Doctoral Thesis ETH No. 17003

Type Safety of C$^\sharp$ and .NET CLR

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Nicu Georgian Fruja
Dipl. Math.–University of Bucharest
born January 13, 1977
citizen of Romania

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Robert Stärk, co-examiner

2007

Abstract

Type safety plays a crucial role in the security enforcement of any typed programming language. This thesis presents a formal proof of C$^\sharp$'s type safety. For this purpose, we develop an abstract framework for C$^\sharp$, comprising formal specifications of the language's grammar, of the statically correct programs, and of the static and operational semantics. Using this framework, we prove that C$^\sharp$ is type-safe, by showing that the execution of statically correct C$^\sharp$ programs does not lead to type errors.

The bytecode resulting from compiling C$^\sharp$ programs is executed on the Common Language Runtime (CLR), the managed execution environment of the .NET Framework. As the bytecode may be transmitted over a network and get corrupted as a result, the bytecode execution may perform malicious operations. Therefore, to guarantee type safety, the CLR employs a bytecode verification, *i.e.*, a static analysis that performs several consistency checks before the bytecode is run. We show type safety of the bytecode language by proving the soundness of the bytecode verification. The abstract framework developed for the proof encompasses formal specifications of the bytecode language's abstract syntax, of the well-typedness constraints ensured by the bytecode verification, and of the bytecode language's static and dynamic semantics. Finally, we demonstrate that legal and well-typed bytecode programs do not lead to any type error when executed.

This thesis reveals an important number of relevant ambiguities in the official specifications of C$^\sharp$ and CLR and, in certain cases, even the absence of a specification at all. Thus, we identify and fill several gaps in the two official documents. We also point out a series of inconsistencies between different implementations of C$^\sharp$ and CLR and their official specifications.

Zusammenfassung

Typsicherheit nimmt einen entscheidenden Platz in der Gewährleistung der Sicherheit einer typisierten Programmiersprache ein. Diese Doktorarbeit bietet einen formalen Beweis der Typsicherheit von $C^\sharp$. Hierfür erarbeiten wir ein abstraktes System für $C^\sharp$, bestehend aus den formalen Spezifikationen der Grammatik der Sprache, der statischen Korrektheit von Programmen und der statischen und operativen Semantik. Basierend auf diesem System beweisen wir die Typsicherheit von $C^\sharp$, indem wir aufzeigen, dass die Ausführung eines statisch korrekten $C^\sharp$ Programms nicht zu Typfehlern führt.

Das kompilierte Ergebnis eines $C^\sharp$ Programms wird als Bytecode auf einer verwalteten Umgebung namens Common Language Runtime (CLR) ausgeführt. Da Bytecode über ein Netzwerk übertragen werden und folglich korrumpiert werden kann, kann die Ausführung von Bytecode zu böswilligen Operationen führen. Um die Typsicherheit zu garantieren, setzt CLR folglich eine Bytecode-Verifikation ein, das heisst eine statische Analyse, welche mehrere Konsistenzüberprüfungen vornimmt bevor der Bytecode ausgeführt wird. Wir zeigen, dass der Bytecode typsicher ist, indem wir die Korrektheit der Bytecode-Verifikation beweisen. Das abstrakte System, welches wir für den Beweis entwickelt haben, umfasst die Spezifikationen der abstrakten Syntax der Bytecode-Sprache, der Wohl-Typisiertheits-Bedingungen, welche durch die Bytecode-Verifikation sichergestellt werden müssen, und der statischen und dynamischen Semantik der Bytecode-Sprache. Abschliessend legen wir dar, dass legale, wohl-typisierte Bytecode Programme während ihrer Ausführung zu keinerlei Typffehlern führen.

Diese Doktorarbeit zeigt eine bedeutende Anzahl von relevanten Zweideutigkeiten in der offiziellen Spezifikationen von $C^\sharp$ und CLR auf und in gewissen Fällen sogar das Fehlen einer Spezifikation überhaupt. In diesem Sinne identifizieren wir und schliessen wir einige Lücken in den zwei offiziellen Dokumenten. Des weiteren zeigen wir mehrere Inkonsistenzen zwischen verschiedenen Implementationen von $C^\sharp$ und CLR und deren offiziellen Spezifikationen auf.

Acknowledgments

This thesis would have not been possible without the support of many persons. First of all, I am greatly indebted to my first advisor, Prof. Dr. Robert F. Stärk, not only for providing the challenging and exciting topic of this thesis, but also for his advice and support. I want to express my profound acknowledgement to Prof. Dr. Thomas R. Gross who supervised and stimulated my work during my last two years. My gratitude to both supervisors cannot be realistically expressed and extended in any form of words.

It is a pleasure to thank Prof. Dr. Egon Börger for his advice, valuable criticism, invaluable time, and for sharing his profound knowledge in formal methods. Special thanks go also to Nigel Perry, Peter Sestoft and Jon Jagger (members of the C$^\sharp$ Standardization Committee) and to the Microsoft Corporation employees Jonathan Keljo, Joe Duffy, Chris Brumme, Don Syme and Matt Grice for providing invaluable insights into the design of C$^\sharp$ and CLR.

I thank the students I supervised during their master and diploma thesis for their contribution to the validation of the formal models in this thesis: Christian Marrocco, Horatiu Jula, and Markus Frauenfelder. For a successful thesis, not only the scientific part was important but also the ambient in the group. I thank my long-term group mate Stanislas Nanchen and all members of the LST group for the nice atmosphere.

I want to express my sincere thanks to my mother and father, Dobrita and Marin Fruja, for their unconditional support, absolute confidence in me, and for giving me all the opportunities in the world to explore my potentials and pursue my dreams. Last but not least, Karin has always been at my side and offered me strong support - I am deeply grateful for that and for her infinite patience that accompanied me along this long journey.

Contents

1 Introduction

Well-typed programs never go wrong.
Robin Milner

1.1 Motivation

Type safety is a central theme in *language-based security*. Language type safety is relevant to the security of a system since many attacks exploit vulnerabilities in language's type system, design and implementation, circumventing system security by hijacking the program execution model. Type safety is a formal guarantee that the execution of any program is free of certain forms of erroneous or undesirable program behavior, called *type errors*. The behaviors classified as type errors by any given programming language are generally those that result from attempts to perform on some value an operation that is not appropriate to its type: For example, a method defined on integers is applied to an object.

The enforcement of type safety can be *static*, catching potential errors at compile-time, or *dynamic*, associating type information with values at run-time and consulting them as needed to detect imminent errors, or a *combination* of both approaches. Employing a conservative strong static type analysis and dynamic checks, *e.g.*, cast checking, is fundamental to modern programming practice and is used in programming languages, like C$\sharp$ [74], Java [67], and OCaml [12]. This thesis studies C$\sharp$'s type safety, primarily since, at the time of this writing, C$\sharp$ is, besides Java, one of the most *popular modern general-purpose* programming languages.

C$\sharp$ is a class-based single-inheritance object-oriented programming language. It has been developed by Microsoft Corporation specifically for the *Common Language Runtime* (CLR) of the *.NET Framework* [2, 3], a managed execution environment with *Common Intermediate Language* (CIL), often referred to as CLR bytecode language. Approved as a *standard* by ECMA International [44], C$\sharp$ has a procedural, object-oriented syntax based on C++ that includes several novel features, but also features of other programming languages, most notably of Java, Visual Basic, and Delphi. Of particular interest are features like *struct types* as user defined value types, *call-by-reference* parameter passing for methods, *unified type system* with *boxing* and *unboxing*, and *delegates* – methods as values, also known as *closures* – for invoking one or multiple methods. It must be emphasized that C$\sharp$ also provides the ability to write *unsafe code* with C-style pointers, but only in clearly marked *unsafe contexts*. For obvious reasons related to their unsafe nature, the C$\sharp$ language analyzed in this thesis does not include unsafe features.

The C$^\sharp$ language is intended for writing applications for both hosted and embedded systems, ranging from the very large ones that use elaborated operating systems, down to the very small ones having dedicated functions. It is aimed to be used, in particular, for developing desktop and internet applications and software components suitable for deployment in distributed environments. Given the importance of security for these kinds of applications, an investigation of C$^\sharp$'s type safety, a key ingredient for security, is justified and necessary.

C$^\sharp$'s type safety alone is, however, *not* sufficient to guarantee type-safe execution of bytecode programs on the CLR. The reason for this is that the bytecode might be produced by an *illegal* (buggy) C$^\sharp$ compiler or might not even be related to a C$^\sharp$ program. Therefore, this thesis also analyzes type safety of the bytecode language executed on the CLR.

The CLR, approved as a *standard* by ECMA International [45], has been designed from the ground up as a target for a wide variety of languages, called *.NET compliant languages*. A non-comprehensive list of programming languages compiling to CIL ranges from object-oriented languages, like C$^\sharp$, Visual Basic, Managed C++, Eiffel [46], Smalltalk, Spec$^\sharp$, and Active Oberon, to functional programming languages, like Haskell [14], Scheme [43], Mercury [11], and AsmL [49], and from pure procedural programming languages, like COBOL and Modula 2, to scripting languages, like JScript, Perl, and Phyton. At the time of this writing, there are 53 .NET compliant languages (see [17] for an up-to-date list), out of which C$^\sharp$ is the most popular. The CLR makes possible the *cross-language compatibility*, *i.e.*, the .NET components can interact with each other regardless of the languages they are written in. Due to the wide spectrum of .NET compliant languages, the CLR is required to support various features, such as, for example, *tail method calls*, *pointers* (including *method pointers*), *typed references*, *generics* with *F-bounded polymorphism*, *covariance*, and *contravariance*, a *structured exception handling model*, and a *unified type system* with *boxing* and *unboxing*.

As programs have become more dynamic and distributed, operating systems and browsers have been required more often to execute downloaded or mobile code. This execution can be problematic as there exists malicious programs, intended to cause mischief when executed on unsuspecting hosts. As the CLR has been designed to be suitable for the development of a variety of applications, ranging from web services to web and Windows applications, CIL's type safety, central to the notion of safe code execution within the CLR, is crucial. Executing non type-safe bytecode programs on the CLR, regardless whether the bytecode was accidentally or maliciously created, can produce erroneous or destructive behavior within the execution system. It is worth mentioning that CLR's type safety also includes checks other than just type constraints, such as, for example, prohibition of *stack underflow / overflow*, correct use of the exception handling facilities, and *object initialization*.

To ensure type safety, the CLR employs a conservative static analysis, called *bytecode verification*, that performs several *consistency checks* before the bytecode is executed. Bytecode verification rejects, for example, a bytecode program that performs operations on values of wrong types, accesses an uninitialized object, calls a method with an incorrect number of parameters, calls a method with a parameter of an incorrect type or executes a method which returns a value of an incorrect type. The checks are not only limited to type checks: Bytecode verification prohibits bytecode programs that jump to an invalid code index. Similarly to the C$^\sharp$'s type analysis, the bytecode verification is *conservative*, meaning that not only all the faulty programs are rejected, but also programs that would never exhibit a run-time type error. The reasons this conservative analysis are the *undecidability of the halting problem* and efficiency considerations. It must be emphasized that, analogously to the C$^\sharp$'s type analysis, the CLR's bytecode verification by itself does not guarantee type-safe execution of the bytecode: Many crucial properties of the code still need to be ensured dynamically, *e.g.*, array bound checks and null pointer checks. The purpose of bytecode verification is to shift as many dynamic checks as possible from run-time to verification-time.

For the sake of clarity, this thesis only analyzes type safety of large, yet representative, subsets of C$^\sharp$ and CIL. Correspondingly, two formal proofs are developed, basically showing the *soundness* of the C$^\sharp$'s type analysis and CLR's bytecode verification defined accordingly to [74] and [45], respectively. For that, we formally define the *abstract syntax*, *type system*, *type constraints* guaranteed by the C$^\sharp$'s type analysis and by the CLR's bytecode verification, and the *static* and *dynamic semantics* of C$^\sharp$ and CLR. Moreover, we provide a formal specification of the *definite assignment rules* that must be satisfied by a C$^\sharp$ method in order to be accepted by the *definite assignment analysis*, *i.e.*, a dataflow analysis performed by the C$^\sharp$ compiler to guarantee that all local variables are assigned to before their value is used. The two type safety proofs basically establish that the run-time execution, according to the semantics models, of legal C$^\sharp$ and CIL programs satisfying the type constraints and, in case of C$^\sharp$, the definite assignment rules does not lead to any type error. The proved type safety properties attest that C$^\sharp$'s and CIL's type safety are strictly stronger than the *memory safety* property. In particular, this means that memory is only ever accessed in a well-known and controlled manner through typed references.

