash A \Longrightarrow R;$
$\quad \tau \vDash B \Longrightarrow R\ \rrbracket$
$\Longrightarrow\ R$
apply (*unfold OclOr_def*)
apply (*auto elim*!: *and_TRUE_E and_FALSE_E*
simp add: *isDefined_andD0 OclOr_def*)
done

lemma *or_FALSE_E*:
$\llbracket\ \ \tau \vDash \neg(A \vee B);$
$\quad \llbracket\ \tau \vDash \neg\ A;\ \tau \vDash \neg\ B\ \rrbracket \Longrightarrow R\ \rrbracket$
$\Longrightarrow\ R$
apply (*unfold OclOr_def*)
apply (*auto elim*!: *and_TRUE_E and_FALSE_E*
simp add: *isDefined_andD0 OclOr_def*)
done

lemma *isDefined_or_TRUE_I*:
$(\llbracket\ (\tau \vDash \not\partial\ B) \vee (\tau \vDash \neg\ B);$
$\quad (\tau \vDash \not\partial\ A) \vee (\tau \vDash \neg\ A)\ \rrbracket$
$\quad \Longrightarrow (\tau \vDash \partial\ A) \wedge (\tau \vDash \partial\ B)\ )$
$\Longrightarrow \tau \vDash \partial\ (A \vee B)$
apply (*unfold OclOr_def*)
apply (*rule isDefined_not_TRUE_I*)
apply (*rule isDefined_and_TRUE_I*)

```
  apply auto
  done

lemma isDefined_or_FALSE_I1:
⟦ τ ⊨ ∂ B ⟹ τ ⊨ ∅ A;
  τ ⊨ ∂ A ⟹ τ ⊨ ¬ A;
  τ ⊨ ∂ B ⟹ τ ⊨ ¬ B ⟧
 ⟹ (τ ⊨ ∅ (A ∨ B))
  apply (unfold OclOr_def)
  apply (rule isDefined_not_FALSE_I)
  apply (rule isDefined_and_FALSE_I4)
  apply auto
  done

lemma isDefined_or_FALSE_I:
⟦ (τ ⊨ A) ∨ ~(τ ⊨ ∅ B) ⟹ (τ ⊨ ∅ A);
  (τ ⊨ A) ∨ ~(τ ⊨ ∅ B) ⟹ ~(τ ⊨  B) ⟧
 ⟹ (τ ⊨ ∅ (A ∨ B))

  apply (unfold OclOr_def)
  apply (rule isDefined_not_FALSE_I)
  apply (rule isDefined_and_FALSE_I)

  apply (rule and_FALSE_I)
  apply (rule isDefined_not_FALSE_I)
  apply (rule_tac [2] and_FALSE_I)
  apply (rule_tac [2] isDefined_not_FALSE_I)
  apply (rule_tac [3] and_FALSE_I)
  apply (rule_tac [3] isDefined_not_FALSE_I)

  apply auto
  apply (simp_all only: not_valid)
  apply (auto simp: isDefined_if_valid)
  done

lemma isDefined_or_FALSE_E:
  ⟦ τ ⊨ ∅(A ∨ B);
    ⟦ τ ⊨ ∅ A; τ ⊨ ∅ B⟧ ⟹ R;
    ⟦ τ ⊨ ∅ A; τ ⊨ ¬ B ⟧     ⟹ R;
    ⟦ τ ⊨ ¬ A; τ ⊨ ∅ B ⟧     ⟹ R   ⟧
  ⟹  R
  apply (unfold OclOr_def)
  apply (drule isDefined_not_FALSE_E)
  apply (auto elim!: and_TRUE_E and_FALSE_E isDefined_and_FALSE_E
             simp add: isDefined_andD0 OclOr_def)
  done

lemma isDefined_or_TRUE_E:
  ⟦ τ ⊨ ∂(A ∨ B);
```

$[\![\ \tau \models \partial\ A;\ \tau \models \partial\ B]\!] \Longrightarrow R;$
$\tau \models B \quad \Longrightarrow R;$
$\tau \models A \quad \Longrightarrow R \quad]\!]$
$\Longrightarrow\ R$
apply (*unfold OclOr_def*)
apply (*drule isDefined_not_TRUE_E*)
apply (*erule isDefined_and_TRUE_E*)
apply *auto*
done

lemma *strongEq_distrib_if1*:
$\models \partial\ X \Longrightarrow$
$(A \doteq (\textit{if}\ X\ \textit{then}\ (Y{::}'a{=>}('b{::}bot))\ \textit{else}\ Z\ \textit{endif})) =$
$(\textit{if}\ X\ \textit{then}\ A \doteq Y\ \textit{else}\ A \doteq Z\ \textit{endif})$
by(*rule if_distrib2, auto simp: OclValid_def OclLocalValid_def*)

lemma *strongEq_distrib_if2*:
$\models \partial\ X \Longrightarrow$
$((\textit{if}\ X\ \textit{then}\ (Y{::}'a{=>}('b{::}bot))\ \textit{else}\ Z\ \textit{endif}) \doteq\ A) =$
$(\textit{if}\ X\ \textit{then}\ Y \doteq A\ \textit{else}\ Z \doteq A\ \textit{endif})$
by(*rule if_distrib2, auto simp: OclValid_def OclLocalValid_def*)

lemma *cp_local_valid* [*simp,intro!*]:
$cp\ P \Longrightarrow cp\ (\lambda\ X\ \tau.\ (\tau \models P\ X))$
by(*auto simp: OclValid_def OclLocalValid_def*)

lemma *and_congr*:
$[\![\ (\tau \models A) \lor (\tau \models \not\partial\ A) \Longrightarrow (\tau \models B) = (\tau \models B')\ ]\!]$
$\Longrightarrow (\tau \models A \land B) = (\tau \models A \land B')$
apply (*cut_tac* $\tau = \tau$ **and** $A = A$ **in** *non_quatrium_datur*)
apply (*rule_tac* $X = A$ **in** *cp_distinct_core_P*)
apply (*auto simp: OclLocalValid_def[symmetric]*)
done

lemma *and_congr_a*:
$[\![\ \tau \models \partial\ A;\ \tau \models A \Longrightarrow (\tau \models B) = (\tau \models B')\ ]\!]$
$\Longrightarrow\ (\tau\ \models A \land B) = (\tau \models A \land B')$
apply (*rule and_congr*)
apply *auto*
done

lemma *and_congr_b*:

$\llbracket\ \tau \vDash \neg\ A;\ \tau \vDash \not\partial\ A \Longrightarrow (\tau \vDash B) = (\tau \vDash B')\rrbracket$
$\Longrightarrow\ (\ \tau\ \ \vDash A \wedge B) = (\tau \vDash A \wedge B')$
apply (*rule and_ congr*)
apply *auto*
done

lemma *not_ congr*:
$\llbracket\ A\ \tau = A'\ \tau\rrbracket\ \Longrightarrow\ (\tau \vDash \neg\ A) = (\tau \vDash \neg\ A')$
apply (*rule sym*)
apply (*erule_ tac* $P = \lambda\ A\ \tau.\ (\tau \vDash \neg\ A)$ **in** *ocl_ cp_ subst*)
apply *auto*
done

The relevance for automated deduction becomes apparent with the next rewrite rule, which boils down reductions in local congruences to check if the definedness is equivalent. This can be decomposed into two meta-implications; the advantage of this is presentation comes from the fact that such derivations are particularly easy and can be supported automatically by a forward-closure tactic for literals contained in valid definedness formulas. It has to be checked if the recomputation of these literals can be avoided.

lemma *not_ congr0*:
$\llbracket\ (\tau \vDash \partial\ A) \longrightarrow (\tau \vDash \partial\ A');$
$(\tau \vDash \partial\ A') \longrightarrow (\tau \vDash \partial\ A);$
$(\tau \vDash A) = (\tau \vDash A')\rrbracket$
$\Longrightarrow\ (\tau \vDash \neg\ A) = (\tau \vDash \neg\ A')$
apply (*rule not_ congr*)
apply (*subst equiv_ D_ local*) **back**
apply *auto*
done

lemma *or_ congr*:
$\llbracket\ A\ \tau = A'\ \tau;\ B\ \tau = B'\ \tau\ \rrbracket$
$\Longrightarrow\ (\tau \vDash A \vee B) = (\tau \vDash A' \vee B')$
apply (*rule_ tac* $P = \lambda\ A\ \tau.\ (\tau \vDash A \vee B)$ **in** *ocl_ cp_ subst*)
apply (*rule sym*, *assumption*)
apply (*rule_ tac* $P = \lambda\ B\ \tau.\ (\tau \vDash A' \vee B)$ **in** *ocl_ cp_ subst*)
apply (*rule sym*, *assumption*)
apply *auto*
done

lemma *or_ congr0*:
$\llbracket\ (\tau \vDash \partial\ A)\ \ \longrightarrow (\tau \vDash \partial\ A');$
$(\tau \vDash \partial\ A') \longrightarrow (\tau \vDash \partial\ A);$
$(\tau \vDash \partial\ B)\ \ \longrightarrow (\tau \vDash \partial\ B');$
$(\tau \vDash \partial\ B') \longrightarrow (\tau \vDash \partial\ B);$
$A\ \tau = A'\ \tau\ ;\ B\ \tau = B'\ \tau\ \rrbracket$
$\Longrightarrow\ (\tau \vDash A \vee B) = (\tau \vDash A' \vee B')$
apply (*rule or_ congr*)

```
    apply (subst equiv_D_local) back
    prefer 2
    apply (subst equiv_D_local) back
    apply (auto simp: OclLocalValid_def)
    apply (auto simp: OclLocalValid_def[symmetric])
    done

lemma if_congr:
    ⟦ τ ⊨ C ⟹ (τ ⊨ A) = (τ ⊨ A′);
      τ ⊨ ¬ C ⟹ (τ ⊨ B) = (τ ⊨ B′) ⟧
    ⟹ ((τ ⊨ if C then A else B endif) =
        (τ ⊨ if C then A′else B′endif))
    apply (cut_tac τ = τ and A = C in non_quatrium_datur)
    apply (rule_tac X = C in cp_distinct_core_P)
    apply (auto simp: is_FALSE_charn_local OclLocalValid_def[symmetric])
    done

lemma implies_undef_congr[simp]:
    (τ ⊨ ⊥ ⟶ B) = (τ ⊨ B)
    by (simp add:OclImplies_def)

lemma implies_congr:
    ⟦ (τ ⊨ A) ∨ (τ ⊨ ∅ A) ⟹ (τ ⊨ B) = (τ ⊨ B′) ⟧
    ⟹ ((τ ⊨ A ²⟶ B) = (τ ⊨ A ²⟶ B′))
    apply (cut_tac τ = τ and A = A in non_quatrium_datur)
    apply (rule_tac X = A in cp_distinct_core_P)
    apply (auto simp: isUndefined_charn_local OclLocalValid_def[symmetric])
    done

lemmas LEC = and_congr not_congr or_congr if_congr
              not_and_undef_congr1
              not_and_undef_congr2
              or_undef_congr1
              or_undef_congr2
              and_undef_congr3
              and_undef_congr4
              implies_undef_congr

end
```

B.5.3. Encoding of OCL Operations

```
theory OCL_Operation
imports
  $HOLOCL_HOME/src/OCL_Calculi
begin
```

Standard and Strict Method Invocation, and Recursion

The core primitives *OCL_choose* and *OCL_invoke*, its strict variant *OCL_invokeS* are used to represent method.

constdefs *OCL_choose* ::
$(('tau, 'b)\ VAL \Rightarrow ('tau)\ Boolean) \Rightarrow ('tau, 'b)\ VAL$
$OCL\_choose\ F \equiv (\lambda\ \tau.\ SOME\ x.\ F\ (\lambda\ \tau.\ x)\ \tau = \top\ \tau)$

lemma $cp\ F \Longrightarrow cp(OCL\_choose\ F)$
apply(*simp add: cp_def OCL_choose_def*)
apply *safe*
apply (*rule exI*, *rule allI*, *rule allI*)
apply (*simp*)
oops

consts $OCL\_invoke :: ('e \Rightarrow 's)$
$\Rightarrow ('s\ set \rightharpoonup ('a, 'b)\ VAL \Rightarrow 'c)$
$\Rightarrow ('a, 'b)\ VAL \Rightarrow 'c$
consts $OCL\_invokeS :: ('b \Rightarrow 's)$
$\Rightarrow ('s\ set \rightharpoonup ('a, 'b)\ VAL \Rightarrow 'c)$
$\Rightarrow ('a, 'b)\ VAL \Rightarrow 'c$

Instance for null-ary invocation with return value:

$$\begin{aligned}&\mathrm{OCL\_invoke}\,(C ::' a \Rightarrow' d)\\ &\qquad (tab ::' dset \rightharpoonup ('s \Rightarrow' a) \Rightarrow ('s \Rightarrow' b) \Rightarrow' s \Rightarrow' c)\\ &\qquad (self ::' s \Rightarrow' a)(result ::' s \Rightarrow' b)\end{aligned}$$

defs *OCL_invoke0_def* :
$OCL\_invoke\ C\ tab\ self\ result \equiv$
$\lambda\ St.\ (case\ tab\ (LEAST\ X.\ X \in dom\ tab \wedge C(self\ St) \in X)\ of$
$None \Rightarrow\ arbitrary$
$\mid Some\ f \Rightarrow f\ (self::'s \Rightarrow\ 'a)$
$(result::'s \Rightarrow 'b)\ St\ )$

defs
OCL_invoke0S_def:
$OCL\_invokeS\ C\ tab\ self\ result \equiv$
$(if\ \partial(self)\ \ then\ OCL\_invoke\ C\ tab\ self\ result\ else\ \bot\ endif)$

lemma *OCL_invoke0S_undef0*[*simp*] :
$OCL\_invokeS\ C\ (tab::'c\ set \rightharpoonup ('a \Rightarrow 'd::bot) \Rightarrow ('a \Rightarrow 'e::bot) \Rightarrow 'a \Rightarrow 'b::bot)\ \bot\ X$
$= \ \bot$
by(*simp add: OCL_invoke0S_def*)

lemma *OCL_invoke0S_is_def*:
$\models \partial(OCL\_invokeS\ C$
$(tab::'c\ set \rightharpoonup ('a \Rightarrow 'd::bot) \Rightarrow ('a \Rightarrow 'e::bot) \Rightarrow 'a \Rightarrow 'b::bot)$

$$self\ result) \Longrightarrow$$
$$\models \partial(self)$$
by(*simp add: OCL_invoke0S_def isDefined_ifD0 globalValidAnd2conj*)

Instance for binary invocation:

$$\text{OCL\_invoke}\ (C ::' a \Rightarrow' e)$$
$$(tab ::' eset \rightharpoonup ('s \Rightarrow' a) \Rightarrow ('s \Rightarrow' b) \Rightarrow ('s \Rightarrow' c) \Rightarrow' s \Rightarrow' d)(self ::' s \Rightarrow' a)$$
$$(a ::' s \Rightarrow' b)(b ::' s \Rightarrow' c)$$

defs *OCL_invoke1_def* :
OCL_invoke C tab self a result
$\equiv \lambda\ \tau\ .\ (case\ tab\ (LEAST\ X.\ X \in dom\ tab \wedge C(self\ \tau) \in X)\ of$
$None \Rightarrow\ \ arbitrary$
$\mid Some\ f \Rightarrow f\ (self{::}'tau \Rightarrow\ 'a)$
$(a{::}'tau \Rightarrow\ 'b)$
$(result{::}'tau \Rightarrow\ 'c)\ \tau)$

defs *OCL_invoke1S_def*:
OCL_invokeS C tab self a result
$\equiv (if\ \partial(self)\ and\ \partial(a)$
$then\ OCL\_invoke\ C\ tab\ self\ (a{::}'tau \Rightarrow\ 'b{::}bot)(result{::}'tau \Rightarrow\ 'c)$
$else\ \perp\ endif)$

lemma *OCL_invoke1S_undef0*[*simp*] :
OCL_invokeS C
$(tab{::}'e\ set{\rightharpoonup}('s{\Rightarrow}'a{::}bot){\Rightarrow}('s{\Rightarrow}'b{::}bot){\Rightarrow}('s{\Rightarrow}'c{::}bot){\Rightarrow}'s{\Rightarrow}'d{::}bot)$
$\perp\ X\ Y$
$= \perp$
by(*simp add: OCL_invoke1S_def*)

lemma *OCL_invoke1S_undef1*[*simp*] :
OCL_invokeS C
$(tab{::}'e\ set{\rightharpoonup}('s{\Rightarrow}'a{::}bot){\Rightarrow}('s{\Rightarrow}'b{::}bot){\Rightarrow}('s{\Rightarrow}'c{::}bot){\Rightarrow}'s{\Rightarrow}'d{::}bot)$
$self\ \perp\ Y$
$=\ \perp$
by(*simp add: OCL_invoke1S_def*)

lemma *OCL_invoke2S_is_def'*:
$\models \partial(OCL\_invokeS\ C$
$(tab{::}'e\ set{\rightharpoonup}('s{\Rightarrow}'a{::}bot){\Rightarrow}('s{\Rightarrow}'b{::}bot){\Rightarrow}('s{\Rightarrow}'c{::}bot){\Rightarrow}'s{\Rightarrow}'d{::}bot)$
$self\ X\ Y) \Longrightarrow$
$\models \partial(self)\ and\ \partial(X)$
by(*simp add: OCL_invoke1S_def isDefined_ifD0 globalValidAnd2conj*)

Instance for Binary Invocation:

defs *OCL_invoke2_def* :

$OCL\_invoke\ C\ tab\ self\ a\ b\ result$
$\equiv \lambda\ St.\ (case\ tab\ (LEAST\ X.\ X \in dom\ tab \wedge C(self\ St) \in X)\ of$
$None \Rightarrow\ arbitrary$
$|\ Some\ f \Rightarrow f\ (self::'s{\Rightarrow}'a)(a::'s{\Rightarrow}'b)(b::'s{\Rightarrow}'c)(result::'s{\Rightarrow}'d)\ St\ )$

defs
$OCL\_invoke2S\_def$:
$OCL\_invokeS\ C\ tab\ self\ a\ b\ result$
$\equiv (if\ \partial(self)\ and\ \partial(a)\ and\ \partial(b)$
$then\ OCL\_invoke\ C\ tab\ (self::'s{\Rightarrow}'a::bot)$
$(a::'s{\Rightarrow}'b::bot)(b::'s{\Rightarrow}'c::bot)$
$(result::'s{\Rightarrow}'d)$
$else\ \bot$
$endif)$

lemma $OCL\_invoke2S\_undef0[simp]$:
$OCL\_invokeS\ C$
$(tab::'e\ set \rightharpoonup ('s{\Rightarrow}'a::bot){\Rightarrow}('s{\Rightarrow}'b::bot)$
${\Rightarrow}('s{\Rightarrow}'c::bot){\Rightarrow}('s{\Rightarrow}'d::bot){\Rightarrow}('s{\Rightarrow}'e::bot))$
$\bot\ X\ Y\ Z$
$= \bot$
by$(simp\ add{:}\ OCL\_invoke2S\_def)$

lemma $OCL\_invoke2S\_undef1[simp]$:
$OCL\_invokeS\ C$
$(tab::'e\ set \rightharpoonup ('s{\Rightarrow}'a::bot){\Rightarrow}('s{\Rightarrow}'b::bot)$
${\Rightarrow}('s{\Rightarrow}'c::bot){\Rightarrow}('s{\Rightarrow}'d::bot){\Rightarrow}('s{\Rightarrow}'e::bot))$
$X\ \bot\ Y\ Z$
$= \bot$
by$(simp\ add{:}\ OCL\_invoke2S\_def)$

lemma $OCL\_invoke2S\_undef2[simp]$:
$OCL\_invokeS\ C$
$(tab::'e\ set \rightharpoonup ('s{\Rightarrow}'a::bot){\Rightarrow}('s{\Rightarrow}'b::bot)$
${\Rightarrow}('s{\Rightarrow}'c::bot){\Rightarrow}('s{\Rightarrow}'d::bot){\Rightarrow}('s{\Rightarrow}'e::bot))$
$X\ Y\ \bot\ Z$
$= \bot$
by$(simp\ add{:}\ OCL\_invoke2S\_def)$

lemma $OCL\_invoke2S\_is\_def$:
$\models \partial(OCL\_invokeS\ C$
$(tab::'e\ set \rightharpoonup ('s{\Rightarrow}'a::bot){\Rightarrow}('s{\Rightarrow}'b::bot)$
${\Rightarrow}('s{\Rightarrow}'c::bot){\Rightarrow}('s{\Rightarrow}'d::bot){\Rightarrow}('s{\Rightarrow}'e::bot))$
$self\ X\ Y\ result) \Longrightarrow$
$\models \partial(self)\ and\ \partial(X)\ and\ \partial(Y)$
by$(simp\ add{:}\ OCL\_invoke2S\_def\ isDefined\_ifD0\ globalValidAnd2conj)$
end

B.5.4. The Object Constraint Language: OCL

theory *UML_OCL*
imports
$HOLOCL_HOME/src/OCL_State
$HOLOCL_HOME/src/OCL_Operation
begin

The Theory-Morpher

In the theory file, this section contains an SML program implementing a theory morphism from HOL to HOL-OCL. It filters all combinator-based definitions in HOL-OCL, contructs a one-to-one constant symbol based signature morphism between the HOL and the HOL-OCL theory (modulo argument permutation), uses this signature morphism to translate HOL theorems to HOL-OCL proof goals and attempts to prove them with a generic case split tactic. For conceptual details, see [13].

end

B.6. Requirements

In this section, we present the requirements from the normative part of the OCL standard [41, Chapter 11] and prove (or disprove) that our embedding is faithfull to these requirements.

At time of writing, this is work in progress and not all requirements are analyzed. This section will be improved in one of the next releases of HOL-OCL.

B.6.1. Requirements for OCL Primitive Types

theory *OCL_Basic_requirements*
imports
$HOLOCL_HOME/src/OCL_Library
begin

This theory contains the requirements for the primitive types defined in the OCL standard library (Integer, Real, String, and Boolean) [41, Chapter 11.5].

Real

+ (r : Real) : Real (11.5.1-a)

The value of the addition of self and r.

- (r : Real) : Real (11.5.1-b)

The value of the subtraction of r from self.

*** (r : Real) : Real** (11.5.1-c)

The value of the multiplication of self and r.

- : Real (11.5.1-d)

The negative value of self.

/ (r : Real) : Real (11.5.1-e)

The value of self divided by r.

---

abs() : Real (11.5.1-f)

The absolute value of self.

```
post: if self < 0 then result = - self
                  else result = self
                  endif
```

lemma req_11_5_1−f:
(self −>abs()) = (if (self < 0) then OclNegative self else self endif)
oops

---

floor() : Integer (11.5.1-g)

The largest integer which is less than or equal to self.

```
post: (result <= self) and (result + 1 > self)
```

lemma req_11_5_1−g:
$\models$ ((self −>floor())−>toReal() $\leq$ self) $\wedge$ ((self −>floor() + 1) −>toReal() $>$ self)
oops

---

round() : Integer (11.5.1-h)

The integer which is closest to self. When there are two such integers, the largest one.

```
post: ((self - result).abs() < 0.5)
      or ((self - result).abs() = 0.5 and (result > self))
```

lemma req_11_5_1−h: true
oops

---

max(r : Real) : Real (11.5.1-i)

The maximum of self and r.

```
post: if self >= r then result = self
                  else result = r
                  endif
```

---

min(r : Real) : Real (11.5.1-j)

The minimum of self and r.

```
post: if self <= r then result = self
                  else result = r
                  endif
```

< (r : Real) : Boolean (11.5.1-k)

True if self is less than r.

> (r : Real) : Boolean (11.5.1-l)

True if self is greater than r.

```
post: result = not (self <= r)
```

<= (r : Real) : Boolean (11.5.1-m)

True if self is less than or equal to r.

```
post: result = ((self = r) or (self < r))
```

>= (r : Real) : Boolean (11.5.1-n)

True if self is greater than or equal to r.

```
post: result = ((self = r) or (self > r))
```

Integer

- : Integer (11.5.2-a)

The negative value of self.

+ (i : Integer) : Integer (11.5.2-b)

The value of the addition of self and i.

- (i : Integer) : Integer (11.5.2-c)

The value of the subtraction of i from self.

*** (i : Integer) : Integer** (11.5.2-d)

The value of the multiplication of self and i.

/ (i : Integer) : Real (11.5.2-e)

The value of self divided by i.

---

abs() : Integer (11.5.2-f)

The absolute value of self.

```
post: if self < 0 then result = - self
                  else result = self
                  endif
```

---

div(i : Integer) : Integer (11.5.2-g)

The number of times that i fits completely within self.

```
pre : i <> 0
post: if self / i >= 0 then result = (self / i).floor()
                       else result = -((-self/i).floor())
                       endif
```

---

mod(i : Integer) : Integer (11.5.2-h)

The result is self modulo i.

```
post: result = self - (self.div(i) * i)
```

---

max(i : Integer) : Integer (11.5.2-i)

The maximum of self an i.

```
post: if self >= i then result = self
                   else result = i
                   endif
```

---

min(i : Integer) : Integer (11.5.2-j)

The minimum of self an i.

```
post: if self <= i then result = self
                    else result = i
                    endif
```

String

size() : Integer (11.5.3-a)

The number of characters in self.

concat(s : String) : String (11.5.3-b)

The concatenation of self and s.

```
post: result.size() = self.size() + string.size()
post: result.substring(1, self.size() ) = self
post: result.substring(self.size() + 1, result.size() ) = s
```

substring(lower : Integer, upper : Integer) : String (11.5.3-c)

The sub-string of self starting at character number lower, up to and including character number upper. Character numbers run from 1 to self.size().

```
pre: 1 <= lower
pre: lower <= upper
pre: upper <= self.size()
```

toInteger() : Integer (11.5.3-d)

Converts self to an Integer value.

toReal() : Real (11.5.3-e)

Converts self to a Real value.

Boolean

or (b : Boolean) : Boolean (11.5.4-a)

True if either self or b is true.

---

xor (b : Boolean) : Boolean (11.5.4-b)

True if either self or b is true, but not both.

```
post: (self or b) and not (self = b)
```

---

and (b : Boolean) : Boolean (11.5.4-c)

True if both b1 and b are true.

---

not : Boolean (11.5.4-d)

True if self is false.

```
post: if self then result = false
              else result = true
              endif
```

---

implies (b : Boolean) : Boolean (11.5.4-e)

True if self is false, or if self is true and b is true.

```
post: (not self) or (self and b)
```

---

end

B.6.2. OCL Collection requirements

theory *OCL_Collection_requirements*
imports
 $*HOLOCL_HOME/src/OCL_Library*
begin

This theory contains the requirements for the collection types defined in the OCL standard library (Set, OrderedSet, Bag, and Sequence) [41, Chapter 11.7].

Collection

---

size() : Integer (11.7.1-a)

The number of elements in the collection *self*.

```
post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)
```

Set

OrderedSet

Bag

lemma req_bag_11_7_1−a:
$\|(\text{self}::('a,'b::bot)\text{Bag})\| = \text{OclIterate self } (\lambda\ (\text{elem}::('a,'b::bot)\text{VAL})\ (\text{acc}::'a\ \text{Integer}).\ \text{acc} + 1)\ 0$
by(simp add: OCL_Bag.size_by_iterate)

Sequence

lemma req_seq_11_7_1−a:
$\|(\text{self}::('a,'b::bot)\text{Sequence})\| = \text{OclIterate self } (\lambda\ (\text{elem}::('a,'b::bot)\text{VAL})\ (\text{acc}::'a\ \text{Integer}).\ \text{acc} + 1)\ 0$
by(simp add: OCL_Sequence.size_by_iterate)

---

includes(object : T) : Boolean (11.7.1-b)

True if object is an element of self, false otherwise.

```
post: result = (self->count(object) > 0)
```

Set

OrderedSet

Bag

lemma req_bag_11_7_1−b:
$((\text{obj}::('a,'b)\text{VAL}) \in (\text{self}::('a,'b::bot)\text{Bag})) = ((\text{self} \rightarrow\!\texttt{count}\ \text{obj}) > 0)$
by(simp only: OCL_Bag.includes_by_count_UC)

Sequence

lemma req_seq_11_7_1−b:
$((\text{obj}::('a,'b)\text{VAL}) \in (\text{self}::('a,'b::bot)\text{Sequence})) = ((\text{self} \rightarrow\!\texttt{count}\ \text{obj}) > 0)$
by(simp only: OCL_Sequence.includes_by_count_UC)

---

excludes(object : T) : Boolean (11.7.1-c)

True if object is not an element of self, false otherwise.

```
post: result = (self->count(object) = 0)
```

Set

OrderedSet

Bag

lemma req_bag_11_7_1−c:

((obj::('a,'b)VAL) $\notin$ (self::('a,'b::bot)Bag)) = ((self ->count obj) $\doteq$ 0)

oops

Sequence

lemma req_seq_11_7_1−c:

((obj::('a,'b)VAL) $\notin$ (self::('a,'b::bot)Sequence)) = ((self ->count obj) $\doteq$ 0)

by(simp)

count(object : T) : Integer (11.7.1-d)

The number of times that object occurs in the collection self.

```
post: result = self->iterate( elem; acc : Integer = 0 |
        if elem = object then acc + 1 else acc endif)
```

Set

OrderedSet

Bag

lemma req_bag_11_7_1−d:
$\llbracket\ \tau \vDash \partial(\text{obj}::('a,'b)\text{VAL})\ \rrbracket \Longrightarrow$
$((\text{self}::('a,'b::\text{bot})\text{Bag})\ \text{->count}\ \text{obj}\ )\ \tau =$
$(\text{self}\ \text{->iterate}(x;y{=}0 \mid (\text{if}\ (x \doteq \text{obj})\ \text{then}\ (y + 1)\text{else}\ y\text{endif})))\ \tau$
by(simp add: OCL_Bag.count_by_iterate OCL_Bag.OclIterate_def
OclIf_def localValid2sem ss_lifting′)

Sequence

lemma req_seq_11_7_1−d:
$\llbracket\ \tau \vDash \partial(\text{obj}::('a,'b)\text{VAL})\ \rrbracket \Longrightarrow$
$((\text{self}::('a,'b::\text{bot})\text{Sequence})\ \text{->count}\ \text{obj}\ )\ \tau =$
$(\text{self}\ \text{->iterate}(x;y{=}0 \mid (\text{if}\ (x \doteq \text{obj})\ \text{then}\ (y + 1)\text{else}\ y\text{endif})))\ \tau$
by(simp add: OCL_Sequence.count_by_iterate OCL_Sequence.OclIterate_def

OclIf_def localValid2sem ss_lifting′)

includesAll(c2 : Collection(T)) : Boolean (11.7.1-e)

Does self contain all the elements of c2 ?

```
post: result = c2->forAll(elem | self->includes(elem))
```

Set

If *self* $= \bot$ and ($obj = \emptyset$) then the left-hand side of the equation evaluates to $\bot$ whereas the right-hand side evaluates to $\top$.