1.2 Thesis Statement and Contributions

In this thesis, we establish the following results:

- *A large and representative subset of* C$^\sharp$ *is type-safe.*
- *A large and representative subset of CIL is type-safe.*

Besides the formal proofs of the two statements, this thesis makes several other contributions:

- Two important by-products of this thesis are represented by the formal models developed for CIL and its bytecode verification. The CIL model, including a fairly elaborate model for the CLR exception handling mechanism, can serve, for example, as the basis for building a robust interpreter for the CIL bytecode [92], whereas the bytecode verification model can assist the implementation of a prototype of a CLR bytecode verifier. Note that the C$^\sharp$ model, on which the C$^\sharp$'s type safety proof relies, has been defined in [24] and does *not* constitute a contribution of this thesis.
- This thesis reveals an important number of relevant ambiguities in the official specifications of C$^\sharp$ [74] and CLR [45] and, in certain cases, even the absence of a specification at all. Thus, we identify and fill several gaps in the two official documents. Annotations corresponding to some of these gaps have been included in the C$^\sharp$ *Annotated Standard* [82], a book for which this thesis' author was an invited guest annotator. We also point out a series of inconsistencies between different implementations of C$^\sharp$ and CLR and the standards [74] and [45], respectively. Moreover, many explanations presented in this thesis,

alongside the specifications [74] and [45], provide the rationale behind the design, that is often missing from the official specifications.

- Last but not the least, the abstract frameworks defined in this thesis serve not only as a basis for the type safety proofs, but also as starting point for reasoning about a wide variety of aspects of C$^\sharp$ and CLR. Thus, one can investigate, for example, stronger safety guarantees for C$^\sharp$ and CIL programs, or ways in which programs can be optimized in a type-safe manner.

1.3 Thesis Road Map

The semantic models of C$^\sharp$ and CLR, serving as the rigorous basis for the type safety proofs, are defined in terms of abstract interpreters, using the framework of *Abstract State Machines* (ASMs). To allow the reader a correct understanding of the formal specifications and proofs in Chapters 3 and 4, Chapter 2 includes a brief overview as well as a mathematical definition of the ASMs, based on the definition presented in [115]. The formal specifications and the corresponding type safety proofs of C$^\sharp$ and CIL are presented in Chapter 3 and 4. The two chapters may be read independently of each other, but each of them presupposes Chapter 2. Chapter 5 concludes. For the reader's convenience, the evaluation rules of the semantics model for C$^\sharp$ defined in [24] are included in the Appendix.

2
The ASM Method

So-called "natural language" is wonderful for the purposes it was created for, such as to be rude in, to tell jokes in, to cheat or to make love in, but it is hopelessly inadequate when we have to deal unambiguously with situations of great intricacy, situations which unavoidably arise in such activities as legislation, arbitration, mathematics, or programming.

Edsgar W. Dijkstra

The *Abstract State Machines* (ASMs) [70, 27] are a mathematical formalism, widely used for high-level design and analysis of computing systems, in particular for the formal definition of programming language semantics. The notion of ASMs captures precisely some basic operational intuitions of computing. The ASMs describe a system by means of a particular syntax and associated semantics and use classical, well-understood, mathematical structures (algebras) to describe the states of a computation. This distinguishes ASMs from informal methodologies. The ASM programs use a very simple and expressive syntax, whose semantics can be understood without any further explanation, simply viewing the ASM programs as "*pseudo-code over abstract data structures*". In contrast, other specification methods, notably *denotational semantics*, use complicated syntax whose semantics is more difficult to read and write. This is one of the reasons we used the ASM method for the formal specifications in Chapters 3 and 4. Despite of their abstract nature, the ASM specifications are executable in the .NET ASM Language (AsmL) [49]. Thereby, a way to validate and verify the ASM specifications is provided.

ASMs in a nutshell In the ASM framework, the systems are formalized in a state-based way, by means of *states* and *state transitions*. An ASM state is represented by an algebra over a given *vocabulary*, where data come as abstract objects, that is, as elements of *universes* (one for each category of data) which are equipped with basic operations (*functions*) and predicates. The state transitions are defined by transition rules which specify the possible state changes in terms of *updates*. The notion of ASM *run* is the well-known notion of computation of transition systems. In a given state, an ASM computation step consists in executing *simultaneously* all updates prescribed by the *applicable* transition rules, *i.e.*, the rules whose *guard* is true in the given state, if there no clashing updates. Simultaneous execution is a convenient way to make use of synchronous parallelism and to abstract from irrelevant sequentiality.

ASMs defining language semantics The semantics of programming languages is described operationally using ASMs based on the abstract syntax trees. Part of the current state is the current task, a pointer to the currently to be executed node in the abstract syntax tree. During program execution, states are transformed into new states, by that also updating the pointer to the current task. During a state transition, the interpretation of some function symbols may change. To specify the semantics of a language, the abstract syntax tree is assumed to contain attributes defining all continuations, especially for the non-compositional changes of the control flow. The ASM definition models the program counter during program execution, by that using the continuation attributes which might be split up according to the truth value of guards. The semantics of each program node is described by a finite number of transition rules. Typically, the guard of such a transition rule specifies the nodes in the abstract syntax tree for which the transition rule is applicable. The transition rules define updates, thereby employing child nodes as well as statically computed continuations.

2.1 A Mathematical Definition of ASMs

In this section, we present a mathematical definition for the syntax and semantics of ASMs. An interested reader can find two excellent formal definitions of ASMs in [115, §2] and [27, §2.4]. The definition included here follows very closely the definition in [115, §2].

2.1.1 States

The abstract states of the ASMs are given as *algebras*. An algebra is defined with respect to a *vocabulary* (sometimes called *signature*), *variables*, and *terms*. Additionally, an *interpretation* and a *variable assignment* are considered to allow evaluation of syntactical entities to semantical values.

Definition 2.1.1 (Vocabulary) *A* vocabulary Σ *is given by a finite set of function names, each with a fixed* arity. *Every vocabulary should contain the nullary functions True, False, and undef.*

Functions Classification As the states are mostly characterized in terms of functions, one distinguishes different kinds of functions:

- *Static* functions never change during any run. The nullary functions *True*, *False* and *undef* are considered static functions.
- *Dynamic* functions may change their interpretation during a run.
- *External* functions, also known as *oracles*, whose interpretation cannot be changed by the system itself but by the outer environment.
- *Derived* functions are defined in terms of other functions, known as *basic* functions.
- *Relational* functions (also known as *characteristic functions* or *predicates*), with the boolean universe as codomain.

The derived functions can be viewed as *macros*, applied to shorten expressions. The relational functions are typically used to describe sets. The nullary function names of a vocabulary are called *constants*, though, as we have seen above, the interpretation of dynamic nullary functions can change from one state to the next state.

Definition 2.1.2 (State) *Given a vocabulary Σ, a* state *$\mathcal{A}$ for Σ is given by a non-empty set X and the* interpretations *of the function names of Σ:*

- *the interpretation $c^{\mathcal{A}}$ of a constant c of Σ is an element of X.*
- *the interpretation $f^{\mathcal{A}}$ of a n-ary function name f of Σ is a function from X^n into X.*

The set X in Definition 2.1.2 is called the *superuniverse* of the state $\mathcal{A}$ and is denoted by $\|\mathcal{A}\|$. The interpretations of the nullary logic names *True*, *False* and *undef* are distinct elements of X. The boolean operations behave in the usual way on the boolean values *True* and *False* and produce *undef* if at least one of the arguments is not boolean.

Formally, all the basic functions are *total*. They may, however, return the special element *undef* if they are not defined at an argument. Let us stress though that *undef* is an ordinary element of the superuniverse. Often, a basic function returns *undef* if at least one argument is *undef*.

The terms, typically denoted by t or s, are defined recursively, as in first-order logic.

Definition 2.1.3 (Term) *The* terms *of a vocabulary Σ are syntactic expressions inductively produced as follows:*

- *The variables, x, y, z, ..., elements of the set $\mathcal{V}$, are terms.*
- *The constants c of Σ are terms.*
- *If f is an n-ary function name of Σ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*

To provide terms with semantics, one needs first to assign values to variables. This is accomplished by a function called *variable assignment*.

Definition 2.1.4 (Variable assignment) *A* variable assignment *over a state $\mathcal{A}$ is a function*

$$\zeta : Map(\mathcal{V}, \|\mathcal{A}\|)$$

Given a variable assignment ζ, $\zeta[x \mapsto a]$ denotes the variable assignment which coincides with ζ except that it assigns the element a to the variable x.

The evaluation of terms is defined with respect to a variable assignment and a state:

Definition 2.1.5 (Term evaluation) *Given a state $\mathcal{A}$ of Σ, a variable assignment ζ over $\mathcal{A}$ and a term t of Σ, the value $[\![t]\!]^{\mathcal{A}}_{\zeta}$ is defined by induction on t's length as follows:*

- $[\![x]\!]^{\mathcal{A}}_{\zeta} = \zeta(x)$
- $[\![c]\!]^{\mathcal{A}}_{\zeta} = c^{\mathcal{A}}$
- $[\![f(t_1, \ldots, t_n)]\!]^{\mathcal{A}}_{\zeta} = f^{\mathcal{A}}([\![t_1]\!]^{\mathcal{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathcal{A}}_{\zeta})$

Similarly to the terms, the formulas can be interpreted in a state, with respect to a variable assignment. Note that the formulas cannot return *undef*: They are either *True* or *False*.

Definition 2.1.6 (Formula semantics) *For a state $\mathcal{A}$ of Σ, a variable assignment ζ over $\mathcal{A}$ and a formula ϕ consisting of terms over Σ, the truth value $[\![\phi]\!]_\zeta^\mathcal{A}$ is defined by induction on ϕ's length as follows:*

$$[\![s = t]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if}\ [\![s]\!]_\zeta^\mathcal{A} = [\![t]\!]_\zeta^\mathcal{A}; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

$$[\![\neg\phi]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if}\ [\![\phi]\!]_\zeta^\mathcal{A} = \mathit{False}; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

$$[\![\phi \wedge \psi]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if}\ [\![\phi]\!]_\zeta^\mathcal{A} = \mathit{True\ and}\ [\![\psi]\!]_\zeta^\mathcal{A} = \mathit{True}; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

$$[\![\phi \vee \psi]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if}\ [\![\phi]\!]_\zeta^\mathcal{A} = \mathit{True\ or}\ [\![\psi]\!]_\zeta^\mathcal{A} = \mathit{True}; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

$$[\![\phi \Rightarrow \psi]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if}\ [\![\phi]\!]_\zeta^\mathcal{A} = \mathit{False\ or}\ [\![\psi]\!]_\zeta^\mathcal{A} = \mathit{True}; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

$$[\![\forall x\, \phi]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if}\ [\![\phi]\!]_{\zeta[x\mapsto a]}^\mathcal{A} = \mathit{True\ for\ every}\ a \in \|\mathcal{A}\|; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

$$[\![\exists x\, \phi]\!]_\zeta^\mathcal{A} = \begin{cases} \mathit{True}, & \mathit{if\ there\ exists}\ a \in \|\mathcal{A}\|\ \mathit{with}\ [\![\phi]\!]_{\zeta[x\mapsto a]}^\mathcal{A} = \mathit{True}; \\ \mathit{False}, & \mathit{otherwise}. \end{cases}$$

2.1.2 Locations and Updates

As a state is described through functions and their current interpretation, the change of a state is described in terms of changing its function values at particular points. To capture state changes, the notions of *locations* and *updates* are introduced.

Definition 2.1.7 (Location) *Given a state $\mathcal{A}$ of Σ, a* location *of $\mathcal{A}$ is a pair $(f, (a_1, \ldots, a_n))$, where f is an n-ary function name of Σ and $a_1, \ldots, a_n$ are elements of $\|\mathcal{A}\|$.*

To change the value of a location, the notion of *update* is used:

Definition 2.1.8 (Update) *Given a state $\mathcal{A}$, an* update *of $\mathcal{A}$ is a pair (loc, val) where loc is a location of the state $\mathcal{A}$ and val is an element of $\|\mathcal{A}\|$. A set of updates is known as an* update set.

Given the parallelism underlying several transition rules, a prescribed update set might be inconsistent, *i.e.*, it contains at least two clashing updates.

Rule	Meaning
skip	do nothing
$f(t_1, \ldots, t_n) := s$	update f at the arguments $t_1, \ldots, t_n$ to s
$P\ Q$	execute P and Q in parallel
if ϕ **then** P **else** Q	if ϕ, then execute P, else execute Q
forall x **with** ϕ **do** $P(x)$	execute $P(x)$ in parallel for each x satisfying ϕ
choose x **with** ϕ **do** $P(x)$	choose an arbitrary x satisfying ϕ and then execute $P(x)$
let $x = t$ **in** P	assign the value of t to x and then execute P
P **seq** Q	execute P and then execute Q
$r(t_1, \ldots, t_n)$	call r with the values of $t_1, \ldots, t_n$ as parameters
P **where** Q	execute P assuming the abbreviations implied by Q

Table 2.1: The ASM transition rules.

Definition 2.1.9 (Consistent update set) *An update set U is* consistent *if the following condition is satisfied for any location loc and elements val, val′:*

$$(loc, val) \in U \wedge (loc, val') \in U \Rightarrow val = val'$$

2.1.3 Transition Rules

A means of updating the abstract states of ASMs is given by *transition rules*, informally described in Table 2.1. The *syntax of ASM programs* is defined by these rules (and a rule introduced in Section 2.1.4), typically denoted by P or Q.

Definition 2.1.10 (ASM) *An* abstract state machine *M is made of a vocabulary Σ, a set of initial states for Σ, a set of rule declarations, and a special rule name of arity zero, named the* main rule *of the machine.*

In a given state, under a given variable assignment, a transition rule yields an update set. Note that a transition rule can be recursive and consequently produce no update set at all. Therefore, the semantics of transition rules is defined by the calculus in Table 2.2.

Definition 2.1.11 (Transition rule semantics) *A transition rule P generates the update set U in a state $\mathcal{A}$ under a variable assignment ζ if and only if $[\![P]\!]_\zeta^\mathcal{A} \rhd U$ is derivable in the calculus defined in Table 2.2.*

Table 2.2 skips the semantics of the **where** rule, since this is easily reducible to the **let** construct. In Definition 2.1.11, if $[\![P]\!]_\zeta^\mathcal{A} \rhd U$ is derivable in the calculus, then the update set U is typically identified with $[\![P]\!]_\zeta^\mathcal{A}$.

To fire a consistent update set, all its members are fired *simultaneously*. The result is a new state, with the same vocabulary and superuniverse, where the interpretations of the function names are changed accordingly to the updates in the update set.

$\dfrac{}{[\![\mathbf{skip}]\!]^{\mathcal{A}}_{\zeta} \rhd \emptyset}$	
$\dfrac{}{[\![f(t_1,\ldots,t_n) := s]\!]^{\mathcal{A}}_{\zeta} \rhd \{(loc, val)\}}$	where $loc = (f, ([\![t_1]\!]^{\mathcal{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathcal{A}}_{\zeta}))$ and $val = [\![s]\!]^{\mathcal{A}}_{\zeta}$
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta} \rhd U \quad [\![Q]\!]^{\mathcal{A}}_{\zeta} \rhd V}{[\![P\ Q]\!]^{\mathcal{A}}_{\zeta} \rhd U \cup V}$	
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta} \rhd U}{[\![\mathbf{if}\ \phi\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!]^{\mathcal{A}}_{\zeta} \rhd U}$	if $[\![\phi]\!]^{\mathcal{A}}_{\zeta} = True$
$\dfrac{[\![Q]\!]^{\mathcal{A}}_{\zeta} \rhd V}{[\![\mathbf{if}\ \phi\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!]^{\mathcal{A}}_{\zeta} \rhd V}$	if $[\![\phi]\!]^{\mathcal{A}}_{\zeta} = False$
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} \rhd U}{[\![\mathbf{let}\ x = t\ \mathbf{in}\ P]\!]^{\mathcal{A}}_{\zeta} \rhd U}$	where $a = [\![t]\!]^{\mathcal{A}}_{\zeta}$
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} \rhd U_a \quad \text{for each } a \in I}{[\![\mathbf{forall}\ x\ \mathbf{with}\ \phi\ \mathbf{do}\ P]\!]^{\mathcal{A}}_{\zeta} \rhd \bigcup_{a \in I} U_a}$	where $I = \{a \in \Vert\mathcal{A}\Vert \mid [\![\phi]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} = True\}$
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} \rhd U}{[\![\mathbf{choose}\ x\ \mathbf{with}\ \phi\ \mathbf{do}\ P]\!]^{\mathcal{A}}_{\zeta} \rhd U}$	if $a \in \{a \in \Vert\mathcal{A}\Vert \mid [\![\phi]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} = True\}$
$\dfrac{}{[\![\mathbf{choose}\ x\ \mathbf{with}\ \phi\ \mathbf{do}\ P]\!]^{\mathcal{A}}_{\zeta} \rhd \emptyset}$	if $\{a \in \Vert\mathcal{A}\Vert \mid [\![\phi]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} = True\} = \emptyset$
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta} \rhd U \quad [\![Q]\!]^{\mathcal{A}+U}_{\zeta} \rhd V}{[\![P\ \mathbf{seq}\ Q]\!]^{\mathcal{A}}_{\zeta} \rhd U \oplus V}$	if U is consistent
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta} \rhd U}{[\![P\ \mathbf{seq}\ Q]\!]^{\mathcal{A}}_{\zeta} \rhd U}$	if U is inconsistent
$\dfrac{[\![P]\!]^{\mathcal{A}}_{\zeta[x \mapsto a]} \rhd U}{[\![r(t)]\!]^{\mathcal{A}}_{\zeta} \rhd U}$	where $r(x) = P$ is a rule declaration and $a = [\![t]\!]^{\mathcal{A}}_{\zeta}$

Table 2.2: The semantics of the ASM rules.

Definition 2.1.12 (Firing updates) *Given a consistent update set U and a state $\mathcal{A}$, the result of firing U in $\mathcal{A}$, denoted by $\mathcal{A} + U$, is a state $\mathcal{A}'$, where the following conditions are met for every function name f of Σ:*

- $\Vert\mathcal{A}\Vert = \Vert\mathcal{A}'\Vert$;
- $f^{\mathcal{A}'}(a_1, \ldots, a_n) = b$ *for every update* $((f, (a_1, \ldots, a_n)), b) \in U$.
- *if f is not an external function, then $f^{\mathcal{A}'}(a_1, \ldots, a_n) = f^{\mathcal{A}}(a_1, \ldots, a_n)$, for every $(a_1, \ldots, a_n) \in \Vert\mathcal{A}\Vert^n$ for which there exists no $b \in \Vert\mathcal{A}\Vert$ with $((f, (a_1, \ldots, a_n)), b) \in U$.*

A run of an ASM consists of a sequence of states, each derived from the preceding state by firing the updates yielded by the main rule. If the generated update set is inconsistent or not defined, the ASM reached the final state in the respective run.

Definition 2.1.13 (ASM run) *Given a variable assignment ζ and an ASM M with vocabulary Σ, initial state $\mathcal{A}_0$ and main rule r, a* run *of M is a possibly infinite sequence $\mathcal{A}_0, \mathcal{A}_1, \ldots$ of states for Σ such that:*

- *if $[\![P]\!]_{\zeta}^{\mathcal{A}_n}$ is consistent and defined, then $\mathcal{A}_{n+1}$ is the state obtained by firing the updates of the set $[\![P]\!]_{\zeta}^{\mathcal{A}_n}$ in the state $\mathcal{A}_n$.*
- *if $[\![P]\!]_{\zeta}^{\mathcal{A}_n}$ is not consistent or not defined, then $\mathcal{A}_n$ is the last state of the sequence.*

2.1.4 Importing New Elements

The transition rules in Table 2.1 suffice for many purposes, but they do not suffice, for instance, for systems that require to increase their working space. For this purpose, as a means to extend the vocabulary of a state, an **import** construct is considered together with a possibly infinite universe, called *reserve*, from which *new* (fresh) elements are imported.

The **import** construct, whose general form is given below, has the following meaning: "*Choose an element from the reserve, delete it from the reserve, and then execute P where every occurrence of x is replaced by the reserve element*".

import x **do** $P(x)$

The precise semantics of this construct, defined in [27, §2.4], goes much beyond the scope of this chapter and therefore is skipped. The different rules specified for the **import** construct guarantee, as shown in [27, §2.4], that "imports" executed in parallel generate fresh, *pairwise different*, elements and that permutations of the reserve universe do not change the semantics of ASM rules. Otherwise said, the semantic does not depend on the elements picked up from the reserve universe by the **import** construct.

2.2 Notational Conventions

Before getting started, we should establish some notations. This section includes the most frequently used ones, the majority of which are list operations. Many readers could just skip this section and refer back to it as necessary.

We denote by $\mathbb{N}$ the set of *natural numbers*, including 0.

$\mathcal{P}(A)$ is used to denote the *powerset* of the set A, *i.e.*, the set of all subsets of A.

We denote by $Map(A, B)$ the set of all *mappings* (functions) from a *domain* A to a *codomain* B.

$[x_1,\ldots,x_n]$ is the list containing the elements $x_1,\ldots,x_n$; $[\,]$ is the empty list.

$L(i)$ is the i-th element of the list L.

$\mathit{length}(L)$ returns the length of the list L.

$L \cdot L'$ is the concatenation of the lists L and L'.

$\mathit{sublist}(L,L')$ is true if the list L is a sublist of the list L'. More exactly, $\mathit{sublist}(L,L')$ holds if there exist two lists L'' and L''' such that $L'' \cdot L \cdot L''' = L'$.

$\mathit{last}(L,L')$ is true if the last occurrence of the list L in the list L' is at the beginning of L'. More precisely, $\mathit{last}(L,L')$ holds if for every two lists L'' and L''' such that $L'' \cdot L \cdot L''' = L'$, it holds $L'' = [\,]$.

$\mathit{prefix}(L,L')$ returns the sublist of the list L' before the last occurrence of L in L'. More exactly, $\mathit{prefix}(L,L') = L''$ if there exists a list L''' such that $\mathit{last}(L, L \cdot L''')$ and $L'' \cdot L \cdot L''' = L'$.

$\mathit{suffix}(L,L')$ returns the sublist of the list L' after the last occurrence of the list L in L'. More precisely, $\mathit{suffix}(L,L') = L''$ if $\mathit{last}(L, L \cdot L'')$ and there exists a list L''' such that $L''' \cdot L \cdot L'' = L'$.

$\mathit{top}(L)$ returns the last element of the list L.

$\mathit{pop}(L)$ returns the list L without the last element.

$\mathit{push}(L,x)$ is the result of pushing x to the right of the list L, *i.e.*, the list $L \cdot [x]$.

$\mathit{take}(L,n)$ returns the list consisting of the last n elements of the list L.

$\mathit{drop}(L,n)$ returns the list resulting from dropping the last n elements from the list L.

$\mathit{split}(L,n)$ splits off the last n elements of the list L. More exactly, $\mathit{split}(L,n)$ is the pair (L',L''), where $L' \cdot L'' = L$ and $\mathit{length}(L'') = n$.

$L[y/x]$ yields the list obtained by replacing all occurrences of x by y in the list L.

$L \sqsubseteq_{\mathit{suf}} L'$ is true if the lists L and L' of lengths m and n, respectively, and the relation $\sqsubseteq$ satisfy the following conditions: $m \geq n$ and $L(m-n+i) \sqsubseteq L'(i)$, for every $i = \overline{0, n-1}$.

$L \sqsubseteq_{\mathit{len}} L'$ is true if the lists L and L' of lengths m and n, respectively, and the relation $\sqsubseteq$ satisfy the following conditions: $m = n$ and $L(i) \sqsubseteq L'(i)$, for every $i = \overline{0, m-1}$.

$L \in_{\mathit{suf}} \mathcal{L}$ is true if the list L and the set of lists $\mathcal{L}$ satisfy the following condition: There exists a list $L' \in \mathcal{L}$ such that $L \sqsubseteq_{\mathit{suf}} L'$.

$L \in_{\mathit{len}} \mathcal{L}$ is true if the lists L and the set of lists $\mathcal{L}$ satisfy the following condition: There exists a list $L' \in \mathcal{L}$ such that $L \sqsubseteq_{\mathit{len}} L'$.

3

Type Safety of C$^\sharp$

Program testing can be used to show the presence of bugs, but never to show their absence.

Edsgar W. Dijkstra

In this chapter, we *formally prove* type safety of a large and representative subset of C$^\sharp$, named C$^\sharp{}_\mathcal{S}$. First, the *abstract syntax* and the *execution environment* of C$^\sharp{}_\mathcal{S}$ programs are formally specified. We then define the *type constraints* and the *definite assignment analysis dataflow equations* that are guaranteed to hold for C$^\sharp{}_\mathcal{S}$ programs, upon their successful compilation. The formal specifications of the *static* and *dynamic semantics* of C$^\sharp{}_\mathcal{S}$ are further provided through an *abstract interpreter* for C$^\sharp{}_\mathcal{S}$ programs, defined as an *ASM model*. Finally, we prove that C$^\sharp{}_\mathcal{S}$ is type-safe, *i.e.*, the execution on the ASM model (for the C$^\sharp{}_\mathcal{S}$ semantics) of legal C$^\sharp{}_\mathcal{S}$ methods, accepted by the definite assignment analysis and satisfying the type constraints, does not produce any *run-time type violations*. Moreover, we show that the execution leaves the C$^\sharp{}_\mathcal{S}$ programs in a good state, where certain structural constraints are satisfied.

Basis Our type safety proof is built on top of the ASM model for the C$^\sharp$'s semantics, defined by Stärk *et al*. [24]. As this model is *not* a contribution of this thesis, to aid the reader, we describe it separately in Section 3.5 and include its operational semantics rules in the Appendix. We use this model, but with a few *minor* changes. It is about some differences, pointed out in detail in Section 3.5, between the original model [24] and the C$^\sharp$ Language Specifications [74], which we were able to fix as a result of considering through our proof all language intricacies.