Wrong definition of the correspondence of includesAll and iterate

lemma req_set_11_7_1−e:
(((self::($'\tau$,$'\alpha$::bot)Set) −>includesAll(obj))
= (obj −>forAll(elem | (self −>includes((elem::($'\tau$,$'\alpha$::bot)VAL)))))) = False
apply(simp add: OclIncludes_def OclIncludesAll_def OclForAll_def
lift0_def lift1_def lift2_def strictify_def DEF_def)
oops

OrderedSet

Bag

lemma req_bag_11_7_1−e:
$[\![$ $\tau \models \partial$(self::($'$a,$'$b::bot)Bag) $]\!] \Longrightarrow$
(OclIncludesAll self obj) τ = ($\forall$ elem $\in$ obj . ((elem::($'$a,$'$b)VAL) $\in$ self)) τ
oops

Sequence

lemma req_seq_11_7_1−e:
$[\![$ $\tau \models \partial$(self::($'$a,$'$b::bot)Sequence) $]\!] \Longrightarrow$
(OclIncludesAll self obj) τ = ($\forall$ elem $\in$ obj . ((elem::($'$a,$'$b)VAL) $\in$ self)) τ
apply(rule_tac X=self ***in*** Sequence_sem_cases)
apply(rule_tac X=obj ***in*** Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(auto simp: OCL_Sequence.OclForAll_def OCL_Sequence.OclIncludesAll_def OclStrongEq_def
OCL_Sequence.OclIncludes_def ss_lifting$'$ localValid2sem)
done

excludesAll(c2 : Collection(T)) : Boolean (11.7.1-f)

Does self contain none of the elements of c2 ?

```
post: result = c2->forAll(elem | self->excludes(elem))
```

Set

Wrong definition of the correspondence of excludesAll and iterate

If *self* $= \perp$ and *obj* $= \emptyset$ then the left-hand side of the equation evaluates to $\perp$ whereas the right-hand side evaluates to $\top$.

lemma req_set_11_7_1−f:
((self::(′τ,′α::bot)Set) −>excludesAll(obj))
= (obj −>forAll(elem | (self −>excludes((elem::(′τ,′α::bot)VAL)))))
apply(rule ext)
apply(simp add: OclExcludes_def OclExcludesAll_def OclForAll_def
lift0_def lift1_def lift2_def strictify_def DEF_def)
apply(auto)
oops

OrderedSet

Bag

lemma req_bag_11_7_1−f:
$[\![\ \tau \models \partial$(self::(′a,′b::bot)Bag) $]\!] \Longrightarrow$
$\tau \models$ (OclExcludesAll self obj) $\triangleq$ ($\forall$ elem $\in$ obj . ((elem::(′a,′b)VAL) $\not\in$ self))
oops

Sequence

lemma req_seq_11_7_1−f:
$[\![\ \tau \models \partial$(self::(′a,′b::bot)Sequence) $]\!] \Longrightarrow$
$\tau \models$ (OclExcludesAll self obj) $\triangleq$ ($\forall$ elem $\in$ obj . ((elem::(′a,′b)VAL) $\not\in$ self))
apply(rule_tac X=self ***in*** Sequence_sem_cases)
apply(rule_tac X=obj ***in*** Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(auto simp: OCL_Sequence.OclForAll_def OCL_Sequence.OclExcludesAll_def OclStrongEq_def
OCL_Sequence.OclIncludes_def ss_lifting′ localValid2sem OclNot_def)
done

isEmpty() : Boolean (11.7.1-g)

Is self the empty collection?

```
post: result = ( self->size() = 0 )
```

Set

Due to our extension to infinite sets, the usual connection between isEmpty and size holds only under a precondition

lemma REQ11_7_1_7:
$\models \partial(\text{self} \; {-}{>}\text{size}())$
$\Longrightarrow$ self −>isEmpty() = (self::('τ,'α::bot)Set) −>size() $\doteq$ 0
apply (rule ext)
apply (drule_tac τ = x *in* valid_elim)
apply (subgoal_tac finite (Lifting.drop (Rep_Set_0 (self x))))
prefer 2
apply (erule finite_Sets)
apply (simp add: OclIsEmpty_def OclSize_def OclStrictEq_def
OCL_Integer.Zero_ocl_int_def)
apply (simp only: lift0_def lift1_def lift2_def
strictify_def o_def not_def)
apply (simp_all (no_asm_use) add: UU_fun_def DEF_def strictify_def
split add: split_if up.split)
apply (simp only: lift0_def lift1_def lift2_def
strictify_def o_def not_def OclIsDefined_def DEF_def OclTrue_def)
apply auto
done

OrderedSet

Bag

lemma req_bag_11_7_1−g:
($\doteq \emptyset$ self) = $\|$(self::('a,'b::bot)Bag)$\|$ $\doteq$ 0
apply(rule Bag_sem_cases_ext, simp_all)
apply(simp_all add: OCL_Bag.OclSize_def OclMtBag_def OclStrictEq_def
Zero_ocl_int_def ss_lifting')
done

Sequence

lemma req_seq_11_7_1−g:
($\doteq \emptyset$ self) = $\|$(self::('a,'b::bot)Sequence)$\|$ $\doteq$ 0
by simp

notEmpty() : Boolean (11.7.1-h)

Is self not the empty collection?

```
post: result = ( self->size() <> 0 )
```

Set

lemma req_set_11_7_1−h:
OclNotEmpty(X::('τ, 'α::bot) Set) = ¬ (OclIsEmpty(X))
apply (rule ext)
apply (simp add: OclNotEmpty_def OclIsEmpty_def OclNot_def)
apply (simp only: lift1_def lift2_def
strictify_def o_def not_def)
apply (simp_all (no_asm_use) add: UU_fun_def DEF_def strictify_def
split add: split_if up.split)
done

OrderedSet

Bag

lemma req_bag_11_7_1−h:
(≠ ∅ self) = (‖(self::('a,'b::bot)Bag)‖) ‘<>‘ 0
by(simp add: req_bag_11_7_1−g[symmetric])

Sequence

lemma req_seq_11_7_1−h:
(≠ ∅ self) = (‖(self::('a,'b::bot)Sequence)‖) ‘<>‘ 0
by(simp add: req_seq_11_7_1−g[symmetric])

sum() : T (11.7.1-i)

The addition of all elements in self. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T and be both associative: (a+b)+c = a+(b+c), and commutative: a+b = b+a. Integer and Real fulfill this condition.

```
post: result = self->iterate( elem; acc : T = 0 | acc + elem )
```

Set

OrderedSet

Bag

lemma req_bag_11_7_1−i:
(->sum() S) = ((S::('a,'b::{bot,plus,zero})Bag) ->iterate(x;y=0 | (x + y)))
by(simp add: OCL_Bag.OclSum_def)

Sequence

lemma req_seq_11_7_1−i:
(->sum() S) = ((S::('a,Integer_0)Sequence) ->iterate(x;y=0 | (x + y)))
by(simp add: OCL_Sequence.OclSum_def)

Note: That this requirement is not fulfilled completely as it currently holds only for integers. But this will change in the future.

product(c2: Collection(T2)) : Set(Tuple(first: T, second: T2)) (11.7.1-j)

The cartesian product operation of self and c2.

```
post: result =
self->iterate(e1; acc: Set(Tuple(first: T, second: T2)) = Set{} |
 c2->iterate(e2; acc2: Set(Tuple(first: T, second: T2)) = acc |
             acc2->including (Tuple{first = e1, second = e2}) ) )
```

Set

OrderedSet

Bag

lemma req_bag_11_7_1−j:
(OclProduct (self::('a,'b::bot)Bag) (c2::('a,'c::bot)Bag)) =
 (self ->iterate(e1; acc=OclMtSet |
  (c2 ->iterate(e2; acc2=acc |
   (acc2 ->including (OclTuple e1 e2))))))
oops

Sequence

lemma req_seq_11_7_1−j:
(OclProduct (self::('a,'b::bot)Sequence) (c2::('a,'c::bot)Sequence)) =
 (self ->iterate(e1; acc=OclMtSet |
  (c2 ->iterate(e2; acc2=acc |
   (acc2 ->including (OclTuple e1 e2))))))
oops

Set

OrderedSet

Bag

= (bag : Bag(T)) : Boolean (11.7.7-b)

True if self and bag contain the same elements, the same number of times.

```
post:
result=(self->forAll(elem | self->count(elem) = bag->count(elem))
       and
       bag->forAll(elem | bag->count(elem) = self->count(elem)))
```

---

union(bag : Bag(T)) : Bag(T) (11.7.7-b)

The union of self and bag.

```
post: result->forAll( elem |
      result->count(elem) = self->count(elem) + bag->count(elem))
post: self ->forAll( elem |
      result->count(elem) = self->count(elem) + bag->count(elem))
post: bag ->forAll( elem |
      result->count(elem) = self->count(elem) + bag->count(elem))
```

union(set : Set(T)) : Bag(T) (11.7.7-b)

The union of self and set.

```
post: result->forAll(elem |
      result->count(elem) = self->count(elem) + set->count(elem))
post: self ->forAll(elem |
      result->count(elem) = self->count(elem) + set->count(elem))
post: set ->forAll(elem |
      result->count(elem) = self->count(elem) + set->count(elem))
```

intersection(bag : Bag(T)) : Bag(T) (11.7.7-b)

The intersection of self and bag.

```
post: result->forAll(elem |
  result->count(elem)=self->count(elem).min(bag->count(elem)))
post: self->forAll(elem |
  result->count(elem)=self->count(elem).min(bag->count(elem)))
post: bag->forAll(elem |
  result->count(elem)=self->count(elem).min(bag->count(elem)))
```

intersection(set : Set(T)) : Set(T) (11.7.7-b)

The intersection of self and set.

```
post: result->forAll(elem|
  result->count(elem) = self->count(elem).min(set->count(elem)))
post: self ->forAll(elem|
  result->count(elem) = self->count(elem).min(set->count(elem)))
post: set ->forAll(elem|
  result->count(elem) = self->count(elem).min(set->count(elem)))
```

including(object : T) : Bag(T) (11.7.7-b)

The bag containing all elements of self plus object.

```
post: result->forAll(elem |
     if elem = object then
        result->count(elem) = self->count(elem) + 1
     else
        result->count(elem) = self->count(elem)
     endif)
post: self->forAll(elem |
     if elem = object then
        result->count(elem) = self->count(elem) + 1
     else
        result->count(elem) = self->count(elem)
     endif)
```

excluding(object : T) : Bag(T) (11.7.7-b)

The bag containing all elements of self apart from all occurrences of object.

```
post: result->forAll(elem |
     if elem = object then
        result->count(elem) = 0
     else
        result->count(elem) = self->count(elem)
     endif)
post: self->forAll(elem |
     if elem = object then
        result->count(elem) = 0
     else
        result->count(elem) = self->count(elem)
     endif)
```

count(object : T) : Integer (11.7.7-b)

The number of occurrences of object in self.

flatten() : Bag(T2) (11.7.7-b)

If the element type is not a collection type this result in the same bag. If the element type is a collection type, the r is the bag containing all the elements of all the elements of self.

```
post:
  result = if self.type.elementType.oclIsKindOf(CollectionType)
      then
        self->iterate(c; acc : Bag() = Bag{} |
            acc->union(c->asBag() ) )
      else
        self
      endif
```

asBag() : Bag(T) (11.7.7-b)

A Bag identical to self. This operation exists for convenience reasons.

```
post: result = self
```

asSequence() : Sequence(T) (11.7.7-b)

A Sequence that contains all the elements from self, in undefined order.

```
post:
  result->forAll(elem | self->count(elem)=result->count(elem))
post:
  self ->forAll(elem | self->count(elem)=result->count(elem))
```

asSet() : Set(T) (11.7.7-b)

The Set containing all the elements from self, with duplicates removed.

```
post: result->forAll(elem | self ->includes(elem))
post: self ->forAll(elem | result->includes(elem))
```

asOrderedSet() : OrderedSet(T) (11.7.7-b)

An OrderedSet that contains all the elements from self, in undefined order, with duplicates removed.

```
post: result->forAll(elem | self ->includes(elem))
post: self ->forAll(elem | result->includes(elem))
post: self ->forAll(elem | result->count(elem) = 1)
```

Sequence

---

count(object : T) : Integer (11.7.5-a)

The number of occurrences of object in self.

= (s : Sequence(T)) : Boolean (11.7.5-b)

True if self contains the same elements as s in the same order.

```
post:
  result=Sequence{1..self->size()}->forAll(index : Integer |
                  self->at(index) = s->at(index))
                  and
                  self->size() = s->size()
```

lemma req_11_7_5−b:
((S1::('a,'b::bot)Sequence) ≐ S2) =
(∀ i ∈ ((OclCollectionRange 1 ‖S1‖)::('a,Integer_0)Sequence) . ((OclAt S1 i) ≐ (OclAt S2 i))) ∧
(‖S1‖ ≐ ‖S2‖)
oops

---

union (s : Sequence(T)) : Sequence(T) (11.7.5-c)

The sequence consisting of all elements in self, followed by all elements in s.

```
post: result->size() = self->size() + s->size()
post: Sequence{1..self->size()}->forAll(index : Integer |
                          self->at(index) = result->at(index))
post: Sequence{1..s->size()}->forAll(index : Integer |
```

lemma req_11_7_5−c−1:
‖((self::('a,'b::bot)Sequence) ∪ s)‖ = (‖self‖ + ‖s‖)
by(simp)
lemma req_11_7_5−c−2:
⟦ τ ⊨ ∂(self::('a,'b::bot)Sequence) ⟧ ⟹
τ ⊨ ∀ index ∈ ((OclCollectionRange 1 ‖self‖)::('a,Integer_0)Sequence) .
((OclAt (self ∪ s) index) ≐ (OclAt self index))
apply(rule OCL_Sequence.ForAllI)
apply(simp add: OCL_Sequence.OCL_is_defopt_OclCollectionRange)
apply(simp add: OCL_Sequence.includes_of_collectionRange)
oops
lemma req_11_7_5−c−3:
⊨ ∀ index ∈ ((OclCollectionRange 1 ‖s‖)::('a,Integer_0)Sequence) .
((OclAt s index) ≐ (OclAt (self ∪ s) (index + ‖self‖)))
oops

---

flatten() : Sequence(T2) (11.7.5-d)

If the element type is not a collection type this result in the same self. If the element type is a collection type, the result is the seuqnce containing all the elements of all the elements of self. The order of the elements is partial.

```
post:
  result = if self.type.elementType.oclIsKindOf(CollectionType)
        then
          self->iterate(c; acc : Sequence() = Sequence{} |
             acc->union(c->asSequence() ) )
        else
          self
        endif
```

The postcondition in the standard is not typeable because it must have these two types at the same time which is not possible:
`(('a,'b::bot Sequence_0)Sequence) \<Rightarrow> ('a,'b)Sequence`
`('a,'b::bot)Sequence \<Rightarrow> ('a,'b)Sequence`
therefore we split the requirement into two parts:

1. characterises the case of a flatten of a sequence of sequences
2. characterises the case of a flatten of a sequence of a not collection type. This follows by definition of flatten on sequences of non collection types

lemma req_11_7_5−d−1:
(⟦(self::('a,'b::bot Sequence_0)Sequence)⟧::('a,'b)Sequence) =
(self ->iterate(x;y=[] | (y ∪ x)))
by(simp add: OCL_Sequence.flatten_by_iterate)

append (object: T) : Sequence(T) (11.7.5-e)

The sequence of elements, consisting of all elements of self, followed by object.

```
post: result->size() = self->size() + 1
post: result->at(result->size() ) = object
post: Sequence{1..self->size() }->forAll(index : Integer |
       result->at(index) = self ->at(index))
```

lemma req_11_7_5−e−1:
  $\tau \models (\partial\ (obj::('a,'b::bot)VAL)) \Longrightarrow$
  $(\|OclAppend\ (self::('a,'b)Sequence)\ obj\|\ \tau) = ((\|self\| + 1)\ \tau)$
  by(simp)
lemma req_11_7_5−e−2:
  $\tau \models (\partial\ (self::('a,'b::bot)Sequence)) \Longrightarrow$
  $(OclAt\ (OclAppend\ self\ (obj::('a,'b)VAL))\ \|OclAppend\ self\ obj\|\ \tau) = (obj\ \tau)$
  apply(case_tac $\tau \models \not\partial$ obj)
  apply(ocl_hypsubst, simp_all)
  apply(rule OCL_Sequence.at_last_of_including)
  apply(rule subst, rule sym)
  apply(rule_tac A=‖self ->including obj ‖ *in* cp_charn)
  apply(rule OCL_Sequence.size_of_including)
  apply(simp_all)
  oops
lemma req_11_7_5−e−3:
  $\models \forall$ index $\in$ ((OclCollectionRange 1 ‖(self::('a,'b::bot)Sequence)‖)::('a,Integer_0)Sequence)
  .
    ((OclAt self index) $\doteq$ (OclAt (OclAppend self (obj::('a,'b)VAL)) index))
  oops

prepend(object : T) : Sequence(T) (11.7.5-f)

The sequence consisting of object, followed by all elements in self.

```
post: result->size = self->size() + 1
post: result->at(1) = object
post: Sequence{1..self->size()}->forAll(index : Integer |
      self->at(index) = result->at(index + 1))
```

lemma req_11_7_5−f−1:
 $\tau \models (\partial\ (obj::('a,'b::bot)VAL)) \Longrightarrow$
 $(\|OclPrepend\ (self::('a,'b)Sequence)\ \ obj\|\ \tau) = ((\|self\| + 1)\ \tau)$
 apply(simp)
 apply(rule trans)
 apply(rule_tac A=‖?S ->including obj ‖ *in* cp_charn)
 apply(rule OCL_Sequence.size_of_including)
 apply(simp_all add: localValidDefined2sem)
 apply(simp add: localValid2sem OCL_Sequence.OclSize_def OclStrictEq_def
  One_ocl_int_def Zero_ocl_int_def plus_def ss_lifting′)
 oops
lemma req_11_7_5−f−2:
 $\tau \models (\partial\ (self::('a,'b::bot)Sequence)) \Longrightarrow$
 $(OclAt\ (OclPrepend\ self\ (obj::('a,'b)VAL))\ 1\ \tau) = (obj\ \tau)$
 by(simp only: at_first_of_prepend)
lemma req_11_7_5−f−3:
 $\models \forall\ index\ \in ((OclCollectionRange\ 1\ \|(self::('a,'b::bot)Sequence)\|)::('a,Integer\_0)Sequence)$
.
 $((OclAt\ self\ index) \doteq (OclAt\ (OclPrepend\ self\ (obj::('a,'b)VAL)\ )\ (index+1)))$
 oops

insertAt(index : Integer, object : T) : Sequence(T) (11.7.5-g)

The sequence consisting of self with object inserted at position index.

```
post: result->size = self->size() + 1
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer |
      self->at(i) = result->at(i))
post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |
      self->at(i) = result->at(i + 1))
```

no preconditions about range of index. But they are surely needed because of the correspondence to at()

lemma req_11_7_5–g–1:
$[\![\tau \models \partial\, (obj::('a,'b::bot)VAL); \tau \models 1 \leq index; \tau \models index \leq (\|self\| + 1)]\!] \Longrightarrow$
$\|OclInsertAt\ self\ index\ obj\|\ \tau = (\|(self::('a,'b)Sequence)\| + 1)\ \tau$
by(simp add: size_of_insertAt)

lemma req_11_7_5–g–2:
$[\![\tau \models 1 \leq index; \tau \models index \leq (\|self\| + 1)]\!] \Longrightarrow$
OclAt (OclInsertAt (self::('a,'b::bot)Sequence) index obj) index τ = (obj::('a,'b)VAL) τ
by(simp add: at_of_insertAt)

lemma req_11_7_5–g–3:
$[\![\tau \models \partial\, obj; \tau \models 1 \leq i; \tau \models index \leq (\|(self::('a,'b::bot)Sequence)\| + 1)]\!] \Longrightarrow$
$\tau \models \forall\, i \in ((OclCollectionRange\ 1\ (index - 1))::('a,Integer\_0)Sequence)\ .$
$((OclAt\ self\ i) \doteq (OclAt\ (OclInsertAt\ self\ index\ (obj::('a,'b)VAL))\ i))$
oops

the valid range must be from `index` to `self->size()`

lemma req_11_7_5–g–4:
$[\![\tau \models \partial\, obj; \tau \models 1 \leq i; \tau \models index \leq (\|(self::('a,'b::bot)Sequence)\| + 1)]\!] \Longrightarrow$
$\tau \models \forall\, i \in ((OclCollectionRange\ index\ \|(self::('a,'b::bot)Sequence)\|)::('a,Integer\_0)Sequence)\ .$
$((OclAt\ self\ i) \doteq (OclAt\ (OclInsertAt\ self\ index\ (obj::('a,'b)VAL))\ (i+1)))$
oops

subSequence(lower : Integer, upper : Integer) : Sequence(T) (11.7.5-h)

The sub-sequence of self starting at number lower, up to and including element number upper.

```
pre : 1 <= lower
pre : lower <= upper
pre : upper <= self->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
        result->at(index - lower + 1) =
                       self->at(index))
```

lemma req_11_7_5−h−2:
$[\![\ \tau \models 1 \leq \text{lower}; \tau \models \text{lower} \leq \text{upper} - 1; \tau \models \text{upper} \leq \|\text{self}\|\ ]\!] \Longrightarrow$
$\|(\text{OclSubSequence (self::('a,'b::bot)Sequence) lower upper})\|\ \ \tau = (\text{upper} - \text{lower} + 1)$
τ
by(simp add: size_of_subsequence)

lemma req_11_7_5−h−2:
$[\![\ \tau \models 1 \leq \text{lower}; \tau \models \text{lower} \leq \text{upper} - 1; \tau \models \text{upper} \leq \|\text{self}\|\ ]\!] \Longrightarrow$
$\tau \models \forall$ index $\in$ ((OclCollectionRange lower upper)::('a,Integer_0)Sequence) .
(OclAt (OclSubSequence self lower upper) (index − lower + 1) $\doteq$
OclAt (self::('a,'b::bot)Sequence) index)
oops

at(i : Integer) : T (11.7.5-i)

The i-th element of sequence.

```
pre : i >= 1 and i <= self->size()
```

lemma req_11_7_5−i:
$[\![\ \tau \models 1 \leq \text{i}; \tau \models \text{i} \leq \|\text{S}\|\ ]\!] \Longrightarrow$
($\tau \models \partial$((OclAt (S::('a,'b::bot)Sequence) i)::('a,'b)VAL))
apply(rotate_tac 1, frule isDefined_if_valid)
apply(simp only: OCL_is_defopt_OclLe OCL_Sequence.OCL_is_defopt_OclSize OCL_is_defopt_OclAt)
oops

indexOf(obj : T) : Integer (11.7.5-j)

The index of object obj in the sequence.

```
pre  : self->includes(obj)
post : self->at(i) = obj
```

lemma req_11_7_5−j:
$[\![$ $\tau \models$ (obj::('a,'b)VAL) $\in$ (self::('a,'b::bot)Sequence) $]\!] \Longrightarrow$
OclAt self (OclIndexOf self obj) τ = obj τ
by(simp add: at_indexOf)

---

first() : T (11.7.5-k)

The first element in self.

```
post: result = self->at(1)
```

lemma req_11_7_5−k:
OclFirst (self::('a,'b::bot)Sequence) = ((OclAt self 1)::('a,'b)VAL)
by(rule first_at_UC)

---

last() : T (11.7.5-l)

The last element in self.

```
post: result = self->at(self->size() )
```

lemma req_11_7_5−l:
OclLast (self::('a,'b::bot)Sequence) = ((OclAt self ||self||)::('a,'b)VAL)
by(rule last_at_UC)

---

including(object : T) : Sequence(T) (11.7.5-m)

The sequence containing all elements of self plus object added as the last element.

```
post: result = self.append(object)
```

lemma req_11_7_5−m:
self `->including` (obj::('a,'b)VAL) = OclAppend (self::('a,'b::bot)Sequence) obj
by(rule append_including_UC[symmetric])

---

excluding(object : T) : Sequence(T) (11.7.5-n)

The sequence containing all elements of self apart from all occurrences of object. The order of the remaining elements is not changed.

```
post:result->includes(object) = false
post: result->size() = self->size() - self->count(object)
post: result = self->iterate(elem; acc : Sequence(T)
   = Sequence{}|
      if elem = object then acc else acc->append(elem) endif )
```

lemma req_11_7_5−n−1:
$[\![\ \tau \models \partial(\text{obj}::('a,'b)\text{VAL});\ \ \tau \models \partial(\text{self}::('a,'b::\text{bot})\text{Sequence})\ ]\!] \Longrightarrow$
(obj $\in$ (self ->excluding obj)) τ = F τ
by(simp add: OCL_Sequence.includes_of_excluding)
lemma req_11_7_5−n−2:
$\|$(self::('a,'b::bot)Sequence) ->excluding (obj::('a,'b)VAL)$\|$ =
$\|$self$\|$ − OclCount self obj
by(simp add: OCL_Sequence.size_of_excluding)
lemma req_11_7_5−n−3:
$[\![\ \tau \models \partial(\text{obj}::('a,'b)\text{VAL})\ ]\!] \Longrightarrow$
((self::('a,'b::bot)Sequence) ->excluding (obj::('a,'b)VAL)) τ =
(self ->iterate(elem; acc=[] | (if (elem $\doteq$ obj) then acc else (OclAppend acc elem)endif))) τ
by(simp add: OCL_Sequence.excluding_by_iterate localValid2sem
OCL_Sequence.OclIterate_def OclIf_def ss_lifting')

---

asBag() : Bag(T) (11.7.5-o)

The Bag containing all the elements from self, including duplicates.

```
post:
  result->forAll(elem | self->count(elem)=result->count(elem))
post:
  self->forAll(elem | self->count(elem)=result->count(elem))
```

lemma req_11_7_5−o−1:
$\models \forall$ elem $\in$ ((->asBag() (self::('a,'b::bot)Sequence))::('a,'b::bot)Bag) .
((self ->count elem) $\doteq$ (((->asBag() self)::('a,'b::bot)Bag) ->count elem))
oops
lemma req_11_7_5−o−2:
$\models \forall$ elem $\in$ (self::('a,'b::bot)Sequence) .
((self ->count elem) $\doteq$ (((->asBag() self)::('a,'b::bot)Bag) ->count elem))
oops

---

asSequence() : Sequence(T) (11.7.5-p)

The Sequence identical to the object itself. This operation exists for convenience reasons.

```
post: result = self
```

lemma req_11_7_5–p:
((->asSequence() (self::('a,'b::bot)Sequence))::('a,'b::bot)Sequence) = self
by(simp)

asSet() : Set(T) (11.7.5-q)

The Set containing all the elements from self, with duplicated removed.

```
post: result->forAll(elem | self ->includes(elem))
post: self ->forAll(elem | result->includes(elem))
```

lemma req_11_7_5–q–1:
⊨ ∀ elem ∈ ((->asSet() (self::('a,'b::bot)Sequence))::('a,'b Set_0)VAL) . ((elem::('a,'b)VAL) ∈ self)
oops
lemma req_11_7_5–q–2:
⊨ ∀ elem ∈ (self::('a,'b::bot)Sequence) . ((elem::('a,'b)VAL) ∈((->asSet() self)::('a,'b Set_0)VAL))
oops

asOrderedSet() : OrderedSet(T) (11.7.5-r)

An OrderedSet that contains all the elements from self, in the same order, with duplicates removed.

```
post: result->forAll(elem | self ->includes(elem))
post: self ->forAll(elem | result->includes(elem))
post: self ->forAll(elem | result->count(elem) = 1)
post: self ->forAll(elem1, elem2 |
             self->indexOf(elem1) < self->indexOf(elem2)
      implies result->indexOf(elem1) < result->indexOf(elem2) )
```

lemma req_11_7_5–q–1:
⊨ ∀ elem ∈ ((->asOrderedSet() (self::('a,'b::bot)Sequence))::('a,'b OrderedSet_0)VAL) . ((elem::('a,'b)VAL) ∈ self)
oops
lemma req_11_7_5–q–2:
⊨ ∀ elem ∈ (self::('a,'b::bot)Sequence) . ((elem::('a,'b)VAL) ∈((->asOrderedSet() self)::('a,'b::bot OrderedSet_0)VAL))
oops

end

B.6.3. OCL Iterator requirements

theory *OCL_Iterator_requirements*
imports
$*HOLOCL_HOME/src/OCL_Library*
begin

This theory contains the requirements for iterators defined in the OCL standard library [41, Chapter 11.9].

Collection

---

exists (11.9.1-a)

Results in true if body evaluates to true for at least one element in the source collection.

```
source->exists(iterators | body) =
        source->iterate(iterators; result : Boolean = false
                        | result or body)
```

Set

OrderedSet

Bag

lemma req_bag_11_9_1−a:
(∃ x ∈ (source::('a,'b::bot)Bag) . (P::('a,'b::bot)VAL ⇒ 'a Boolean) x) =
(source ->iterate(x;y= F | (P x) ∨ y))
by(rule OCL_Bag.exists_by_iterate)

Sequence

lemma req_seq_11_9_1−a:
(∃ x ∈ (source::('a,'b::bot)Sequence) . (P::('a,'b::bot)VAL ⇒ 'a Boolean) x) =
(source ->iterate(x;y= F | (P x) ∨ y))
by(rule OCL_Sequence.exists_by_iterate)

---

forAll (11.9.1-b)

Results in true if the body expression evaluates to true for each element in the source collection; otherwise, result is false

```
source->forAll(iterators | body ) =
          source->iterate(iterators; result : Boolean = true
                          | result and body)
```

Set

OrderedSet

Bag

lemma req_bag_11_9_1−b:
(∀ x ∈ (source::('a,'b::bot)Bag) . (P::('a,'b::bot)VAL ⇒ 'a Boolean) x) =
(source ->iterate(x;y= T | (P x) ∧ y))
by(rule OCL_Bag.forall_by_iterate)

Sequence

lemma req_seq_11_9_1−b:
(∀ x ∈ (source::('a,'b::bot)Sequence) . (P::('a,'b::bot)VAL ⇒ 'a Boolean) x) =
(source ->iterate(x;y= T | (P x) ∧ y))
by(rule OCL_Sequence.forall_by_iterate)

---

isUnique (11.9.1-c)

Results in true if body evaluates to a different value for each element in the source collection; otherwise, result is false.

```
source->isUnique (iterators | body) =
source->collect(iterators
             | Tuple{iter = Tuple{iterators}, value = body})
   ->forAll (x, y | (x.iter <> y.iter)
                    implies (x.value <> y.value))
```

isUnique may have at most one iterator variable.