Challenges C$^\sharp{}_\mathcal{S}$ is a large sequential sublanguage of C$^\sharp$ which includes features that we believe to be important for an investigation of C$^\sharp$'s type safety. Despite the simple and expressive C$^\sharp$ syntax, we had to consider a large number of proof cases, mainly due to the C$^\sharp$'s unified type system including some special value types, called structs. Besides the lengthy proof, we have encountered the following technical difficulties. A key ingredient of C$^\sharp{}_\mathcal{S}$'s type safety, the soundness of the definite assignment analysis, performed by the C$^\sharp$ compiler to guarantee that every variable is initialized before its value is accessed, is pretty difficult to prove: Firstly, because the data flow equations underlying the definite assignment analysis do not always have a unique solution, and secondly, because the struct type variables are handled specially by the analysis. The "intuitive" invariants, *i.e.*, the invariants apparently expressing C$^\sharp{}_\mathcal{S}$'s type safety, cannot be proved without considering and proving other ”less intuitive” (rather technical) invariants. Due to some circular dependencies, attempting to prove the latter (as separate lemmas) before the type safety theorem is hopeless as they require the proof of the intuitive invariants. Therefore, an important idea for the proof consists in identifying the (least) set of invariants needed for the type safety result. Moreover, it was not trivial to relate analyzes performed by the C$^\sharp$ compiler, *e.g.*, the definite assignment analysis, to run-time properties since the evaluation of certain constructs, e.g, struct fields, might involve several intermediate steps. Furthermore, in the context of C$^\sharp$'s call-by-reference mechanism, it was tricky to track what locations are initialized as, for example, several field locations of a struct value might be initialized though the struct value location is considered uninitialized.

Features omitted from C$^\sharp$ To give a flavor of the full C$^\sharp$, we list here the main features omitted:

- *properties*, *indexers*, and *events*: They are implemented in C$^\sharp$ through syntactic sugar, and therefore can easily be reduced to C$^\sharp{}_{\mathcal{S}}$ constructs, as shown in [24].
- *arrays* and *static members*: Including them would only trigger the extension of some definitions and a few more proof cases, but it would not increase the technical complexity of the proof.
- *multi-threading*: The C$^\sharp$ thread model has been formally specified by Stärk and Börger [112]. The main invariants of this model have then been formally verified by Stärk [111] in the AsmTP system, an interactive proof assistant implemented in Prolog and based on LPTP [110].
- *generics*, *anonymous methods*, and *iterators*: The semantics of these features has been formalized by Jula [83]. The anonymous methods and the iterators are implemented through syntactic sugar, as shown in [83].

Chapter outline The rest of this chapter is organized as follows. Section 3.1 gives an overview of the C$^\sharp{}_{\mathcal{S}}$ language. A formal description of C$^\sharp{}_{\mathcal{S}}$ consisting of definitions for the execution environment and the grammar is provided in Section 3.2. Section 3.3 formalizes the type constraints that should be satisfied by every C$^\sharp{}_{\mathcal{S}}$ program, upon successful compilation. Section 3.4 formally specifies the definite assignment analysis, which is then proved sound. The static and dynamic semantics of C$^\sharp{}_{\mathcal{S}}$, as defined in [24], are included in Section 3.5. Finally, Section 3.6 presents a proof of C$^\sharp{}_{\mathcal{S}}$'s type safety. Section 3.8 summarizes and gives directions for future work. For the reader's convenience, the operational semantics rules of [24]'s model are included in the Appendix.

3.1 An Overview of C$^\sharp{}_{\mathcal{S}}$

C$^\sharp{}_{\mathcal{S}}$ is a large and representative sublanguage of C$^\sharp$ that includes classes, interfaces, instance members (fields, methods, and constructors), inheritance of members, dynamic method dispatch, local variables, and exceptions. Moreover, C$^\sharp{}_{\mathcal{S}}$ has delegates, structs, call-by-reference (`ref`/`out` parameters), boxing, and unboxing.

Delegates The *delegate types* are reference types, built with the idea of being the type-safe method pointers in C$^\sharp$. Informally, a *delegate*, *i.e.*, an instance of a delegate type, is an object that points towards an *invocation list* of pairs of *target objects* and *target methods*. Upon the invocation of a delegate with a list of arguments, the methods in its invocation list are invoked *sequentially* with the corresponding target object and the given arguments, returning to the delegate caller the return value of the last method in the list.

Structs Besides primitive types, C$^\sharp$ supports some special value types called *structs*. Every struct has the library class `System.ValueType` [10] as the direct base class. In particular, a struct is *sealed*, meaning that it is not possible for a struct to extend another struct. Additionally, a struct can implement one or more interfaces. Similarly to a class, a struct can declare instance fields. A *struct value* is a “self-contained” value. This is so because a *struct instance* can be represented as a *mapping* assigning values to the instance fields of the struct. As the structs are sealed, the instance methods in structs cannot be declared as virtual, since it is just not possible to subtype them and consequently, the definitions of methods cannot be overridden.

Boxing and unboxing To fill the gap between value types and reference types, *i.e.*, to have an *unified type system*, C$^\sharp$ supports two special operations called *boxing* and *unboxing*. Through *boxing*, a *copy* of a value type instance can be ”packed” in an object on the heap, along with the necessary internal data required to create a valid reference object. The inverse process, *i.e.*, converting a reference type to a value type, is called *unboxing*. An unboxing ”back” to a value type returns the value boxed in the object if the object is indeed a boxed value of the given value type.

Although the structs cannot declare virtual methods, the virtual methods defined by `System.ValueType`, `object`, and by the interfaces implemented by a struct (if any) can, however, be called on the struct instances, but *only* if the instances are boxed. This makes sense as the value types have identity only when boxed.

Call-by-reference In C$^\sharp$ a `ref` parameter is used for ”by reference” parameter passing in which the parameter acts as an alias for the caller-provided argument. An `out` parameter is similar to a `ref` parameter except that the initial value (if any) of the caller-provided argument is not important. This is because a variable is required to hold a value before it can be passed as a `ref` parameter, whereas a variable need not hold a value before it can be passed as an `out` parameter.

3.2 A Formal Description of C$^\sharp{}_S$

In this section, we provide a formal description of C$^\sharp{}_S$ along the lines of the C$^\sharp$ model [24]. Section 3.2.1 specifies the components of the execution environment. The grammar of C$^\sharp{}_S$ is defined in Section 3.2.2.

3.2.1 The Execution Environment

Every C$^\sharp{}_S$ method runs in an *execution environment* that contains the type hierarchy and the field and method declarations. The classification of the C$^\sharp{}_S$ *types* is given in Table 3.1. Thus, a

Universe of types			Typical use
Type	=	*RefType* ∪ *ValueType*	T, T', T''
RefType	=	*Class* ∪ *Interface*	–
ValueType	=	*Struct* ∪ *PrimitiveType*	–
Class			C, C'
Delegate	⊂	*Class*	D
Interface			I, I'
Struct			S, S'
PrimitiveType	=	{`bool`, `int`, `long`, `double`}	–

Table 3.1: C$\sharp_S$'s types.

Universe of values			Typical use
Val	=	*ObjRef* ∪ *SimpleVal* ∪ *StructVal*	*val*, *val*′, *val*″
SimpleVal			–
ObjRef			*ref*, *ref*′, *d*
StructVal	=	*Map*(*Struct*::*Field*, *Val*)	–

Table 3.2: C$\sharp_S$'s values.

type is a reference type or a value type. Additionally, `void` can *only* be used as a method return type. The *reference types* are the *classes* (including the *delegate types*[1]) and the *interfaces*. A *value type* is either a *struct* or a *primitive type*.

Corresponding to the above types, there exists the following kinds of values, gathered in Table 3.2: *object references*, *simple values*, *i.e.*, values of primitive types, and *struct values*.

We assume that the execution environment organizes the classes, structs and interfaces into an *inheritance hierarchy*. The *subtype relation*, introduced in Definition 3.2.1, depends on this hierarchy.

Definition 3.2.1 (Compile-time compatibility) *The* subtype relation $\preceq$ *is the least reflexive and transitive relation such that*

- *if* $T =$ `int` *and* $T' \in \{$`long`, `double`$\}$*, or*
- *if* $T =$ `long` *and* $T' =$ `double`*, or*
- *if* $T \in$ *Type and* $T' =$ `object`*, or*
- *if* $T \in$ *Class extends* $T' \in$ *Class, or*
- *if* $T \in$ *Class* ∪ *Struct* ∪ *Interface implements* $T' \in$ *Interface, or*
- *if* $T \in$ *ValueType and* $T' =$ `System.ValueType`*, or*

[1]Due to a syntactic sugar defined and explained in Section 3.2.2, we treat the delegate types as special class types. The official documentation [74] is ambiguous in that respect: It defines the delegate types as classes, but treats them as distinct in most places.

Universe	Typical use	Element name
Field	F	field name
MRef	$C{::}M$, $S{::}M$, $T{::}M$, $T'{::}M$, $T'{::}M'$	method reference
Loc	loc, loc'	local variable or parameter

Table 3.3: The environment specific universes.

Function definition			Function name
instFields	:	$Map(Class \cup Struct, \mathcal{P}(Type{::}Field))$	class/struct instance fields
fieldType	:	$Map(Class{::}Field \cup Struct{::}Field, Type)$	field declared type
localVars	:	$Map(MRef, \mathcal{P}(Loc))$	method local variables
locType	:	$Map(MRef \times Loc, Type)$	local variable type
valueParams	:	$Map(MRef, \mathcal{P}(Loc))$	”by value” parameters
refParams	:	$Map(MRef, \mathcal{P}(Loc))$	`ref` parameters
outParams	:	$Map(MRef, \mathcal{P}(Loc))$	`out` parameters
paramType	:	$Map(MRef \times Loc, Type)$	parameter declared type
paramIndex	:	$Map(MRef \times Loc, \mathbb{N})$	parameter index
retType	:	$Map(MRef, Type \cup \{\texttt{void}\})$	method return type
callKind	:	$Map(MRef, \{\texttt{Virtual}, \texttt{NonVirtual}\})$	method kind

Table 3.4: The components of the execution environment.

- *if* $T \in Delegate$ *and* $T' =$ `System.Delegate`,

then $T \preceq T'$.

Incomplete specifications There are two specification gaps in [74], and each of them would imply the non-transitivity of $\preceq$:

- Concerning the definition of *interfaces implementation* in [74, §13.4], there are only two cases stated for a class C to *implement* an interface I: The base class list of C includes I or an interface I' which has I as a base interface (not necessarily direct). The following case is missing: C implements I if C extends a class C' and C' implements I.
- In [74, §6.1.4], there should also be stated an implicit reference conversions from any delegate type to the class `System.Delegate` and to the two interfaces implemented by the class `System.Delegate`, namely, `System.ICloneable` and `System.ISerializable`.

Definition 3.2.1 fills these gaps by defining $\preceq$ as a transitive closure.

Each class and struct declares types for a set of instance fields, and each class, struct or interface specifies signatures and return types[2] for a set of instance methods. Formally, we consider the universes in Table 3.3 and the environment components in Table 3.4.

[2]Note that, following [74, §1.7.3], the signature of a method does not include its return type.

For any class or struct T, $\mathit{instFields}(T)$ is the set of instance fields of T, including the inherited fields[3]. For any class or struct T and for any field $T'::F \in \mathit{instFields}(T)$, $\mathit{fieldType}(T'::F)$ is the *declared type* of $T'::F$. For any method $T::M$, $\mathit{localVars}(T::M)$ returns the set of local variables of $T::M$. The type of a local variable loc declared by a method $T::M$ is maintained in $\mathit{locType}(T::M, \mathit{loc})$. The parameters passed "by value" of a method $T::M$ are given by $\mathit{valueParams}(T::M)$, whereas the parameters passed "by reference", *i.e.*, the `ref` and `out` parameters, are maintained in $\mathit{refParams}(T::M)$ and $\mathit{outParams}(T::M)$, respectively. The type of a parameter loc of a method $T::M$ is returned by $\mathit{paramType}(T::M, \mathit{loc})$, while the index of loc in the parameter list of $T::M$ is given by $\mathit{paramIndex}(T::M, \mathit{loc})$. Note that, the parameter indexed with 0 is the `this` pointer.

The definition of $\mathit{paramType}$ is subject to the following restriction concerning the type of the `this` pointer:

[`this` parameter type] If $T::M \in \mathit{MRef}$, then $\mathit{paramType}(T::M, \texttt{this}) = T$.

For any method $T::M$, $\mathit{retType}(T::M)$ returns the possibly `void` return type of $T::M$. Depending on whether a method $T::M$ is virtual, $\mathit{callKind}(T::M)$ is `Virtual` or `NonVirtual`, respectively.