Set

OrderedSet

Bag

Sequence

any (11.9.1-d)

Returns any element in the source collection for which body evaluates to true. If there is more than one element for which body is true, one of them is returned. There must be at least one element fulfilling body, otherwise the result of this IteratorExp is OclUndefined.

```
source->any(iterator | body) =
    source->select(iterator | body)->asSequence()->first()
```

any may have at most one iterator variable.

Set

OrderedSet

Bag

*lemma*req_bag_11_9_1−d:
OCL_Collection.OclAny (Source::('a,'b::bot)Bag) (P::(('a,'b::bot)VAL ⇒ 'a Boolean)) =
OclFirst ((->asSequence() (⦇x : source | (P x)⦈)::('a,'b::bot)Sequence))
oops

Sequence

lemma req_seq_11_9_1−d:
OCL_Collection.OclAny (source::('a,'b::bot)Sequence) (P::('a,'b::bot)VAL ⇒ 'a Boolean) =
OclFirst ((->asSequence() (⦇x : source | (P x)⦈)::('a,'b::bot)Sequence))
by(rule OCL_Sequence.any_by_first_of_select)

one (11.9.1-e)

Results in true if there is exactly one element in the source collection for which body is true.

```
source->one(iterator | body) =
   source->select(iterator | body)->size() = 1
```

one may have at most one iterator variable.

Set

OrderedSet

Bag

lemmareq_bag_11_9_1−e:
(OclOne (source::('a,'b::bot)Bag) (P::('a,'b::bot)VAL ⇒ 'a Boolean)) =
($\|(\{\!|x : source \mid (P\ x)|\!\})\| \doteq 1$)
by(simp add: OCL_Bag.OclOne_def)

Sequence

lemma req_seq_11_9_1−e:
(OclOne (source::('a,'b::bot)Sequence) (P::('a,'b::bot)VAL ⇒ 'a Boolean)) =
($\|(\{\!|x : source \mid (P\ x)|\!\})\| \doteq 1$)
by(rule OCL_Sequence.one_by_size_of_select)

collect (11.9.1-f)

The Collection of elements which results from applying body to every member of the source set. The result is flattened. Notice that this is based on collectNested, which can be of different type depending on the type of source. collectNested is defined individually for each subclass of CollectionType.

```
source->collect (iterators | body)
     = source->collectNested (iterators | body)->flatten()
```

collect may have at most one iterator variable.

Set

OrderedSet

Bag

*lemma*req_bag_11_7_4−f:
OclCollect S P ≡ ⟦(OclCollectNested S P)⟧
oops

Sequence

lemma req_seq_11_9_1−f:
OclCollect S P ≡ ⟦(OclCollectNested S P)⟧
oops

Set

Bag

Sequence

---

select(expression : OclExpression) : Sequence(T) (11.9.4-a)

The subsequence of the source sequence for which body is true.

```
source->select(iterator | body) =
  source->iterate(iterator; result : Sequence(T)=Sequence{} |
            if body then result->including(iterator)
                 else result
            endif)
```

select may have at most one iterator variable. *lemma* req_11_9_4−a:
((|x : (source::('a,'b::bot)Sequence) | ((P::('a,'b::bot)VAL ⇒ 'a Boolean) x)|)) =
(source ->iterate(iterator; result = [] |
if (P iterator)then (OclIncluding result iterator) else result endif))
by(rule OCL_Sequence.select_by_iterate)

---

reject (11.9.4-b)

The subsequence of the source sequence for which body is false.

```
source->reject(iterator | body) =
     source->select(iterator | not body)
```

reject may have at most one iterator variable. ***lemma*** req_11_9_4−b:
OclReject source P = ((|x : (source::('a,'b::bot)Sequence) | (¬ (P::('a,'b::bot)VAL ⇒ 'a Boolean) x)|))
by(rule OCL_Sequence.reject_by_select)

---

collectNested (11.9.4-c)

The Sequence of elements which results from applying body to every member of the source sequence.

```
source->collect(iterators | body) =
     source->iterate(iterators; result : Sequence(body.type)
                  = Sequence{} |
                  result->append(body ) )
```

collectNested may have at most one iterator variable. ***lemma*** req_11_9_4−c:
((OclCollectNested (source::('a,'b::bot)Sequence) (P::('a,'b)VAL ⇒ ('a,'c)VAL))::('a,'c::bot)Sequence) =
(source ->iterate(iterators; result=[] | result ->including (P iterators)))
by(rule OCL_Sequence.collectNested_by_iterate)

---

sortedBy (11.9.4-d)

Results in the Sequence containing all elements of the source collection. The element for which body has the lowest value comes first, and so on. The type of the body expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c.

```
source->sortedBy(iterator | body) =
  iterate( iterator ; result : Sequence(T) : Sequence {} |
      if result->isEmpty() then
        result.append(iterator)
      else
        let position : Integer = result->indexOf (
             result->select (item
               | body (item) > body (iterator)) ->first() )
        in
         result.insertAt(position, iterator)
      endif
```

sortedBy may have at most one iterator variable.

---

end

B.7. OCL

B.7.1. The Object Constraint Language: OCL

theory *OCL*
imports
  $*HOLOCL_HOME/src/UML_OCL*
  $*HOLOCL_HOME/src/library/basic/OCL_Basic_requirements*
  $*HOLOCL_HOME/src/library/collection/*$*COLLECTION/OCL_Collection_requirements*
  $*HOLOCL_HOME/src/library/collection/*$*COLLECTION/OCL_Iterator_requirements*
begin end

Appendix C.

Encoding UML/OCL by Example

C.1. Encoding UML/OCL into HOL by Example

C.2. A Simple Example

In this section we present the details of our encoding by an tiny example. We use the class diagram presented in Figure C.1 together with the OCL specification shown in Listing C.1. Our Example consists only out of two classes with an inheritance relationship. Class `A` has one attribute `i` of type `Integer` that is constraint by an invariant. Further, class `A` has a method `f(x:Integer):Integer` which is described by a pre-condition and post-condition. Class `B` only has one attribute `j` of type `B`. Note that this is, in contrast to the attribute `i` of class `A`, an object type. In the following we describe which definition and theorems are generated during the import of this simple class diagram into HOL-OCL.

C.3. Importing UML/OCL models

UMLOCL models can be directly imported into HOL-OCL if they are available in XMI format, e.g.,

load_xmi *encoding_example_ocl.xmi*

```
package encoding
  context A::i:Integer
    init: 42

  context A
    inv:  i >= 0

  context A::f(x:Integer):Integer
    pre:  x > 0
    post: result = x.div(i)
endpackage
```

Listing C.1: A simple Encoding Example: OCL specification

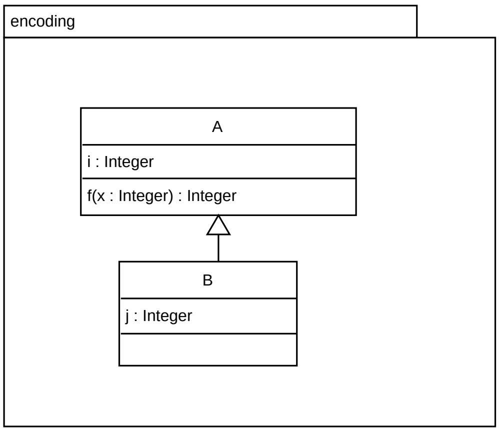


Figure C.1.: A simple Encoding Example: Data Model

imports the UML/OCL specification of our example. During the the import, different definitions and theorems are generated into the current theory context. HOL-OCL organizes these automatically generated stuff into a hierarchy of namespaces, i.e., in our example:

UML_OCL: This is the *standard* namespace for internal definitions, i.e., definitions that normally should not be needed by the users. This namespace is divided as follows:

- *univ.model*: contains the basic type definitions of classes and the type of the universe
- *level0.model*: contains the level 0 definitions of the model.
- *level1.model*: contains the level 1 definitions of the model.

model: It contains all definitions a user would expect, namely all the level 2 stuff which ties the UML and OCL part together.

In all namespaces, "*model*" is replaced by the top-level UML package of the encoded model, i.e., for our example "*encoding*". Thus, in our example all level 0 definitions are defined within "*UML_OCL.level0.encoding*".

C.3.1. Encoding Level 0:

The Universe Type

The first major step of encoding classes into the base OCL universe is to construct the universe type, but first we have to construct the core types for classes. For each class we construct a “identifier” type which is crucial for our strong type discipline:

$$UML\_OCL.univ.encoding.A\_key = A\_key$$

the base type of the class, describing its locally defined attributes

$$UML\_OCL.univ.encoding.A\_base\_type = A\_key \times Integer\_0$$

the type of the class, describing the inherited type together with the base type

$$\begin{aligned} UML\_OCL.univ.encoding.A\_type &= A\_key \times Integer\_0 \\ 'a\ UML\_OCL.univ.encoding.A\_ext &= (OclAny\_key \times oid) \times ((A\_key \times Integer\_0) \times 'a\ up)\ up \end{aligned}$$

These types are analogous defined for the inherited class `B`:

$$\begin{aligned} UML\_OCL.univ.encoding.B\_key &= B\_key \\ UML\_OCL.univ.encoding.B\_base\_type &= B\_key \times oid \\ UML\_OCL.univ.encoding.B\_type &= (A\_key \times Integer\_0) \times B\_key \times oid \\ 'a\ UML\_OCL.univ.encoding.B\_ext &= (OclAny\_key \times oid) \times \\ &\qquad ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up)\ up)\ up \end{aligned}$$

Together with the pre-defined OCL universe type they build the basis for the model specific universe type:

$$\begin{aligned} &('a,\ 'b,\ 'c)\ U\_encoding\_example \\ &\qquad = U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \end{aligned}$$

Now we can encode types for classes represented in the universe, e.g. for class `A`:

$$\begin{aligned} &('a,\ 'b)\ UML\_OCL.univ.encoding.A \\ &\qquad = OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up) \end{aligned}$$

and for class `B`:

$$\begin{aligned} &'a\ UML\_OCL.univ.encoding.B \\ &\qquad = OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up)\ up) \end{aligned}$$

Defining the Basic Methods over Classes

For each class we define two type tests for objects, one for testing if a object is of a specific type or suptype (i.e., this test is true, if the object is of the given kind):

$$\begin{aligned} &UML\_OCL.level0.encoding.is\_A :: \\ &OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up) \Rightarrow bool \\ &level0.encoding.is\_A \equiv DEF \circ base \end{aligned}$$

and one for testing if a object is exactly the requested type

$$
\begin{aligned}
&UML\_OCL.level0.encoding.isType\_A ::\\
&OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up) \Rightarrow bool\\
&level0.encoding.isType\_A \equiv\\
&\qquad \lambda obj.\ level0.encoding.is\_A\ obj \wedge \neg\ (DEF \circ base \circ base)\ obj
\end{aligned}
$$

Further, we define a type for universe instances:

$$
\begin{aligned}
&UML\_OCL.level0.encoding.is\_A\_univ ::\\
&\quad U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \Rightarrow bool\\
&level0.encoding.is\_A\_univ\ ?univ \equiv\\
&level0.is\_OclAny\_univ\ ?univ \wedge (DEF \circ FromL \circ base \circ level0.get\_OclAny)\ ?univ
\end{aligned}
$$

Analogous we define for the inherited class `B`:

$$
\begin{aligned}
&UML\_OCL.level0.encoding.is\_B ::\\
&\quad OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up) \Rightarrow bool\\
&level0.encoding.is\_B \equiv DEF \circ base \circ base
\end{aligned}
$$

$$
\begin{aligned}
&UML\_OCL.level0.encoding.isType\_B ::\\
&\quad OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up) \Rightarrow bool\\
&level0.encoding.isType\_B \equiv\\
&\qquad \lambda obj.\ level0.encoding.is\_B\ obj \wedge \neg\ (DEF \circ base \circ base \circ base)\ obj
\end{aligned}
$$

$$
\begin{aligned}
&UML\_OCL.level0.encoding.is\_B\_univ ::\\
&\quad U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \Rightarrow bool\\
&level0.encoding.is\_B\_univ\ ?univ \equiv\\
&\qquad level0.encoding.is\_A\_univ\ ?univ \wedge\\
&\qquad (DEF \circ FromL \circ base \circ base \circ level0.encoding.get\_A)\ ?univ
\end{aligned}
$$

We define a functions for embeding a class into the corresponding universe

$$UML\_OCL.level0.encoding.mk\_A ::$$
$$OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)$$
$$\Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)$$
$$level0.encoding.mk\_A\ ?obj \equiv$$
$$level0.mk\_OclAny\ \llcorner(sup\ ?obj, (lift \circ Inl \circ Lifting.drop \circ base)\ ?obj)\lrcorner$$

and one for extracting an object out of a given universe:

$$UML\_OCL.level0.encoding.get\_A ::$$
$$U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)$$
$$\Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)$$
$$level0.encoding.get\_A\ ?univ \equiv$$
$$\llcorner((sup \circ level0.get\_OclAny)\ ?univ, (FromL \circ base \circ level0.get\_OclAny)\ ?univ)\lrcorner$$

These two functions can also be understood as conversion between object types and universe types.