We define the (derived) function $\mathit{paramTypes} : \mathit{Map}(\mathit{MRef}, \mathit{List}(\mathit{Type}))$, which, given a method, returns the list of its parameter types.

Definition 3.2.2 *For a method $T::M$ and an index $i = 0, n-1$, where n is the cardinal of the set $A = \mathit{valueParams}(T::M) \cup \mathit{refParams}(T::M) \cup \mathit{outParams}(T::M)$, we define:*

$$\begin{gathered}\mathit{paramTypes}(T::M)(i) = \mathit{paramType}(T::M, \mathit{loc}),\ \textit{where}\\ \mathit{loc} \in A \textit{ such that } \mathit{paramIndex}(T::M, \mathit{loc}) = i\end{gathered}$$

The `this` variable of a class instance method or constructor behaves as a value parameter. The `this` variable of a struct instance method is regarded as a `ref` parameter, whereas the `this` variable of a struct instance constructor behaves as an `out` parameter. We express these conditions in terms of restrictions on the definitions of $\mathit{valueParams}$, $\mathit{refParams}$, and $\mathit{outParams}$:

[`this` variable] Let $T::M \in \mathit{MRef}$.

1. If $T \in \mathit{RefType}$, then `this` $\in \mathit{valueParams}(T::M)$.
2. If $T \in \mathit{Struct}$, then
   (a) if $T::M$ is an instance method, then `this` $\in \mathit{refParams}(T::M)$.
   (b) if $T::M$ is an instance constructor, then `this` $\in \mathit{outParams}(T::M)$.

The implementations of virtual methods declared by a class can be *overridden* in methods (marked with the C$^\sharp$ keyword `override`) of derived classes and structs[4]. Also, the (virtual)

[3]A struct can only inherit fields from `System.ValueType` and `object`. However, as these classes do not declare any instance fields, the set *instFields* associated to a struct can only contain instance fields declared by the struct itself.

[4]A struct can override the virtual methods declared by the classes `System.ValueType` and `object`.

methods declared by an interface have to be *implemented* in classes or structs that implement the interface. The following conditions should be satisfied when a method overrides/implements another method:

[**override/implement**] If $T::M \in MRef$ overrides/implements $T'::M \in MRef$, then:

1. $length(paramTypes(T::M)) = length(paramTypes(T'::M))$
2. for every $i = 1, length(paramTypes(T::M)) - 1$,
$$paramTypes(T::M)(i) = paramTypes(T'::M)(i)$$
3. $retType(T::M) = retType(T'::M)$

3.2.2 The Grammar of $C\sharp_S$

The grammar for the statements and expressions of $C\sharp_S$ is defined in Figure 3.1. This grammar can also be viewed as defining the corresponding ASM domains *Exp* and *Stm*. *Lit* denotes the set of *literals*. To handle literals, we consider two functions $valueOfLiteral : Map(Lit, Val)$ and $typeOfLiteral : Map(Lit, Type)$ which, given a literal, return its value and its type, respectively. These functions are defined according to [74, §2.4.4].

The set of *variable expressions*, also known as ”lvalues”, is given by *Vexp*. The grammar production *RefExp* . *Field* requires a compile-time check to ensure that the type of the expression *RefExp* is a reference type (see Section 3.3 for the compile-time type checks). If the type of the expression is not a reference type, the resulting expression is not considered a variable expression.

The set *Sexp* contains the *statement expressions*, *i.e.*, the statements that return results. The *delegate creation expressions* are represented in the grammar by the production rules `new` *Delegate* (*Dexp*) and `new` *Delegate* (*Exp*), where *Dexp* stands for a special expression consisting of an expression (the value of which gives the target object pointed to by the delegate) and a method (which, as a result of a method lookup, yields the target method pointed to by the delegate) that should have *no* arguments specified. The delegate calls are defined in the grammar through the production *Exp* ([*Args*]), where *Exp* is assumed to return the delegate to be called with the list of arguments produced by *Args*.

Every `try`-`catch`-`finally` statement contains, besides a `try` block, at least one `catch` *clause*, an optional *general* `catch` *clause* (*i.e.*, a clause that can catch every exception), and an optional `finally` block. We assume all the constraints, stipulated by [74], concerning the structure of the `try`-`catch`-`finally` statements. In particular, we assume that:

- no `return` occurs in `finally` blocks;
- no `break`, `continue`, or `goto` statement jumps out of a `finally` block;
- the `throw` statements without expression can only occur in `catch` blocks;
- the exception classes in a *Catch* clause should derive (not necessarily directly) from `System.Exception` [6] and should appear in a non-decreasing type order.

Figure 3.1 The grammar of expressions and statements.

$$
\begin{array}{lll}
\mathit{Exp} & ::= & \mathit{Lit} \mid \mathit{Vexp} \mid \mathit{Exp}\ \mathit{Bop}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{?}\ \mathit{Exp}\ \texttt{:}\ \mathit{Exp} \\
 & \mid & (\ \mathit{Type}\ )\ \mathit{Exp} \mid \mathit{Sexp} \mid (\ \mathit{Exp}\ ) \mid \mathbf{this} \mid \mathbf{base} \mid \mathit{Exp}\ \mathbf{is}\ \mathit{Type} \\
 & \mid & \mathit{Exp}\ \mathbf{as}\ \mathit{RefType} \mid \mathbf{new}\ \mathit{Delegate}\ (\ \mathit{Dexp}\ ) \mid \mathbf{new}\ \mathit{Delegate}\ (\ \mathit{Exp}\ ) \\
\mathit{Vexp} & ::= & \mathit{Loc} \mid \mathit{Field} \mid \mathit{Vexp}\ .\ \mathit{Field} \mid \mathit{RefExp}\ .\ \mathit{Field} \\
\mathit{Sexp} & ::= & \mathit{Vexp}\ =\ \mathit{Exp} \mid \mathit{Vexp}\ \mathit{Aop}\ \mathit{Exp} \\
 & \mid & \mathbf{new}\ \mathit{Type}\ (\ [\mathit{Args}]\ ) \mid \mathit{Exp}\ .\ \mathit{MRef}\ (\ [\mathit{Args}]\ ) \mid \mathit{Exp}\ (\ [\mathit{Args}]\ ) \\
\mathit{Dexp} & ::= & \mathit{Exp}\ .\ \mathit{MRef} \\
\mathit{Bop} & ::= & \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{+} \mid \texttt{-} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{\&} \mid \texttt{|} \\
\mathit{Aop} & ::= & \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{\&=} \mid \texttt{|=} \\
\mathit{Exps} & ::= & \mathit{Exp}\ \{,\ \mathit{Exp}\} \\
\mathit{Stm} & ::= & \mathit{Sexp}\ ; \mid \mathbf{break}\ ; \mid \mathbf{continue}\ ; \mid \mathbf{goto}\ \mathit{Lab}\ ; \mid \mathbf{return}\ \mathit{Exp}\ ; \\
 & \mid & \mathbf{return}\ ; \mid \mathbf{if}\ (\ \mathit{Exp}\ )\ \mathit{Stm}\ \mathbf{else}\ \mathit{Stm} \mid \mathbf{while}\ (\ \mathit{Exp}\ )\ \mathit{Stm} \\
 & \mid & \mathbf{try}\ \mathit{Block}\ \{\mathit{Catch}\}\ [\mathbf{catch}\ \mathit{Block}]\ [\mathbf{finally}\ \mathit{Block}] \mid \mathbf{throw}\ \mathit{Exp}\ ; \\
 & \mid & \mathbf{throw}\ ; \mid \mathit{Block} \\
\mathit{Catch} & ::= & \mathbf{catch}\ (\ \mathit{Class}\ [\mathit{Loc}]\ )\ \mathit{Block} \\
\mathit{Arg} & ::= & \mathit{Exp} \mid \mathbf{ref}\ \mathit{Vexp} \mid \mathbf{out}\ \mathit{Vexp} \\
\mathit{Args} & ::= & \mathit{Arg}\ \{,\ \mathit{Arg}\} \\
\mathit{Sexps} & ::= & \mathit{Sexp}\ \{,\ \mathit{Sexp}\} \\
\mathit{Block} & ::= & \{\ \{\mathit{Bstm}\}\ \} \\
\mathit{Bstm} & ::= & \mathit{Type}\ \mathit{Loc}\ ; \mid \mathit{Lab}\ :\ \mathit{Stm} \mid \mathit{Stm}
\end{array}
$$

C$\sharp$ construct	Semantic equivalent C$\sharp_S$ construct
exp `&&` exp'	exp `?` exp' `:` `false`
exp `||` exp'	exp `?` `true` `:` exp'

Table 3.5: Derived language constructs.

Table 3.5 shows how the conditional logical operators `&&`, `||` can be syntactically reduced to C$\sharp_S$ constructs. We still, however, include `&&`, `||` in our formal approach since the definite assignment analysis (see Section 3.4) treats them specially.

To express the sequentiality of the execution of the delegate invocation list elements, [24, §6] translates every delegate declaration to a special class. An inspection of the CIL bytecode to which C$\sharp$ programs compile shows that this idea closely follows the Microsoft implementation. Thus, the declaration of a delegate type is translated to a class declaration, as described in Figure 3.2. The translation included in Figure 3.2 assumes that T is not `void`. If T is `void`, the translation is similar, except that the local variable *result* is not ”needed” anymore, and consequently, the assignment to *result* in the `while` loop is replaced by a call of `_invoke`.

Figure 3.2 The translation scheme of delegate classes.

```
delegate T D(T₁ loc₁, ..., Tₙ locₙ);

                    ⇓

sealed class D : System.Delegate
{
  public T Invoke(T₁ loc₁, ..., Tₙ locₙ)
  {
    T result = defVal(T);
    int i = 0;
    while ( i < this._length() )
    {
      result = this._invoke(i, loc₁, ..., locₙ);
      i += 1;
    }
    return result;
  }
  private extern int _length();
  private extern T _invoke(int i, T₁ loc₁, ..., Tₙ locₙ);
}
```

Remark 3.2.1 *It is worth mentioning that the translation in Figure 3.2 slightly differs than the one defined in [24], which does not initialize the variable result upon its declaration. But, if this is the case, the* `Invoke` *method would be rejected by the* C$^\sharp$ *compiler, as a result of the definite assignment analysis (Section 3.4). Therefore, our translation initializes the variable result (to T's default value).*

To keep the formal description straightforward and more modular, we get rid of some syntactic sugar in C$^\sharp$:

1. `base` is replaced by (T) `this`, where T is the direct base class of the enclosing type.

2. If F is a field name, then F is replaced by `this.`T::F, where F occurs in type T.

3. If C is a class (other than a delegate), then the instance creation `new` C::M (*args*) is replaced by `new` C`.`C::M (*args*).

4. If S is a struct, then:

   a) an assignment *vexp* $=$ `new` S::M(*args*) is replaced by *vexp*.S::M(*args*), and

   b) every *other* occurrence of `new` S::M(*args*) is replaced by *loc*.S::M(*args*), where *loc* $\in$ *Loc* is a *new temporary local variable* of type S.

5. Every general clause `catch` *block* is replaced by `catch` (`object` *loc*) *block*, with a *new* local variable *loc*.

6. Every `try`-`catch`-`finally` statement is reduced to a `try`-`finally` statement, with the `try` block made of the `try` block and `catch` clauses of the original statement and the `finally` block consisting of the `finally` block of the original statement.

Function definition			Function name
label	:	$Map(Pos, Label)$	node label
first	:	$Map(Pos, Pos \cup \{undef\})$	first child node
up	:	$Map(Pos, Pos \cup \{undef\})$	parent node
next	:	$Map(Pos, Pos \cup \{undef\})$	next brother node

Table 3.6: Attributed syntax tree specific functions.

To guarantee that the splitting in the third transformation reflects the intended meaning of `new` T::$M(args)$, we assume that the class instance constructors return the value of `this`. The transformation 4a) reflects that `new` applied to a struct triggers no object creation or memory allocation since, anyway, the structs, as value types, get their memory allocated at declaration time. The necessity of the transformation 4b) arises from the fact that the struct instance constructors need a ”home”, *i.e.*, an address, for `this` (see condition **[`this` variable]** in Section 3.2.1). Also, we assume that the struct constructors return the value of `this`.

Whenever, instead of a direct formalization of a construct, [24] uses a syntactical translation to C$^\sharp{}_S$ constructs, one should justify that the translation is correct with respect to the semantics of the construct as intended by the standard [74]. The semantics model yields a basis to formally state and prove the intended equivalence. Such a justification is skipped in [24] since it follows well-known patterns.

We denote the *positions* in a program by small Greek letters α, β, γ, etc. One can think of positions either as positions in the source code of the program or as positions in the attributed syntax tree. Positions are displayed as prefixed superscripts, for example, as in ${}^{\alpha}exp$ or in ${}^{\beta}stm$. We denote by *Pos* the universe of positions. We often refer to expressions and statements using their positions only.