Analogous we define for the inherited class `B`:

$$UML\_OCL.level0.encoding.mk\_B ::$$
$$OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)$$
$$\Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)$$
$$level0.encoding.mk\_B\ ?obj \equiv$$
$$level0.encoding.mk\_A$$
$$\llcorner(sup\ ?obj,$$
$$\llcorner((sup \circ base)\ ?obj, (lift \circ Inl \circ Lifting.drop \circ base \circ base)\ ?obj)\lrcorner)\lrcorner$$

$$UML\_OCL.level0.encoding.get\_B ::$$
$$U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)$$
$$\Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)$$
$$level0.encoding.get\_B\ ?univ \equiv$$
$$\llcorner((sup \circ level0.encoding.get\_A)\ ?univ,$$
$$\llcorner((sup \circ base \circ level0.encoding.get\_A)\ ?univ,$$
$$(FromL \circ base \circ base \circ level0.encoding.get\_A)\ ?univ)\lrcorner)\lrcorner$$

For converting along type hierarchies (i.e. down-casting and up-casting) we provide for each class two functions for the conversion to the direct predecessor and direct successor in the class hierarchy:

$$\begin{aligned}
&UML_OCL.level0.encoding.A_2_OclAny ::\\
&\quad OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up)\\
&\Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&level0.encoding.A_2_OclAny \equiv level0.get_OclAny \circ level0.encoding.mk_A
\end{aligned}$$

$$\begin{aligned}
&UML_OCL.level0.encoding.OclAny_2_A ::\\
&\quad OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\quad\quad \Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up)\\
&level0.encoding.OclAny_2_A \equiv level0.encoding.get_A \circ level0.mk_OclAny
\end{aligned}$$

Analogous we define for the inherited class `B`:

$$\begin{aligned}
&UML_OCL.level0.encoding.B_2_A ::\\
&\quad OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up)\ up)\\
&\quad\quad \Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up)\\
&level0.encoding.B_2_A \equiv level0.encoding.get_A \circ level0.encoding.mk_B
\end{aligned}$$

$$\begin{aligned}
&UML_OCL.level0.encoding.A_2_B ::\\
&\quad OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up)\\
&\quad\quad \Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up)\ up)\\
&level0.encoding.A_2_B \equiv level0.encoding.get_B \circ level0.encoding.mk_A
\end{aligned}$$

Finally, we provide accesor functions for the class attributes:

$$\begin{aligned}
&UML_OCL.level0.encoding.A.i ::\\
&OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times 'a\ up + 'b)\ up) \Rightarrow Integer_0\\
&level0.encoding.A.i \equiv snd \circ fst \circ Lifting.drop \circ base
\end{aligned}$$

$UML\_OCL.level0.encoding.B.j ::$
$\quad OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up) \Rightarrow oid$
$level0.encoding.B.j \equiv snd \circ fst \circ Lifting.drop \circ base \circ base$

Proving Properties

Already on this level, we can prove important properties of our encoding. For example, the following theorem "UML_OCL.level0.encoding.is_A_univ_implies_OclAny_univ" shows, that on the universe types, every object of class `A` is also an object of class `OclAny`:

$$level0.encoding.is\_A\_univ\ ?obj \Longrightarrow level0.is\_OclAny\_univ\ ?obj$$

Using get_mk_A_id and get_mk_A_id we can convert "lossless" between our class type representation and universe representation, this is shown by "UML_OCL.level0.encoding.get_mk_A_id":

$$level0.encoding.is\_A\ ?obj \Longrightarrow$$
$$level0.encoding.get\_A\ (level0.encoding.mk\_A\ ?obj) = ?obj$$

and "UML_OCL.level0.encoding.mk_get_A_id":

$$level0.encoding.is\_A\_univ\ ?univ \Longrightarrow$$
$$level0.encoding.mk\_A\ (level0.encoding.get\_A\ ?univ) = ?univ$$

and "UML_OCL.level0.encoding.is_A_univ_implies_is_get" shows that a successfull type test on the universe types also implies a successfull type test on the class types:

$$level0.encoding.is\_A\_univ\ ?univ \Longrightarrow$$
$$level0.encoding.is\_A\ (level0.encoding.get\_A\ ?univ)$$

The theorem "UML_OCL.level0.encoding.is_mk_A" shows that the constructor creates an object that is of type `A`:

$$level0.encoding.is\_A\ ?obj \Longrightarrow$$
$$level0.encoding.is\_A\_univ\ (level0.encoding.mk\_A\ ?obj)$$

and also satisfies the test of its super-type, e.g., `OclAny`. This is shown by the following theorem "UML_OCL.level0.encoding.is_OclAny_mk_A":

$$level0.encoding.is\_A\ ?obj \Longrightarrow level0.is\_OclAny\_univ\ (level0.encoding.mk\_A\ ?obj)$$

"UML_OCL.level0.encoding.cast_A_id" shows that we can successfully downcast objects:

$$level0.encoding.is\_A\ ?obj \Longrightarrow \\ level0.encoding.OclAny\_2\_A\ (level0.encoding.A\_2\_OclAny\ ?obj) = ?obj$$

Analogous, these properties are also shown for class `B`:
UML_OCL.level0.encoding.is_B_univ_implies_A_univ:

$$level0.encoding.is\_B\_univ\ ?obj \Longrightarrow level0.encoding.is\_A\_univ\ ?obj$$

UML_OCL.level0.encoding.is_B_univ_implies_is_get:

$$level0.encoding.is\_B\_univ\ ?univ \Longrightarrow \\ level0.encoding.is\_B\ (level0.encoding.get\_B\ ?univ)$$

UML_OCL.level0.encoding.get_mk_B_id:

$$level0.encoding.is\_B\ ?obj \Longrightarrow \\ level0.encoding.get\_B\ (level0.encoding.mk\_B\ ?obj) = ?obj$$

UML_OCL.level0.encoding.is_mk_B:

$$level0.encoding.is\_B\ ?obj \Longrightarrow \\ level0.encoding.is\_B\_univ\ (level0.encoding.mk\_B\ ?obj)$$

UML_OCL.level0.encoding.is_A_mk_B:

$$level0.encoding.is\_B\ ?obj \Longrightarrow \\ level0.encoding.is\_A\_univ\ (level0.encoding.mk\_B\ ?obj)$$

UML_OCL.level0.encoding.mk_get_B_id:

$$level0.encoding.is\_B\_univ\ ?univ \Longrightarrow$$
$$level0.encoding.mk\_B\ (level0.encoding.get\_B\ ?univ) = ?univ$$

UML_OCL.level0.encoding.cast_B_id:

$$level0.encoding.is\_B\ ?obj \Longrightarrow$$
$$level0.encoding.A\_2\_B\ (level0.encoding.B\_2\_A\ ?obj) = ?obj$$

C.3.2. Encoding Level 1:

The encoding on level 1 follows strictly the schema from level 0, i.e, we also start two type tests for objects, one for testing if a object is of a specific type or subtype (i.e., this test is true, if the object is of the given kind):

$$
\begin{aligned}
&UML\_OCL.level1.encoding.is\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\quad\ \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad\ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow Boolean\_0\\
&level1.encoding.is\_A \equiv lift_1\ (strictify\ (lift \circ level0.encoding.is\_A))
\end{aligned}
$$

and one for testing if a object is exactly the requested type

$$
\begin{aligned}
&UML\_OCL.level1.encoding.isType\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\quad\ \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad\ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow Boolean\_0\\
&level1.encoding.isType\_A \equiv lift_1\ (strictify\ (lift \circ level0.encoding.isType\_A))
\end{aligned}
$$

Further, we define a type for universe instances:

$$
\begin{array}{l}
UML\_OCL.level1.encoding.is\_A\_univ :: \\
\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})) \\
\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
\hfill \Rightarrow Boolean\_0 \\
level1.encoding.is\_A\_univ \equiv lift_1\ (lift \circ level0.encoding.is\_A\_univ)
\end{array}
$$

Analogous we define for the inherited class `B`:

$$
\begin{array}{l}
UML\_OCL.level1.encoding.is\_B :: \\
\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up)) \\
\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
\hfill \Rightarrow Boolean\_0 \\
level1.encoding.is\_B \equiv lift_1\ (strictify\ (lift \circ level0.encoding.is\_B))
\end{array}
$$

$$
\begin{array}{l}
UML\_OCL.level1.encoding.isType\_B :: \\
\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up)) \\
\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
\hfill \Rightarrow Boolean\_0 \\
level1.encoding.isType\_B \equiv lift_1\ (strictify\ (lift \circ level0.encoding.isType\_B))
\end{array}
$$

$$
\begin{aligned}
&UML\_OCL.level1.encoding.is\_B\_univ ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad\qquad \Rightarrow Boolean\_0\\
&level1.encoding.is\_B\_univ \equiv lift_1\ (lift \circ level0.encoding.is\_B\_univ)
\end{aligned}
$$

We define a functions for embedding a class into the corresponding universe

$$
\begin{aligned}
&UML\_OCL.level1.encoding.mk\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad\quad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&level1.encoding.mk\_A \equiv lift_1\ level0.encoding.mk\_A
\end{aligned}
$$

and one for extracting an object out of a given universe:

$$
\begin{aligned}
&UML\_OCL.level1.encoding.get\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up)\\
&level1.encoding.get\_A \equiv lift_1\ level0.encoding.get\_A
\end{aligned}
$$

These two functions can also be understood as conversion between object types and universe types.

Analogous we define for the inherited class `B`:

$$
\begin{aligned}
&UML\_OCL.level1.encoding.mk\_B :: \\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up)) \\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\quad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&level1.encoding.mk\_B \equiv lift_1\ level0.encoding.mk\_B
\end{aligned}
$$

$$
\begin{aligned}
&UML\_OCL.level1.encoding.get\_B :: \\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})) \\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up) \\
&level1.encoding.get\_B \equiv lift_1\ level0.encoding.get\_B
\end{aligned}
$$

For converting along type hierarchies (i.e. down-casting and up-casting) we provide for each class two functions for the conversion to the direct predecessor and direct successor in the class hierarchy:

$$
\begin{aligned}
&UML\_OCL.level1.encoding.A\_2\_OclAny :: \\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up)) \\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&level1.encoding.A\_2\_OclAny \equiv lift_1\ level0.encoding.A\_2\_OclAny
\end{aligned}
$$

$$
\begin{aligned}
&UML\_OCL.level1.encoding.OclAny\_2\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)\\
&level1.encoding.OclAny\_2\_A \equiv lift_1\ level0.encoding.OclAny\_2\_A
\end{aligned}
$$

Analogous we define for the inherited class `B`:

$$
\begin{aligned}
&UML\_OCL.level1.encoding.B\_2\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)\\
&level1.encoding.B\_2\_A \equiv lift_1\ level0.encoding.B\_2\_A
\end{aligned}
$$

$$
\begin{aligned}
&UML\_OCL.level1.encoding.A\_2\_B ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)\\
&level1.encoding.A\_2\_B \equiv lift_1\ level0.encoding.A\_2\_B
\end{aligned}
$$

Finally, we provide accessor functions for the class attributes:

$$
\begin{aligned}
&UML\_OCL.level1.encoding.A.i ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\hfill \Rightarrow Integer\_0\\
&level1.encoding.A.i \equiv lift_1\ (strictify\ level0.encoding.A.i)
\end{aligned}
$$

$$
\begin{aligned}
&UML\_OCL.level1.encoding.B.j ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up))\\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)\\
&level1.encoding.B.j \equiv\\
&\qquad \lambda X\ St.\ case\ fst\ St\ (level0.encoding.B.j\ (X\ St))\ of\ None \Rightarrow \bot\\
&\qquad\qquad\qquad \mid Some\ x \Rightarrow level0.encoding.get\_B\ x
\end{aligned}
$$

Proving Properties

Already on this level, we can prove important properties of our encoding. For example, the following theorem "UML_OCL.level1.encoding.is_A_univ_implies_OclAny_univ" shows, that on the universe types, every object of class `A` is also an object of class `OclAny`:

$$?St \vDash level1.encoding.is\_A\_univ\ ?obj \Longrightarrow ?St \vDash level1.is\_OclAny\_univ\ ?obj$$

Using get_mk_A_id and get_mk_A_id we can convert "lossless" between our class type representation and universe representation, this is shown by "UML_OCL.level1.encoding.get_mk_A_id":

$$?St \models level1.encoding.is\_A\ ?obj \Longrightarrow \\ level1.encoding.get\_A\ (level1.encoding.mk\_A\ ?obj)\ ?St = ?obj\ ?St$$

and "UML_OCL.level1.encoding.mk_get_A_id":

$$?St \models level1.encoding.is\_A\_univ\ ?univ \Longrightarrow \\ level1.encoding.mk\_A\ (level1.encoding.get\_A\ ?univ)\ ?St = ?univ\ ?St$$

and "UML_OCL.level1.encoding.is_A_univ_implies_is_get" shows that a successfull type test on the universe types also implies a successfull type test on the class types:

$$[\![DEF\ (level1.encoding.get\_A\ ?univ\ ?St);\ ?St \models level1.encoding.is\_A\_univ\ ?univ]\!] \\ \Longrightarrow ?St \models level1.encoding.is\_A\ (level1.encoding.get\_A\ ?univ)$$

The theorem "UML_OCL.level1.encoding.is_mk_A" shows that the constructor creates an object that is of type `A`:

$$?St \models level1.encoding.is\_A\ ?obj \Longrightarrow \\ ?St \models level1.encoding.is\_A\_univ\ (level1.encoding.mk\_A\ ?obj)$$

and also satisfies the test of its super-type, e.g., `OclAny`. This is shown by the following theorem "UML_OCL.level1.encoding.is_OclAny_mk_A":

$$?St \models level1.encoding.is\_A\ ?obj \Longrightarrow \\ ?St \models level1.is\_OclAny\_univ\ (level1.encoding.mk\_A\ ?obj)$$

"UML_OCL.level1.encoding.cast_A_id" shows that we can successfully downcast objects:

$$?St \models level1.encoding.is\_A\ ?obj \Longrightarrow \\ level1.encoding.OclAny\_2\_A\ (level1.encoding.A\_2\_OclAny\ ?obj)\ ?St = ?obj\ ?St$$

Analogous, these properties are also shown for class `B`:
UML_OCL.level1.encoding.is_B_univ_implies_A_univ:

$$?St \models level1.encoding.is\_B\_univ\ ?obj \Longrightarrow ?St \models level1.encoding.is\_A\_univ\ ?obj$$

UML_OCL.level1.encoding.is_B_univ_implies_is_get:

$$[\![DEF\ (level1.encoding.get\_B\ ?univ\ ?St); ?St \models level1.encoding.is\_B\_univ\ ?univ]\!] \Longrightarrow ?St \models level1.encoding.is\_B\ (level1.encoding.get\_B\ ?univ)$$

UML_OCL.level1.encoding.get_mk_B_id:

$$?St \models level1.encoding.is\_B\ ?obj \Longrightarrow level1.encoding.get\_B\ (level1.encoding.mk\_B\ ?obj)\ ?St = ?obj\ ?St$$

UML_OCL.level1.encoding.is_mk_B:

$$?St \models level1.encoding.is\_B\ ?obj \Longrightarrow ?St \models level1.encoding.is\_B\_univ\ (level1.encoding.mk\_B\ ?obj)$$

UML_OCL.level1.encoding.is_A_mk_B:

$$?St \models level1.encoding.is\_B\ ?obj \Longrightarrow ?St \models level1.encoding.is\_A\_univ\ (level1.encoding.mk\_B\ ?obj)$$

UML_OCL.level1.encoding.mk_get_B_id:

$$?St \models level1.encoding.is\_B\_univ\ ?univ \Longrightarrow level1.encoding.mk\_B\ (level1.encoding.get\_B\ ?univ)\ ?St = ?univ\ ?St$$

UML_OCL.level1.encoding.cast_B_id:

$$?St \models level1.encoding.is\_B\ ?obj \Longrightarrow level1.encoding.A\_2\_B\ (level1.encoding.B\_2\_A\ ?obj)\ ?St = ?obj\ ?St$$

Further we show that our definitions are context passing:
UML_OCL.level1.encoding.cp_get_A:

$cp\ level1.encoding.get\_A$

UML_OCL.level1.encoding.cp_mk_A:

$cp\ level1.encoding.mk\_A$

UML_OCL.level1.encoding.cp_is_A:

$cp\ level1.encoding.is\_A$

UML_OCL.level1.encoding.cp_is_univ_A:

$cp\ level1.encoding.is\_A\_univ$

UML_OCL.level1.encoding.cp_get_B:

$cp\ level1.encoding.get\_B$

UML_OCL.level1.encoding.cp_mk_B:

$cp\ level1.encoding.mk\_B$

UML_OCL.level1.encoding.cp_is_B:

$cp\ level1.encoding.is\_B$

UML_OCL.level1.encoding.cp_is_univ_B:

$cp\ level1.encoding.is\_B\_univ$

C.3.3. Encoding Level 2:

Encoding of OCL Invariants: Part One

Here we bring the data part (i.e., UML) together with the OCL specification. The encoding of OCL formulae is two-staged. First, the formulae are encoding without requiring the definedness of the occurring path expressions, we extend these formulae later with the required definedness constraints. This two-staged process allows us to break cyclic dependencies and thus encode OCL conservatively.