To specify the dynamic semantics in Section 3.5.2, we define the (compiler specific) functions in Table 3.6, that are used during the traversal of the *attributed syntax tree* (AST). The function $label : Map(Pos, Label)$ decorates the tree nodes with *labels*, elements of the universe *Label*. A label is information – in the form of concrete C$^\sharp{}_S$ syntax – that identifies the grammar rule associated to the node. Given the node at a position α:

- $first(\alpha)$ returns the position of the first AST ”child” node (if any) of the node at α;
- $up(\alpha)$ returns the position of the ”parent” AST node (if any) of the node at α;
- $next(\alpha)$ yields the position of the next AST ”brother” node (if any) of the node at α;

3.3 The Type Constraints

To enforce type safety, the C$^\sharp$ compiler has to verify that a program is well-typed. Following the parsing and elaboration, the evaluation tree of a program is annotated with *type information*, used in the evaluation rules in Section 3.5.2.1. Thus, the compiler *infers* a static function $\mathcal{ST} : Map(Pos, Type)$ – relative to an environment defined as in Section 3.2.1 – which assigns a type to each position in an expression.

We assume that, as a result of field and method *resolution* [74, §7.4.2] performed by the compiler simultaneously with the type inference, the attributed syntax tree has exact information. In particular, every field reference is of the form T::F, and every method reference has the form T::M. Moreover, as a result of the delegate transformation in Section 3.2.2, we assume that every delegate call ${}^{\alpha}exp(args)$ (so, $\mathcal{ST}(\alpha) \in Delegate$) is replaced by the method call $exp.\mathcal{ST}(\alpha)$::`Invoke`$(args)$.

Tables 3.7 and 3.8 contain the *type constraints*, expressed in terms of conditions for $\mathcal{ST}$, which should be satisfied *after* the type inference.

For example, for an operation *exp bop exp*$'$, it is ensured that the types of the operands *exp* and *exp*$'$ are suitable for the binary operator *bop*. We assume that the set of types for which a binary operator is defined is maintained in the function $opTypes : Map(Bop, \mathcal{P}(Type \times Type))$ (implicitly) defined in [74, §7]. Also, we consider that, given a binary operator and a pair of types, the function $opResType : Map(Bop \times Type \times Type, Type)$ (implicitly) defined in [74, §7] returns the type of the result of applying *bop* to operands of the given types[5].

For a conditional expression *exp* `?` *exp*$'$ `:` *exp*$''$, either the type of *exp*$'$ is a subtype of the type of *exp*$''$ or vice-versa. The type of the whole expression is determined as the *supremum* of the operand types.

Definition 3.3.1 (Supremum) *For two types T and T',* $\sup(T, T')$ *denotes the* supremum *of T and T' with respect to the compile-time compatibility relation $\preceq$.*

In case of a boxing $(T)exp$, that is, if T is a reference type and the type of *exp* is a value type, it is ensured that the value type is a subtype of T.

For a method call, the passing mode of each argument, *i.e.*, ”by value”, `ref` or `out`, should be identical to the passing mode of the corresponding parameter. Moreover, for a value parameter, the type of the argument should be a subtype of the declared type of the corresponding parameter, whereas for a `ref` or `out` parameter, the type of the argument should be identical to the declared type of the corresponding parameter. Due to our assumption in Section 3.2.2 that a constructor returns the value of the `this` pointer, it is guaranteed that a constructor of a type T returns the type T.

In a delegate creation expression of the form `new` $D(exp.T$::$M)$, the type of *exp* should be a subtype of T and the method T::M to which the newly created delegate is going to refer should have a return type and a signature compatible with the delegate type D. This leads to the definition of *consistency*[6] in Definition 3.3.2 that allows covariance in return type and contravariance in parameter types.

Definition 3.3.2 (Consistency) *A method T::M and a delegate type D are* consistent *if the following conditions are met:*

- *for every $loc \in valueParams(T$::$M)$, there exists $loc' \in valueParams(D$::*`Invoke`*$)$ such that*

[5]We suppose that a *binary operator overload resolution* [74, §7.2.4] has been previously applied to select a specific operator to be invoked.

[6]Initially, the definition of consistency specified in [74] required that the return types and signatures of the method and delegate, respectively, should coincide. However, as one can observe in our type safety proof, the requirement is too strong, and therefore has been weakened in [82].

Expression	Constraints
${}^{\alpha}lit$	$\mathcal{ST}(\alpha) = typeOfLiteral(lit)$
${}^{\alpha}loc$	$\mathcal{ST}(\alpha) = locType(T{::}M, loc)$, where $T{::}M$ is the enclosing method
${}^{\alpha}\texttt{this}$	$\mathcal{ST}(\alpha) = T$, where T is the enclosing type
${}^{\alpha}({}^{\beta}exp\ bop\ {}^{\gamma}exp')$	$(\mathcal{ST}(\beta), \mathcal{ST}(\gamma)) \in opTypes(bop)$, $\mathcal{ST}(\alpha) = opResType(bop, \mathcal{ST}(\beta), \mathcal{ST}(\gamma))$
${}^{\alpha}({}^{\beta}exp\ \texttt{?}\ {}^{\gamma}exp'\ \texttt{:}\ {}^{\delta}exp'')$	$\mathcal{ST}(\beta) = \texttt{bool}$, $\mathcal{ST}(\gamma) \preceq \mathcal{ST}(\delta) \vee \mathcal{ST}(\delta) \preceq \mathcal{ST}(\gamma)$, $\mathcal{ST}(\alpha) = \sup(\mathcal{ST}(\gamma), \mathcal{ST}(\delta))$
${}^{\alpha}((T)\ {}^{\beta}exp)$	if $\mathcal{ST}(\beta) \in ValueType$ and $T \in RefType$, then $\mathcal{ST}(\beta) \preceq T$, $\mathcal{ST}(\alpha) = T$
${}^{\alpha}({}^{\beta}vexp\ =\ {}^{\gamma}exp)$	$\mathcal{ST}(\gamma) \preceq \mathcal{ST}(\beta)$, $t\alpha = \mathcal{ST}(\beta)$
${}^{\alpha}({}^{\beta}vexp\ bop =\ {}^{\gamma}exp)$	$opResType(bop, \mathcal{ST}(\beta), \mathcal{ST}(\gamma)) \preceq \mathcal{ST}(\beta)$ or $\mathcal{ST}(\beta) \preceq opResType(bop, \mathcal{ST}(\beta), \mathcal{ST}(\gamma))$, $\mathcal{ST}(\alpha) = \mathcal{ST}(\beta)$, $(\mathcal{ST}(\beta), \mathcal{ST}(\gamma)) \in opTypes(bop)$
${}^{\alpha}(\texttt{ref}\ {}^{\beta}vexp)$	$\mathcal{ST}(\alpha) = \mathcal{ST}(\beta)$
${}^{\alpha}(\texttt{out}\ {}^{\beta}vexp)$	$\mathcal{ST}(\alpha) = \mathcal{ST}(\beta)$
${}^{\alpha}({}^{\beta}exp\ \texttt{is}\ T)$	$\mathcal{ST}(\alpha) = \texttt{bool}$
${}^{\alpha}({}^{\beta}exp\ \texttt{as}\ T)$	$\mathcal{ST}(\alpha) = T$, if $\mathcal{ST}(\beta) \in ValueType$, then $\mathcal{ST}(\beta) \preceq T$
${}^{\alpha}({}^{\beta}exp.T{::}F)$	$\mathcal{ST}(\alpha) = fieldType(T{::}F)$, $\mathcal{ST}(\beta) \preceq T$
${}^{\alpha}({}^{\beta}exp.T{::}M(\ldots, {}^{\gamma_i}exp_i, \ldots))$	$\mathcal{ST}(\beta) \preceq T$, if $loc \in valueParams(T{::}M)$ such that $paramIndex(T{::}M, loc) = i$, then $\mathcal{ST}(\gamma_i) \preceq paramTypes(T{::}M)(i)$ if $loc \in refParams(T{::}M) \cup outParams(T{::}M)$ such that $paramIndex(T{::}M, loc) = i$, then $\mathcal{ST}(\gamma_i) = paramTypes(T{::}M)(i)$ $\mathcal{ST}(\alpha) = retType(T{::}M)$ if $T{::}M$ is not a constructor, otherwise $\mathcal{ST}(\alpha) = T$
${}^{\alpha}(\texttt{new}\ T)$	$\mathcal{ST}(\alpha) = T$
${}^{\alpha}(\texttt{new}\ D({}^{\beta}exp.T{::}M))$	$D \in Delegate$, $\mathcal{ST}(\beta) \preceq T$, $T{::}M$ and D are consistent, $\mathcal{ST}(\alpha) = D$
${}^{\alpha}(\texttt{new}\ D({}^{\beta}exp))$	$D \in Delegate$, $\mathcal{ST}(\beta) = \mathcal{ST}(\alpha) = D$

Table 3.7: The type constraints for expressions.

$$\begin{array}{lcll} paramIndex(T{::}M, loc) & = & paramIndex(D{::}\texttt{Invoke}, loc') & and \\ paramType(T{::}M, loc) & \preceq & paramType(D{::}\texttt{Invoke}, loc') & \end{array}$$

- *for every $loc \in refParams(T{::}M)$, there exists $loc' \in refParams(D{::}\texttt{Invoke})$ such that*

Statement	Constraint
`if` $({}^{\alpha}exp)$ *stm* `else` *stm′*	$\mathcal{ST}(\alpha) =$ `bool`
`while` $({}^{\alpha}exp)$ *stm*	$\mathcal{ST}(\alpha) =$ `bool`
`return`;	$retType(T{::}M) =$ `void`, where $T{::}M$ is the enclosing method
`return` ${}^{\alpha}exp$;	$\mathcal{ST}(\alpha) \preceq retType(T{::}M)$, where $T{::}M$ is the enclosing method
`throw` ${}^{\alpha}exp$;	$\mathcal{ST}(\alpha) \preceq$ `System.Exception`

Table 3.8: The type constraints for statements.

$$\begin{array}{lcll} paramIndex(T{::}M, loc') & = & paramIndex(D{::}\texttt{Invoke}, loc') & and \\ paramType(T{::}M, loc) & = & paramType(D{::}\texttt{Invoke}, loc') & \end{array}$$

- *for every* $loc \in outParams(T{::}M)$, *there exists* $loc' \in outParams(D{::}\texttt{Invoke})$ *such that*

$$\begin{array}{lcll} paramIndex(T{::}M, loc) & = & paramIndex(D{::}\texttt{Invoke}, loc') & and \\ paramType(T{::}M, loc) & = & paramType(D{::}\texttt{Invoke}, loc') & \end{array}$$

- $retType(T{::}M) \preceq retType(D{::}\texttt{Invoke})$.

We say that a method is *well-typed* if it satisfies the type checks performed by the compiler.

Definition 3.3.3 (Well-typed) *A method is* well-typed *if its body contains only expressions and statements that satisfy the type constraints in Tables 3.7 and 3.8.*

After the type constraints in Tables 3.7 and 3.8 are guaranteed, the program transformations defined in Table 3.9 are performed, where necessary. The transformations are executed *only* if the corresponding conditions in the second column of Table 3.9 (if any) hold.

For example, a conditional expression *exp* `?` *exp′* `:` *exp″* requires two type casts to be inserted. Thus, both *exp′* and *exp″* are type casted to the *supremum* of the types of *exp′* and *exp″*.

A boxing is inserted in an assignment $vexp = exp$ if the type of *exp* is a value type, whereas the type of *vexp* is a reference type. This assignment is replaced by $vexp = (T)exp$, where T is the type of *vexp*. A boxing is also added in a method call $exp.T{::}M(args)$ if the type of *exp* is a value type and T is a reference type. This is the case when the method is declared by the classes `System.ValueType` or `object` or by an interface implemented by the value type. However, if T is not a reference type, and moreover, *exp* is not a variable expression (so it could not have an address), then a "home" is created for the value of *exp* (since the `this` pointer of $T{::}M$ behaves as a reference parameter). Thus, a *new temporary local variable*, say *loc*, of the same type as *exp* is inserted into the set of local variables of the enclosing method, and the value of *exp* is assigned to *loc*. In [24], the temporary variable is created on the fly, during the run-time execution. Our formal specifications follow the Microsoft implementation and creates the variable upon the compilation.