Thus, for class `A` we generate two invariants; The first one describes the initial value of the attribute `i`

$$
\begin{aligned}
&UML\_OCL.level1.encoding.A.model.hol\_invariant.init\_i ::\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\quad\ \ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\ set\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad\ \ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\\
&\hspace{20em} \Rightarrow bool
\end{aligned}
$$

$$
\begin{aligned}
&model.hol\_invariant.init\_i \equiv\\
&\qquad \lambda\tau\ C\ self.\ \tau \models OclIsNew\ self \longrightarrow \tau \models level1.encoding.A.i\ self \triangleq 42
\end{aligned}
$$

and the second one describes the user defined invariant:

$$
\begin{aligned}
&UML\_OCL.level1.encoding.A.model.hol\_invariant.inv\_1 ::\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\quad\ \ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\ set\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad\ \ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\\
&\hspace{20em} \Rightarrow bool
\end{aligned}
$$

$$
model.hol\_invariant.inv\_1 \equiv \lambda\tau\ C\ self.\ \tau \models level1.encoding.A.i\ self \geq 0
$$

The overall invariant for class `A` is constructed by conjoining these invariants.

$$
\begin{aligned}
&UML\_OCL.level1.encoding.A.model.hol\_invariant ::\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\quad\ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up))\ set\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad\ state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up))\\
&\hfill \Rightarrow bool
\end{aligned}
$$

$$
\begin{gathered}
A.model.hol\_invariant \equiv\\
\lambda\tau\ C\ obj.\\
model.hol\_invariant.init\_i\ \tau\ C\ obj \wedge\\
model.hol\_invariant.inv\_1\ \tau\ C\ obj \wedge\\
A.model.hol\_invariant.check\_parent\_inv\ \tau\ C\ obj
\end{gathered}
$$

Analgous we are encoding the pre-condition and post-condition of the operation `f`:

$$
\begin{aligned}
&encoding.A.f\_Integer\_Integer.pre1.pre\_0 ::\\
&\ (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up))\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\hspace{14em} \Rightarrow Integer\_0)\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\hfill \Rightarrow Boolean\_0
\end{aligned}
$$

$$pre\_0 \equiv \lambda self\ x.\ x > 0$$

$$\begin{aligned}
&encoding.A.f_Integer_Integer.post1.post_0 ::\\
&\quad (state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow (state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Integer_0)\\
&\quad \Rightarrow (state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Integer_0)\\
&\quad \Rightarrow state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Boolean_0
\end{aligned}$$

$$post\_0 \equiv \lambda self\ x\ result.\ result \triangleq x\ {-}{>}div(\ level1.encoding.A.i\ self\ )$$

$$\begin{aligned}
&encoding.A.f_Integer_Integer.pre1 ::\\
&\quad (state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow (state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Integer_0)\\
&\quad \Rightarrow state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Boolean_0
\end{aligned}$$

$$pre1 \equiv pre\_0$$

$$
\begin{aligned}
&encoding.A.f\_Integer\_Integer.post1 ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow Boolean\_0\\
&post1 \equiv post\_0
\end{aligned}
$$

For class `B` no initial values or invariants are given, thus we generate invariants that are just `true`:

$$
\begin{aligned}
&encoding.B.inv.init\_j ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow Boolean\_0\\
&inv.init\_j \equiv \lambda self.\ \mathsf{T}
\end{aligned}
$$

$$
\begin{aligned}
&encoding.B.inv.init\_j ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow Boolean\_0\\
&inv.init\_j \equiv \lambda self.\ \mathsf{T}
\end{aligned}
$$

The overall invariant for class `B` is constructed by conjoining these invariants.

$$
\begin{aligned}
&\mathit{encoding.B.inv.inv} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\hspace{25em} \Rightarrow Boolean\_0 \\
&B.inv.inv \equiv inv.init\_j
\end{aligned}
$$

Encoding Of Operations

First we define a constant for the operation `f` describing its core semantics specified by the pre-condition and post-condition:

$$
\begin{aligned}
&\mathit{encoding.A.f} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\hspace{15em} \Rightarrow OclAny\_0\ {'d}) \\
&\Rightarrow (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\hspace{17em} \Rightarrow Integer\_0) \\
&\quad \Rightarrow (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\hspace{18em} \Rightarrow Integer\_0) \\
&\qquad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\hspace{25em} \Rightarrow Boolean\_0 \\
&f \equiv \lambda self\ x\ result.\ pre1\ self\ x\ \wedge\ post1\ self\ x\ result
\end{aligned}
$$

For each method, we create a table storing all variants, i.e, overloadingsL

$$
\begin{aligned}
&UML\_OCL.ops.f\_Integer\_Integer.tab ::\\
&\quad {}'a\ set \rightharpoonup\\
&(state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d)\\
&\quad\quad \Rightarrow OclAny\_0\ {}'a)\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d)\\
&\quad\quad \Rightarrow Integer\_0)\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d)\\
&\quad\quad \Rightarrow Integer\_0)\\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'b\ up + {}'c)\ up + {}'d)\\
&\quad\quad \Rightarrow Boolean\_0
\end{aligned}
$$

The entry for `f` in this operation table is defined as follows:

$$
\begin{aligned}
&UML\_OCL.ops.f\_Integer\_Integer.Op ::\\
&(state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad\quad \Rightarrow OclAny\_0\ {}'d)\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad\quad \Rightarrow Integer\_0)\\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\quad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad\quad \Rightarrow Integer\_0\\
&Op \equiv \lambda self\ x.\ OCL\_choose\ (OCL\_invokeS\ arbitrary\ tab\ self\ x)
\end{aligned}
$$

and as UML_OCL.ops.f_Integer_Integer.overwrite_A:

$$
tab\ (range\ encoding.mk\_A) \equiv Some\ f \tag{C.1}
$$

Encoding the Basic UML part

The encoding of the data model on level 2 follows strictly the known schema from level 0 and 1, i.e, we also start two type tests for objects, one for testing if a object is of a specific type or subtype (i.e., this test is true, if the object is of the given kind):

$$
\begin{aligned}
&\mathit{encoding.is\_A} ::\\
&\quad(\mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}) \times\\
&\qquad \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c})\\
&\qquad \Rightarrow \mathit{OclAny\_0}\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up}))\\
&\quad\Rightarrow \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}) \times\\
&\qquad \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{Boolean\_0}\\
&\mathit{encoding.is\_A} \equiv \lambda x.\ \mathsf{if}\ \mathit{A.model.inv}\ x\ \mathsf{then}\ \mathit{level1.encoding.is\_A}\ x \mathsf{else}\ \mathsf{F} \mathsf{endif}
\end{aligned}
$$

and one for testing if a object is exactly the requested type

$$
\begin{aligned}
&\mathit{encoding.isType\_A} ::\\
&\quad(\mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}) \times\\
&\qquad \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c})\\
&\qquad \Rightarrow \mathit{OclAny\_0}\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up}))\\
&\quad\Rightarrow \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}) \times\\
&\qquad \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{Boolean\_0}\\
&\mathit{encoding.isType\_A} \equiv \lambda x.\ \mathsf{if}\ \mathit{A.model.inv}\ x\ \mathsf{then}\ \mathit{level1.encoding.isType\_A}\ x \mathsf{else}\ \mathsf{F} \mathsf{endif}
\end{aligned}
$$

Further, we define a type for universe instances:

$$
\begin{aligned}
&\mathit{encoding.is\_A\_univ} ::\\
&\quad(\mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}) \times\\
&\qquad \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c})\\
&\qquad \Rightarrow U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}))\\
&\quad\Rightarrow \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c}) \times\\
&\qquad \mathit{state}\ U\ ((\mathit{A\_key} \times \mathit{Integer\_0}) \times ((\mathit{B\_key} \times \mathit{oid}) \times {'a}\ \mathit{up} + {'b})\ \mathit{up} + {'c})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{Boolean\_0}\\
&\mathit{encoding.is\_A\_univ} \equiv\\
&\lambda x.\ \mathsf{if}\ (\mathit{A.model.inv} \circ \mathit{level1.encoding.get\_A})\ x\ \mathsf{then}\ \mathit{level1.encoding.is\_A\_univ}\ x \mathsf{else}\ \mathsf{F} \mathsf{endif}
\end{aligned}
$$

Analogous we define for the inherited class `B`:

$$
\begin{aligned}
&\mathit{encoding.is\_B} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\qquad \Rightarrow Boolean\_0 \\
&\mathit{encoding.is\_B} \equiv \lambda x.\ \mathsf{if}\ B.model.inv\ x\ \mathsf{then}\ level1.encoding.is\_B\ x \mathsf{else}\ \mathsf{Fendif}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encoding.isType\_B} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\qquad \Rightarrow Boolean\_0 \\
&\mathit{encoding.isType\_B} \equiv \lambda x.\ \mathsf{if}\ B.model.inv\ x\ \mathsf{then}\ level1.encoding.isType\_B\ x \mathsf{else}\ \mathsf{Fendif}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encoding.is\_B\_univ} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\quad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \\
&\qquad\qquad \Rightarrow Boolean\_0 \\
&\mathit{encoding.is\_B\_univ} \equiv \\
&\lambda x.\ \mathsf{if}\ (B.model.inv \circ level1.encoding.get\_B)\ x\ \mathsf{then}\ level1.encoding.is\_B\_univ\ x \mathsf{else}\ \mathsf{Fendif}
\end{aligned}
$$

We define a functions for embeding a class into the corresponding universe

$$
\begin{aligned}
&\mathit{encoding.mk\_A} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\mathit{encoding.mk\_A} \equiv \\
&\lambda x.\ \mathrm{if}\ A.model.inv\ x\ \mathrm{then}\ level1.encoding.mk\_A\ x \mathrm{else}\ level1.encoding.mk\_A\ \bot \mathrm{endif}
\end{aligned}
$$

and one for extracting an object out of a given universe:

$$
\begin{aligned}
&\mathit{encoding.get\_A} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up) \\
&\mathit{encoding.get\_A} \equiv \\
&\lambda x.\ \mathrm{if}\ (A.model.inv \circ level1.encoding.get\_A)\ x\ \mathrm{then}\ level1.encoding.get\_A\ x \mathrm{else}\ \bot \mathrm{endif}
\end{aligned}
$$

These two functions can also be understood as conversion between object types and universe types.

Analogous we define for the inherited class `B`:

$$
\begin{aligned}
&\mathit{encoding.mk\_B} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\mathit{encoding.mk\_B} \equiv \\
&\lambda x.\ \mathrm{if}\ B.model.inv\ x\ \mathrm{then}\ level1.encoding.mk\_B\ x \mathrm{else}\ level1.encoding.mk\_B\ \bot \mathrm{endif}
\end{aligned}
$$

$$
\begin{aligned}
&encoding.get\_B ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)\\
&encoding.get\_B \equiv\\
&\lambda x.\ \mathrm{if}\ (B.model.inv \circ level1.encoding.get\_B)\ x\ \mathrm{then}\ level1.encoding.get\_B\ x \mathrm{else}\ \bot \mathrm{endif}
\end{aligned}
$$

For converting along type hierarchies (i.e. down-casting and up-casting) we provide for each class two functions for the conversion to the direct predecessor and direct successor in the class hierarchy:

$$
\begin{aligned}
&encoding.A\_2\_OclAny ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&encoding.A\_2\_OclAny \equiv \lambda x.\ \mathrm{if}\ A.model.inv\ x\ \mathrm{then}\ level1.encoding.A\_2\_OclAny\ x \mathrm{else}\ \bot \mathrm{endif}
\end{aligned}
$$

$$
\begin{aligned}
&encoding.OclAny\_2\_A ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)\\
&encoding.OclAny\_2\_A \equiv \lambda x.\ \mathrm{if}\ \top\ \mathrm{then}\ level1.encoding.OclAny\_2\_A\ x \mathrm{else}\ \bot \mathrm{endif}
\end{aligned}
$$

Analogous we define for the inherited class `B`:

$$
\begin{aligned}
&\mathit{encoding}.B\_2\_A :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)) \\
&\Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up) \\
&\mathit{encoding}.B\_2\_A \equiv \lambda x.\ \textsf{if}\ B.model.inv\ x\ \textsf{then}\ level1.encoding.B\_2\_A\ x \textsf{else}\ \bot \textsf{endif}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encoding}.A\_2\_B :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)) \\
&\Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up) \\
&\mathit{encoding}.A\_2\_B \equiv \lambda x.\ \textsf{if}\ A.model.inv\ x\ \textsf{then}\ level1.encoding.A\_2\_B\ x \textsf{else}\ \bot \textsf{endif}
\end{aligned}
$$

Finally, we provide accesor functions for the class attributes:

$$
\begin{aligned}
&\mathit{encoding}.A.i :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up)) \\
&\Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\qquad \Rightarrow Integer\_0 \\
&\mathit{encoding}.A.i \equiv \lambda x.\ \textsf{if}\ A.model.inv\ x\ \textsf{then}\ level1.encoding.A.i\ x \textsf{else}\ \bot \textsf{endif}
\end{aligned}
$$

$$
\begin{aligned}
&encoding.B.j ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up)\ up))\\
&\Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up)\ up)\\
&encoding.B.j \equiv \lambda x.\ \text{if}\ B.model.inv\ x\ \text{then}\ level1.encoding.B.j\ x \text{else}\ \bot \text{endif}
\end{aligned}
$$

Defining `oclAsType` and `allInstances`

The constants for `oclAsType` and `allInstances` must be defined for each class separately (using overloading). We provide for each class two definitions for `oclAsType`: one for the actual state ("UML_OCL.univ.encoding.A.OclAllInstancesencoding_A")

$$OclAllInstances\ ?self \equiv \lambda(s,\ s').\ Abs\_Set\_0\ \llcorner level0.encoding.get\_A\ `\ ran\ s' \lrcorner$$

and one for the previous state ("UML_OCL.univ.encoding.A.OclAllInstancesencoding_AAtpre"):

$$OclAllInstancesAtpre\ ?self \equiv \lambda(s,\ s').\ Abs\_Set\_0\ \llcorner level0.encoding.get\_A\ `\ ran\ s \lrcorner$$

Analgous for class `B` we define "UML_OCL.univ.encoding.B.OclAllInstancesencoding_B":

$$OclAllInstances\ ?self \equiv \lambda(s,\ s').\ Abs\_Set\_0\ \llcorner level0.encoding.get\_B\ `\ ran\ s' \lrcorner$$

and "UML_OCL.univ.encoding.B.OclAllInstancesencoding_BAtpre":

$$OclAllInstancesAtpre\ ?self \equiv \lambda(s,\ s').\ Abs\_Set\_0\ \llcorner level0.encoding.get\_B\ `\ ran\ s \lrcorner$$

In a similar way, we have to overload the definition of `oclAsType`, based on the already defined conversion for adjacent classes. For class `A` the following cases must be considered along the inheritance hierarchy: The downcast from `OclAny` "UML_OCL.univ.encoding.A.from_OclAny":

$$OclAsType \equiv \lambda self\ cSet.\ level1.encoding.OclAny\_2\_A\ self$$

the upcast to `OclAny` "UML_OCL.univ.encoding.A.to_OclAny":

$$OclAsType \equiv \lambda self\ cSet.\ level1.encoding.A\_2\_OclAny\ self$$

and the identity "UML_OCL.univ.encoding.A.to_A":

$$OclAsType \equiv \lambda self\ cSet.\ self$$

For class B more cases have to be considered, i.e., the downcast from A "UML_OCL.univ.encoding.B.fro

$$OclAsType \equiv \lambda self\ cSet.\ (level1.encoding.A\_2\_B \circ level1.encoding.OclAny\_2\_A)\ self$$

the downcast from `OclAny` "UML_OCL.univ.encoding.B.from_OclAny":

$$OclAsType \equiv \lambda self\ cSet.\ level1.encoding.A\_2\_B\ self$$

the upcast to `OclAny` "UML_OCL.univ.encoding.B.to_OclAny":

$$OclAsType \equiv \lambda self\ cSet.\ (level1.encoding.A\_2\_OclAny \circ level1.encoding.B\_2\_A)\ self$$

the upcast to A "UML_OCL.univ.encoding.B.to_A":

$$OclAsType \equiv \lambda self\ cSet.\ level1.encoding.B\_2\_A\ self$$

and the identity "UML_OCL.univ.encoding.B.to_B":

$$OclAsType \equiv \lambda self\ cSet.\ self$$

Optional, the encoder generates all definitions that are orthogonal to the class hierarchy. On the one hand, they are defined to be `OclUndefined` according to the OCL standard, on the other hand, they violate a well-formed constraint.