Old expression	Condition for transformation	New expression
${}^{\beta}exp\ ?\ {}^{\gamma}exp'\ :\ {}^{\delta}exp''$	–	$exp\ ?\ (T)\,exp'\ :\ (T)\,exp''$, where $T = \sup(\mathcal{ST}(\gamma), \mathcal{ST}(\delta))$
${}^{\beta}vexp\ =\ {}^{\gamma}exp$	either $\mathcal{ST}(\beta) \in \mathit{PrimitiveType}$ or $\mathcal{ST}(\beta) \in \mathit{RefType}$ and $\mathcal{ST}(\gamma) \in \mathit{ValueType}$	$vexp\ =\ (\mathcal{ST}(\beta))\,exp$
${}^{\alpha}exp.T{::}M(args)$	$T \in \mathit{RefType}$ and $\mathcal{ST}(\alpha) \in \mathit{ValueType}$	$((T)exp).T{::}M(args)$
${}^{\alpha}exp.T{::}M(args)$	$T \in \mathit{Struct}$ and $\mathcal{ST}(\alpha) \in \mathit{Struct}$ and $exp \notin \mathit{Vexp}$	$(loc\ =\ exp).T{::}M(args)$, where loc is a *new temporary local variable* of the enclosing method $T'{::}M'$ with $locType(T'{::}M', loc) = \mathcal{ST}(\alpha)$
${}^{\gamma_i}exp_i$ in $exp.T{::}M(\ldots,{}^{\gamma_i}exp_i,\ldots)$	either $\mathcal{ST}(\gamma_i) \in \mathit{PrimitiveType}$ or $\mathcal{ST}(\gamma_i) \in \mathit{ValueType}$ and $T' \in \mathit{RefType}$, where $T' = paramTypes(T{::}M)(i)$	$(T')\,exp_i$

Table 3.9: Compile-time transformations.

3.4 The Definite Assignment Analysis in C$^\sharp$

In C$^\sharp$, local variables are not initialized by default, unlike instance fields. Therefore, in order to ensure type-safety, a C$^\sharp$ compiler must guarantee that all local variables are assigned to before their value is used. The C$^\sharp$ compiler enforces this *definite assignment rule* by a static flow analysis. Since the problem is *undecidable* in general, [74, §5.3] contains a definition of a decidable subclass of the set of variables that get assigned at run-time. The static analysis guarantees that there is an initialization to a local variable on every possible execution path before the variable is read.

In this section, we provide a formal specification of the definite assignment analysis in C$^\sharp$ and prove its correctness. The formal specification emphasizes, in particular, the complications caused by the `goto` and `break` statements (incompletely specified in [74]) and by method calls with `ref`/`out` parameters. The correctness of the analysis is later used in Section 3.6 to prove C$^\sharp_{\mathcal{S}}$'s type safety.

To formally define the definite assignment rules, we use *data flow equations*. For a method body without `goto` statements, the equations that characterize the sets of definitely assigned variables can be solved in a single pass. If `goto` statements are present, then the equations defined in our formalization do not uniquely determine the sets of variables that have to be considered definitely assigned. For this reason, a fixed-point computation is performed, and the greatest sets of variables that satisfy the equations of the formalization are computed. Regarding the correctness of the analysis, we prove that these sets of variables represent exactly the sets of variables assigned on all possible execution paths, and in particular, they are a *safe approximation*. A number of bugs in the Rotor SSCLI (v1.0) [16]'s and Mono (v0.26) [15]'s C$^\sharp$ compilers were discovered during the attempts to build the formalization of the definite assignment. We only present here three of them.

The rest of the section is organized as follows. Section 3.4.1 introduces the data flow equations which formalize the C$^\sharp$ definite assignment analysis, while Section 3.4.2 shows that there always exists a maximal fixed point solution for the equations. In order to define the execution paths in a method body, the control flow graph is introduced in Section 3.4.3. Section 3.4.4 concludes with the proof of the correctness of the analysis, Theorem 3.4.1.

3.4.1 The Data Flow Equations

In this section, we formalize the rules of definite assignment analysis from the [74, §5.3] by data flow equations. Since the definite assignment analysis is an intraprocedural analysis, we restrict our formalization only to a given method *mref*.

To precisely specify all the cases of definite assignment, the functions *before*, *after*, *true*, *false*, and *vars* are computed at compile-time. Note that *true* and *false* are only defined for boolean expressions. These functions assign sets of variables to each expression or statement α and have the following meanings:

- *before*(α) contains the variables definitely assigned before the evaluation of α;
- *after*(α) contains the variables definitely assigned after the evaluation of α when α completes normally;
- *true*(α) consists of the variables (in the scope of which α is located) definitely assigned after the evaluation of α when α evaluates to `true`;
- *false*(α) consists of the variables (in the scope of which α is located) definitely assigned after the evaluation of α when α evaluates to `false`;

The sets *true* and *false* are needed because of the conditional operators `&&` and `||`, as we show in Example 3.4.1. The set *vars*(α) contains the local variables in the scope of which α is located, *i.e.*, the *universal set with respect to* α.

For the sake of clarity, we skip those language constructs whose analysis is very similar to the constructs dealt with explicitly in our framework[7]; examples are the pre- and postfix operators (`++`, `--`).

Struct type variables To simplify the proofs, we will treat separately the struct type variables. We point out in Section 3.4.2 how they affect the sets of definitely assigned variables the C$^\sharp$ compiler relies on in order to analyze programs. Also, we show in Section 3.4.4 that allowing variables of struct types does not affect the correctness of the analysis. In the rest of Section 3.4, we state explicitly whenever we include struct type variables.

We are now able to state all the data flow equations. A first equation is given by the method's initial conditions: For the method body *mb* of *mref*, we have *before*$(mb) = \emptyset$. Conceptually, the set *before*(mb) contains the value and reference parameters of *mref*, since they are assumed to be definitely assigned when *mref* is called [74, §5.1].

[7]Some of the features omitted by the definite assignment analysis presented in this section are investigated in detail in [56, 57].

Expression α	Data flow equations
`true`	$\mathit{true}(\alpha) = \mathit{before}(\alpha), \mathit{false}(\alpha) = \mathit{vars}(\alpha)$
`false`	$\mathit{false}(\alpha) = \mathit{before}(\alpha), \mathit{true}(\alpha) = \mathit{vars}(\alpha)$
$({}^{\beta}exp$ `?` ${}^{\gamma}exp'$ `:` ${}^{\delta}exp'')$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{before}(\gamma) = \mathit{true}(\beta)$, $\mathit{before}(\delta) = \mathit{false}(\beta), \mathit{true}(\alpha) = \mathit{true}(\gamma) \cap \mathit{true}(\delta)$, $\mathit{false}(\alpha) = \mathit{false}(\gamma) \cap \mathit{false}(\delta)$
$({}^{\beta}exp$ `&&` ${}^{\gamma}exp')$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{before}(\gamma) = \mathit{true}(\beta)$, $\mathit{true}(\alpha) = \mathit{true}(\gamma), \mathit{false}(\alpha) = \mathit{false}(\beta) \cap \mathit{false}(\gamma)$
$({}^{\beta}exp$ `||` ${}^{\gamma}exp')$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{before}(\gamma) = \mathit{false}(\beta)$, $\mathit{false}(\alpha) = \mathit{false}(\gamma), \mathit{true}(\alpha) = \mathit{true}(\beta) \cap \mathit{true}(\gamma)$

Table 3.10: Definite assignment for boolean expressions.

For the other expressions and statements in *mb*, instead of explaining how the functions are computed, we simply state the equations they have to satisfy. Table 3.10 contains the equations for boolean expressions (including for completeness the literals `true` and `false`). If α is the constant `true`, then $\mathit{false}(\alpha) = \mathit{vars}(\alpha)$ as a consequence of the definition of the *false* set and of the fact that `true` cannot evaluate to `false`. Similar arguments hold for $\mathit{true}(\alpha) = \mathit{vars}(\alpha)$ when α is the constant `false`. We need the sets *true* and *false* since the evaluation of boolean expressions involving the conditional operators `&&` and `||` does not necessarily require the evaluation of all their subexpressions.

Example 3.4.1 *Consider the following expression:*

$$^{\alpha}(b \texttt{ \&\& } (i = 1) >= 0) \texttt{ ? true : } {}^{\gamma}\, i > 0$$

If b evaluates to `false`*, then the test* $(b$ `&&` $(i = 1) >= 0)$ *immediately evaluates to* `false`*, and its second operand, i.e.,* $(i = 1) >= 0$ *is never evaluated. So, in this case i is not assigned; on the other hand, a necessary condition for the test to be evaluated to* `true` *is that both operands of* α *are evaluated. Therefore, the* C$^\sharp$ *compiler is sure that i is assigned only if* α *evaluates to* `true`*. Formally, this means* $i \notin \mathit{false}(\alpha)$ *and* $i \in \mathit{true}(\alpha)$*. Consequently, i is not considered definitely assigned before evaluating* γ*, and the compiler should reject this example. Surprisingly, the Mono (v0.26) [15]'s* C$^\sharp$ *compiler incorrectly accepts it, as it also does for the the other conditional operator,* `||`.

For all expressions in Table 3.10, we have the equation $\mathit{after}(\alpha) = \mathit{true}(\alpha) \cap \mathit{false}(\alpha)$. For any boolean expression α which is not an instance of one of the expressions in Table 3.10, we have $\mathit{true}(\alpha) = \mathit{after}(\alpha)$ and $\mathit{false}(\alpha) = \mathit{after}(\alpha)$.

Table 3.11 lists the equations specific to arbitrary expressions, where *loc* stands for a local variable and *lit* for a literal. Note that the table contains another equation for the conditional expression when its value is not a boolean. The equation for the explicitly boolean expression collects additional information. If the boolean conditional is treated as an arbitrary expression, then the equation for $\mathit{after}(\alpha)$ would still be correct — it can be derived from the other equations for the boolean expressions.

Expression α	Data flow equations
loc	$after(\alpha) = before(\alpha)$
lit	$after(\alpha) = before(\alpha)$
$(loc = {}^{\beta}e)$	$before(\beta) = before(\alpha)$, $after(\alpha) = after(\beta) \cup \{loc\}$
$(loc\, bop = {}^{\beta}e)$	$before(\beta) = before(\alpha), after(\alpha) = after(\beta)$
$({}^{\beta}exp \ \texttt{?} \ {}^{\gamma}\, exp' \ \texttt{:} \ {}^{\delta}\, exp'')$	$before(\beta) = before(\alpha), before(\gamma) = true(\beta)$, $before(\delta) = false(\beta), after(\alpha) = after(\gamma) \cap after(\delta)$
$T{::}F$	$after(\alpha) = before(\alpha)$
`ref` ${}^{\beta}exp$	$before(\beta) = before(\alpha), after(\alpha) = after(\beta)$
`out` ${}^{\beta}exp$	$before(\beta) = before(\alpha), after(\alpha) = after(\beta)$
$T{::}M({}^{\beta_1}arg_1, \ldots, {}^{\beta_k}\, arg_k)$	$before(\beta_1) = before(\alpha)$, $before(\beta_{i+1}) = after(\beta_i),\ i = 1, k-1$, $after(\alpha) = after(\beta_k) \cup OutParams(arg_1, \ldots, arg_k)$

Table 3.11: Definite assignment for arbitrary expressions.

The `ref` arguments must be definitely assigned before the method call, while the `out` arguments are not necessarily assigned before the method is called. However, the `out` arguments must be definitely assigned when the method returns. Note the equation for $after(\alpha)$ of a method call in Table 3.11: The variables passed as `out` arguments, elements of the set $OutParams(arg_1, \ldots, arg_k)$, get definitely assigned. It does not matter if the calls are recursive, since the definite assignment analysis is an *intraprocedural* analysis.

For expressions that do not appear in Tables 3.10 and 3.11 (*e.g.*, exp `|` exp', exp + exp'), if ${}^{\alpha}exp$ is an expression with *direct subexpressions* ${}^{\beta_1}exp_1, \ldots, {}^{\beta_n}\, exp_n$, then the left-to-right evaluation scheme yields the following *general data flow equations*:

$$before(\beta_1) = before(\alpha), \ \ before(\beta_{i+1}) = after(\beta_i),\ i = 1, n-1, \ \ after(\alpha) = after(\beta_n)$$

The equations specific for statements can be found in Table 3.12. For a block of statements α, we have the equation $after(\alpha) = after(\beta_n) \cap vars(\alpha)$: The local variables definitely assigned after the normal execution of the block are the variables which are definitely assigned after the execution of the last statement of the block. However, the variables must still be in the scope of a declaration.

Example 3.4.2 *Consider the example:*

$$\{{}^{\alpha}\{\texttt{int}\ i;\ i = 1;\}\ \{\texttt{int}\ i;\ i = 2\ *\ {}^{\beta}i;\}\}$$

The variable i is not in $after(\alpha)$, since at the end of α, i is not in the scope of a declaration. Thus, $i \notin before(\beta)$, and the block is rejected.