Encoding of OCL Invariants: Part Two

Finally, we generate the the final invariant, e.g., for class A we generate:

$$\begin{aligned}
&encoding.A.inv.init_i ::\\
&\quad (state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny_0\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad\quad state\ U\ ((A_key \times Integer_0) \times ((B_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad\qquad \Rightarrow Boolean_0\\
&inv.init_i \equiv \lambda self.\ OclIsNew\ self \longrightarrow encoding.A.i\ self \triangleq 42
\end{aligned}$$

and the second one describes the user defined invariant:

$$
\begin{aligned}
&encoding.A.inv.inv\_1 ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\hfill \Rightarrow Boolean\_0\\
&inv.inv\_1 \equiv \lambda self.\ encoding.A.i\ self \geq 0
\end{aligned}
$$

The overall invariant for class `A` is constructed by conjoining these invariants.

$$
\begin{aligned}
&encoding.A.inv.inv ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\hfill \Rightarrow Boolean\_0\\
&A.inv.inv \equiv \lambda obj.\ inv.init\_i\ obj\ \wedge inv.inv\_1\ obj
\end{aligned}
$$

Analogous we are encoding the pre-condition and post-condition of the operation `f`:

$$
\begin{aligned}
&encoding.A.f\_Integer\_Integer.pre1.pre\_0 ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\hfill \Rightarrow Boolean\_0\\
&pre\_0 \equiv \lambda self\ x.\ x > 0
\end{aligned}
$$

$$
\begin{aligned}
&encoding.A.f\_Integer\_Integer.post1.post\_0 ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up))\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\qquad \Rightarrow Integer\_0)\\
&\qquad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\qquad \Rightarrow Boolean\_0
\end{aligned}
$$

$$post\_0 \equiv \lambda self\ x\ result.\ result \triangleq x\ {-}{>}div(\ level1.encoding.A.i\ self\ )$$

$$
\begin{aligned}
&encoding.A.f\_Integer\_Integer.pre1 ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up))\\
&\Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times 'a\ up + 'b)\ up + 'c)\\
&\qquad\qquad \Rightarrow Boolean\_0
\end{aligned}
$$

$$pre1 \equiv pre\_0$$

$$
\begin{aligned}
&encoding.A.f\_Integer\_Integer.post1 ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up))\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Integer\_0)\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Boolean\_0\\
&post1 \equiv post\_0
\end{aligned}
$$

For class `B` no initial values or invariants are given, thus we generate invariants that are just `true`:

$$
\begin{aligned}
&encoding.B.inv.init\_j ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Boolean\_0\\
&inv.init\_j \equiv \lambda self.\ \mathsf{T}
\end{aligned}
$$

$$
\begin{aligned}
&encoding.B.inv.init\_j ::\\
&\quad (state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up)\ up))\\
&\quad \Rightarrow state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c}) \times\\
&\qquad state\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {'a}\ up + {'b})\ up + {'c})\\
&\qquad \Rightarrow Boolean\_0\\
&inv.init\_j \equiv \lambda self.\ \mathsf{T}
\end{aligned}
$$

The overall invariant for class `B` is constructed by conjoining these invariants.

$$
\begin{aligned}
&\mathit{encoding.B.inv.inv} :: \\
&\quad (\mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\qquad\quad \Rightarrow OclAny\_0\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up)\ up)) \\
&\quad \Rightarrow \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \times \\
&\qquad\quad \mathit{state}\ U\ ((A\_key \times Integer\_0) \times ((B\_key \times oid) \times {}'a\ up + {}'b)\ up + {}'c) \\
&\hspace{22em} \Rightarrow Boolean\_0
\end{aligned}
$$

$$
\mathit{B.inv.inv} \equiv \mathit{inv.init\_j}
$$

Appendix D.

The GNU General Public Licence

The following is the text of the GNU General Public Licence, under the terms of which this software is distributed.

D.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

D.2. Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

(a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you

(whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision

will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. **Because the Program is licensed free of charge, there is no warranty for the Program, to the extent permitted by applicable law. except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you. Should the Program prove defective, you assume the cost of all necessary servicing, repair or correction.**

12. **In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.**

END OF TERMS AND CONDITIONS

D.3. Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
```

```
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Part V.

Bibliography and Index

Bibliography

[1] Dresden OCL Toolkit. URL `http://dresden-ocl.sourceforge.net/`. (Cited on page 23.)

[2] fxp: The Functional XML Parser. URL `http://atseidl2.informatik.tu-muenchen.de/~berlea/Fxp/`. (Cited on page 16.)

[3] Isabelle. URL `http://isabelle.in.tum.de`. (Cited on pages 12 and 91.)

[4] Poly/ML. URL `http://www.polyml.org/`. (Cited on page 107.)

[5] Proof General. URL `http://proofgeneral.inf.ed.ac.uk`. (Cited on pages 13, 97, and 107.)

[6] SML of New Jersey. URL `http://www.smlnj.org/`. (Cited on pages 107 and 108.)

[7] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005. (Cited on page 135.)

[8] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Computer Science and Applied Mathematics. Academic Press, Orlando, May 1986. ISBN 0120585367. URL `http://gtps.math.cmu.edu/andrews.html`. (Cited on pages 13 and 17.)

[9] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland/Elsevier, Nijmegen, The Netherlands, June 1992. (Cited on page 27.)

[10] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. An MDA framework supporting OCL, 2006. Submitted to OclApps 2006: OCL for (Meta-)Models in Multiple Application Domains. (Cited on page 97.)

[11] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003. ISSN 0948-6968. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-z-2003`. (Cited on pages 31, 32, 77, and 139.)

[12] Achim D. Brucker and Burkhart Wolff. A note on design decisions of a formalization of the OCL. Technical Report 168, Albert-Ludwigs-Universität Freiburg, January 2002. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-note-2002`. (Cited on page 22.)

[13] Achim D. Brucker and Burkhart Wolff. Using theory morphisms for implementing formal methods tools. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proof and Programs*, number 2646 in Lecture Notes in Computer Science, pages 59–77. Springer-Verlag, Nijmegen, 2003. ISBN 3-540-14031-X. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-embedding-2003`. (Cited on pages 173 and 404.)

[14] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. (Cited on page 17.)

[15] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8. (Cited on page 29.)

[16] Dov M. Gabbay. *Labelled Deductive Systems*. Number 33 in Oxford Logic Guides. Oxford University Press, 1997. (Cited on pages 78 and 90.)

[17] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge Press, July 1993, 472 pages. (Cited on page 18.)

[18] Reiner Hähnle. Towards an efficient tableau proof procedure for multiple-valued logics. In Egon Börger, Hans Kleine Büning, Michael M. Richter, and Wolfgang Schönfeld, editors, *Selected Papers from Computer Science Logic, CSL'90, Heidelberg*, volume 533 of *Lecture Notes in Computer Science*, pages 248–260. Springer-Verlag, 1991. (Cited on pages 178 and 192.)

[19] Reiner Hähnle. Efficient deduction in many-valued logics. In *Proc. International Symposium on Multiple-Valued Logics, ISMVL'94, Boston/MA, USA*, pages 240–249. IEEE CS Press, Los Alamitos, 1994. (Cited on pages 80 and 90.)

[20] Reiner Hähnle. Tableaux for many-valued logics. In Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 529–580. Kluwer, Dordrecht, 1999. (Cited on pages 77 and 90.)

[21] A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML98*. Springer-Verlag, Heidelberg, June 1998. URL `http://www.cs.ukc.ac.uk/pubs/1998/788`. (Cited on page 23.)

[22] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, pages 64–72, October 2004. (Cited on page 33.)

[23] Rolf Hennicker, Heinrich Hussmann, and Michel Bidoit. On the precise meaning of OCL constraints. In T. Clark and J.Warmer, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 69–84.

Springer-Verlag, 2002. URL `http://www.lsv.ens-cachan.fr/Publis/PAPERS/HBB-oclBook.ps`. (Cited on page 178.)

[24] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978. (Cited on page 27.)

[25] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, July 1999. (Cited on page 264.)

[26] Formal Specification – Z Notation – Syntax, Type and Semantics. January 2002. URL `http://www.cs.york.ac.uk/~ian/zstan/`. Draft International Standard. (Cited on pages 11 and 139.)

[27] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. URL `ftp://ftp.ncl.ac.uk/pub/users/ncbj/ssdvdm.ps.gz`. 0-13-880733-7. (Cited on pages 11 and 29.)

[28] Manfred Kerber and Michael Kohlhase. A mechanization of strong Kleene logic for partial functions. In Alan Bundy, editor, *Automated Deduction — CADE-12*, Proceedings of the 12th International Conference on Automated Deduction, pages 371–385. Springer Verlag, Berlin, Germany, Nancy, France, 1994. URL `http://www.cs.cmu.edu/~kohlhase/papers/KeKo94-CADE.ps`. LNAI 814. (Cited on pages 77 and 389.)

[29] Manfred Kerber and Michael Kohlhase. A tableau calculus for partial functions. *Collegium Logicum – Annals of the Kurt-Gödel-Society*, **2**:21–49, 1996. URL `www.cs.cmu.edu/~kohlhase/papers/kgs95.ps`. (Cited on page 90.)

[30] Setrag N. Khoshafian and George P. Copeland. Object identity. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 406–416. ACM Press, New York, NY, USA, 1986. ISBN 0-89791-204-7. (Cited on page 70.)

[31] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Univ. Press, 1986. ISBN 9780521356534. (Cited on page 28.)

[32] Luis Mandel and Marìa Victoria Cengarle. On the expressive power of OCL. *World Congress on Formal Methods (FM'99, Proceedings),*, September 1999. URL `http://www.pst.informatik.uni-muenchen.de/personen/cengarle/ocl.ps`. (Cited on page 23.)

[33] Thomas F. Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedigns of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357. IEEE Computer Society Press, 1992. ISBN 0-8186-2460-4. (Cited on page 101.)

[34] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353. (Cited on page 29.)

[35] P. D. Mosses. *Denotational Semantics*, chapter 11. Elsevier, Amsterdam, first edition, 1990. ISBN 0-444-88075-5. (Cited on page 137.)

[36] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999. URL `http://www4.informatik.tu-muenchen.de/~nipkow/pubs/jfp99.html`. (Cited on page 45.)

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. URL `http://www4.in.tum.de/~nipkow/LNCS2283/`. (Cited on page 19.)

[38] Object constraint language specification. September 1997. URL `http://www.rational.com/media/uml/resources/media/ad970808_UML11_OCL.pdf`. Covers OCL 1.1. (Cited on page 23.)

[39] OMG Unified Modeling Language Specification. June 1999. URL `ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf`. Covers UML/OCL 1.3. (Cited on pages 23, 138, and 139.)

[40] OMG Unified Modeling Language Specification. March 2003. URL `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf`. Covers UML/OCL 1.5. (Cited on pages 12, 22, 23, 29, 35, 47, 60, 101, 102, and 113.)

[41] UML 2.0 OCL specification. October 2003. URL `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.pdf`. (Cited on pages 12, 22, 23, 29, 33, 34, 37, 39, 40, 41, 42, 43, 47, 49, 51, 52, 53, 64, 68, 69, 74, 77, 101, 102, 113, 114, 129, 135, 137, 138, 171, 175, 176, 177, 200, 404, 409, and 431.)

[42] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 187–211. MIT Press, 2000. URL `http://www.cl.cam.ac.uk/users/lcp/papers/Sets/milner-ind-defs.pdf`. (Cited on page 63.)

[43] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI 1988*, pages 199–208. 1988. (Cited on page 27.)

[44] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Ph.D. thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002. (Cited on page 33.)

[45] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998. (Cited on page 29.)

[46] Graeme Smith. *The Object Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000. ISBN 0-7923-8684-1, 160 pages. (Cited on page 29.)

[47] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, second edition, 1992. ISBN 013-978529-9. URL `http://spivey.oriel.ox.ac.uk/~mike/zrm/`. (Cited on page 29.)

[48] Victoria Stavridou, Thomas F. Melham, and R. T. Boute, editors. *Experience with embedding hardware description languages in HOL*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*. North-Holland, Nijmegen, The Netherlands, 1993. URL `http://www.cl.cam.ac.uk/users/jrh/papers/EmbeddingPaper.html`. (Cited on page 51.)

[49] H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997. URL `http://www.informatik.uni-freiburg.de/~wolff/papers/1997/CSP.ps.gz`. (Cited on pages 31 and 32.)

[50] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the object constraint language of UML, December 1999. Response to Object Management Group's Request for Information on UML 2.0. (Cited on page 23.)

[51] Luca Viganò. *Labelled Non-Classical Logics*. Kluwer Academic Publishers, Dordrecht, 2000. (Cited on pages 78 and 90.)

[52] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman, Inc., Reading, MA, USA, second edition, August 2003. ISBN 0-321-17936-6. URL `http://www.klasse.nl/ocl-boek/intro.htm`. (Cited on pages 102 and 129.)

[53] Markus Wenzel. *The Isabelle/Isar Reference Manual*. TU München, München, 2005. URL `http://isabelle.in.tum.de/dist/Isabelle2005/doc/isar-ref.pdf`. (Cited on pages 117, 120, and 121.)

[54] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993. ISBN 0-262-23169-7, 384 pages. (Cited on pages 137 and 139.)

List of Acronyms

ACI Associativity, commutativity, and idempotency

CASE Computer Aided Software Engineering

CCS A Calculus of Communicating Systems

CTL Computational Tree Logic

CSP Communicating Sequential Processes

DNF disjunctive normal form

ETH Swiss Federal Institute of Technolgy

EBNF Extended Backus-Naur Form

FXP Functional XML Parser

GNU GNU is not Unix

GUI Graphical User Interface

HOAS higher-order abstract syntax

HOL higher-order logic

Isar Intelligible semi-automated reasoning

LCF Logic for Computable Functions

LE Local (Formula) Equivalence

LJE Local Judgement Equivalence

LTL Linear Temporal Logic

LTC Local Judgement Tableaux Calculus

OCL Object Constraint Language

OMG Object Management Group

SKL Strong Kleene Logic

SML Standard Meta Language

sml/NJ SML of New Jersey

su4sml SecureUML for SML

UC Universal (Formula) Congruence

UJE Universal Judgement Equivalence

UML Unified Modelling Language

VDM Vienna Definition Method

XMI XML Metadata Interchange

XML Extensible Markup Language

ZF Zermelo-Fränkel

ZFC Zermelo-Fränkel set theory with the acxiom of choice.

List of Glitches

List of Extensions

List of Figures

List of Tables

List of Definitions

Listings

Index