Statement α	Data flow equations
;	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
$({}^{\beta}\mathit{exp};)$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{after}(\alpha) = \mathit{after}(\beta)$
$\{{}^{\beta_1}\mathit{stm}_1 \ldots {}^{\beta_n}\mathit{stm}_n\}$	$\mathit{before}(\beta_1) = \mathit{before}(\alpha)$, $\mathit{after}(\alpha) = \mathit{after}(\beta_n) \cap \mathit{vars}(\alpha)$, $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap \mathit{goto}(\beta_{i+1})$, $i = 1, n-1$
`if` $({}^{\beta}\mathit{exp})\, {}^{\gamma}\mathit{stm}$ `else` ${}^{\delta}\mathit{stm}'$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{before}(\gamma) = \mathit{true}(\beta)$, $\mathit{before}(\delta) = \mathit{false}(\beta)$, $\mathit{after}(\alpha) = \mathit{after}(\gamma) \cap \mathit{after}(\delta)$
`while` $({}^{\beta}\mathit{exp})\, {}^{\gamma}\mathit{stm}$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{before}(\gamma) = \mathit{true}(\beta)$, $\mathit{after}(\alpha) = \mathit{false}(\beta) \cap \mathit{break}(\alpha)$
`goto` L;	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`break`;	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`continue`;	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`return`;	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`return` ${}^{\beta}\mathit{exp}$;	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`throw`;	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`throw` ${}^{\beta}\mathit{exp}$;	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`try` ${}^{\beta}\mathit{block}$ `catch` $(E_1\ x_1)\ {}^{\gamma_1}\mathit{block}_1$ $\vdots$ `catch`$(E_n\ x_n)\ {}^{\gamma_n}\mathit{block}_n$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$, $\mathit{before}(\gamma_i) = \mathit{before}(\alpha) \cup \{x_i\}, i = 1, n$, $\mathit{after}(\alpha) = \mathit{after}(\beta) \cap \bigcap_{i=1}^{n} \mathit{after}(\gamma_i)$
`try` ${}^{\beta}\mathit{block}$ `finally` ${}^{\gamma}\mathit{block}'$	$\mathit{before}(\beta) = \mathit{before}(\alpha), \mathit{before}(\gamma) = \mathit{before}(\alpha)$, $\mathit{after}(\alpha) = \mathit{after}(\beta) \cup \mathit{after}(\gamma)$

Table 3.12: Definite assignment for statements.

For the equation $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap \mathit{goto}(\beta_{i+1})$, special attention is given to the case when β_{i+1} is a labelled statement. A key point is that if a `goto` embedded in a `try` block (of a `try`-`finally` statement) points to a labelled statement which is not embedded in the `try` block, then the `finally` block has to be executed (before the labelled statement). Thus, the set of variables definitely assigned before executing a labelled statement consists of the variables definitely assigned both after the previous statement and before each corresponding `goto` statement, or after any of the `finally` blocks of `try`-`finally` statements in which the `goto` is embedded (if any).

Definition 3.4.1 *For two statements α and β, $\mathit{Fin}(\alpha, \beta)$ is defined as the list $[\gamma_1, \ldots, \gamma_n]$ of* `finally` *blocks of all* `try`-`finally` *statements in the innermost to outermost order from α*

to β.

Then we define the set *JoinFin* of variables definitely assigned after the execution of all the `finally` blocks in the list *Fin*.

Definition 3.4.2 *For two statements α and β, we define*

$$\boxed{\mathit{JoinFin}(\alpha,\beta) = \bigcup_{\gamma \in \mathit{Fin}(\alpha,\beta)} \mathit{after}(\gamma)}$$

Definition 3.4.3 *Given a statement β, we define:*

$$\mathit{goto}(\beta) = \begin{cases} \bigcap_{\alpha\,\texttt{goto}\ L;\ \text{in the scope of}\ \beta} (\mathit{before}(\alpha) \cup \mathit{JoinFin}(\alpha,\beta)), & \text{if } \beta \text{ is a labelled statement } {}^{\beta}L\ \texttt{:}\ \mathit{stm};\\ \mathit{vars}(\beta), & \text{otherwise}. \end{cases}$$

We are now able to state the equation $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap \mathit{goto}(\beta_{i+1})$ from Table 3.12. In the case of a labelled statement, the equation captures the idea stated above, while for a non-labelled statement, this equations reduces simply to $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$.

Example 3.4.3 *The following example is a simplification of an example in* [74, §5.3.3.15]*:*

```
int i;
δtry
{
    α goto L;
}
finally
γ{
    i = 3;
}
βL : Console.WriteLine(i);
```

[74] *claims that the variable i is definitely assigned before β, i.e., $i \in \mathit{before}(\beta)$. Our equation $\mathit{before}(\beta) = \mathit{after}(\delta) \cap \mathit{goto}(\beta)$ leads us to the same conclusion. To compute the set $\mathit{goto}(\beta)$, we need the list $\mathit{Fin}(\alpha,\beta) = [\gamma]$ and the set $\mathit{JoinFin}(\alpha,\beta) = \mathit{after}(\gamma)$. We have:*

$$\mathit{goto}(\beta) = \mathit{before}(\alpha) \cup \mathit{JoinFin}(\alpha,\beta) = \mathit{before}(\alpha) \cup \mathit{after}(\gamma)$$

and $i \in \mathit{after}(\gamma) \subseteq \mathit{after}(\delta)$ (see the equations for a `try-finally` in Table 3.12). This means that $i \in \mathit{after}(\delta) \cap \mathit{goto}(\beta) = \mathit{before}(\beta)$. Surprisingly, the example is rejected by .NET Framework (v1.0) [2] *and Rotor SSCLI (v1.0)'s C$^\sharp$ compilers: We get the error that i is unassigned. In the meantime, the problem has been fixed in .NET Framework (v1.1)* [2].

Example 3.4.4 *Although in the next method body i should be considered definitely assigned before β, the example is rejected by the Mono (v0.26)* [15]*'s C$^\sharp$ compiler.*

```
int i;
bool b = false;
if (b)
{
    i = 1;
    αgoto L;
}
δ return;
β L : Console.WriteLine(i);
```

We have $i \in \mathit{before}(\beta)$, since the set $\mathit{before}(\beta)$ is computed as follows:

$$\mathit{before}(\beta) = \mathit{after}(\delta) \cap \mathit{goto}(\beta) = \mathit{vars}(\delta) \cap \mathit{before}(\alpha) = \{i, b\} \cap \{i\} = \{i\}$$

The idea for the equation which computes $\mathit{after}(\alpha)$ of a `while` statement α is the same as for a labelled statement. Similarly as for the set *goto*, we define the set $\mathit{break}(\alpha)$ needed for the equation of $\mathit{after}(\alpha)$ to be the set of variables definitely assigned before all associated `break` statements (and possibly after appropriate `finally` blocks).

Definition 3.4.4 *For a `while` statement α, we define*

$$\mathit{break}(\alpha) = \begin{cases} \bigcap_{\beta\texttt{break};\ \alpha \text{ is the nearest enclosing } \texttt{while} \text{ for } \beta} (\mathit{before}(\beta) \cup \mathit{JoinFin}(\beta, \alpha)), & \text{if } \alpha \text{ has } \texttt{break}\text{;s associated;} \\ \mathit{vars}(\alpha), & \text{otherwise.} \end{cases}$$

With this definition of $\mathit{break}(\alpha)$, we have the equation for $\mathit{after}(\alpha)$ as stated in Table 3.12.

Abnormal termination Finally, we consider the equations for abnormally terminating statements. We now want to state the equation for *after* of a jump statement.

Example 3.4.5 *Let α be the following statement:*

$$\texttt{if}\ (b)\ {}^{\gamma}\{\, i = 1; \}\ \texttt{else}\ {}^{\delta}\texttt{return};$$

It is clear that the variables definitely assigned after α are the variables definitely assigned after the `then` branch, and since our equation takes the intersection of $\mathit{after}(\gamma)$ and $\mathit{after}(\delta)$, it is obvious that one has to require the set-intersection identity for $\mathit{after}(\delta)$.

Considerations of the example above lead to the convention that $\mathit{after}(\alpha)$ should be considered the universal set $\mathit{vars}(\alpha)$ for any jump statement α.

Example 3.4.6 *Consider the next method body:*

```
^δint i = 1;
^βL : ;
try
{
    int j = 2;
    try { ^αgoto L; }
    finally ^γ1 { int k = 3; ^ωthrow new Exception(); }
} finally ^γ2 {}
```

In our formalization, we get:

$$\begin{aligned} \mathit{before}(\beta) &= \mathit{after}(\delta) \cap \mathit{goto}(\beta) \\ &= \mathit{after}(\delta) \cap (\mathit{before}(\alpha) \cup \mathit{JoinFin}(\alpha,\beta)) \\ &= \mathit{after}(\delta) \cap (\mathit{before}(\alpha) \cup \mathit{after}(\gamma_1) \cup \mathit{after}(\gamma_2)) \end{aligned}$$

Note that in the computation of JoinFin(α, β)*, it does not matter whether* γ_1 *and* γ_2 *complete normally. In our case,* γ_1 *does not complete normally, but we still perform our computations with* γ_1 *and* γ_2*. The set after*(γ_1) *is vars*$(\omega) \cap$ *vars*$(\gamma_1) = \{i,j,k\} \cap \{i,j\} = \{i,j\}$*. Note also that after*$(\gamma_1)$ *involved in the equation of before*(β) *also contains j, while* β *is not in the scope of a declaration of j, i.e., j* $\notin$ *vars*(β)*. There is no worry since in the equation of before*(β) *all the sets that might contain variables declared in "deeper" scopes (like j) are intersected with after*(δ)*, which is supposed to contain only variables from vars*$(\delta) =$ *vars*$(\beta) = \{i\}$ *(*β *and* δ *are at the same nesting level).*

The details of Examples 3.4.5 and 3.4.6 become more clear in Lemmas 3.4.4 and 3.4.5 in Section 3.4.4. Whenever a statement does not complete normally, the set of variables considered definitely assigned after its evaluation will be a universal set (see also the proofs of Theorems 3.4.1 and 3.4.2 from Section 3.4.4).

3.4.2 The Maximal Fixed Point Solution

The computation of the sets of definitely assigned variables from the data flow equations described in Section 3.4.1 is relatively straightforward. However, the `goto` statement brings more complexity to the analysis. Since the `goto` statement allows to encode loops, the system of data flow equations does not have always a unique solution.

Example 3.4.7 *Consider a method (with no parameters) that has the following body:*

```
{^αint i = 1; ^βL: ^γgoto L; }
```

We have the following equations after$(\alpha) = \{i\}$*, before*$(\beta) =$ *after*$(\alpha) \cap$ *before*(γ) *and before*$(\gamma) =$ *before*(β)*. After some simplification, we find that before*$(\beta) = \{i\} \cap$ *before*(β)*. Therefore, we get two solutions for before*(β) *(and also for before*(γ)*):* $\emptyset$ *and* $\{i\}$*. This is the reason we perform a fixed point iteration. The set of variables definitely assigned after* α *is* $\{i\}$*; since* β *does not "unassign" i, i is obviously assigned when we enter* β*.*

The above example and the definition of *definitely assigned* indicate that the most informative solution is considered, and therefore the solution we require is the *maximal fixed point MFP*. For computing this solution, various algorithms exist (see *e.g.*, [97]).

Remark 3.4.1 *Although the statements L : `goto` L; and `while` (`true`); are behaviorally similar, they are treated differently by the definite assignment analysis. If α denotes the labelled statement, then the equation for before(α) implies recursion (as noticed above). If α is the `while` statement above, then no equation corresponding to α involves recursion. The set after(α) of the above `while` statement can be computed according to the equations in a single step (i.e., with no fixed point iteration) as follows after(α) = false$(\alpha) \cap$ break(α) = vars$(\alpha) \cap$ vars(α) = vars(α). The set before(α) is determined as for any regular statement using only the after set of the previous statement. Even if a `while` statement α has an associated `continue` statement γ, the equation for before(α) does not involve the `continue`, but only the previous statement β. The reason is that, at the time the analysis is performed, the compiler is sure that the `continue` is embedded in the `while` body, and therefore the set before(γ) includes the variables in after(β) (if the `continue` is executed, then β should have already been executed). This is not always the case for a labelled statement since the associated `goto`s are not necessarily embedded in the labelled block (see the last example of Section 3.4.1). That is why they are involved in the equation for before(α).*

In the rest of this section, we show that there always exists a maximal fixed point for our data flow equations. In order to prove its existence, we first define the function F, which encapsulates the equations. For the domain and codomain of this function, we use the set *localVars*$(mref)$ of all local variables from the method body *mb*. A simple inspection of the equations shows that they all have at the left side either a *before*, *after*, *true* or a *false* set and at the right side a combination of these kinds of sets and *vars* sets. We define the function F : $Map(D, D)$, with $D = \mathcal{P}(localVars(mref))^r$, such that $F(X_1, \ldots, X_r) = (Y_1, \ldots, Y_r)$, where r is the number of equations and the sets Y_i are defined by the data flow equations, with the *vars* sets interpreted as constants. For example, in the case of an `if`-`then`-`else` statement, if the equation for the *after* set of this statement is the i-th data flow equation, then the set of variables Y_i is defined by $Y_i = X_j \cap X_k$, where j and k are the indices of the equations for the *after* sets of the `then` and the `else` branch, respectively.

We now define the relation $\ll$ on D to be the pointwise set inclusion relation:

Definition 3.4.5 *If $(X_1, \ldots, X_r)$ and $(X_1', \ldots, X_r')$ are elements of the set D, then we have $(X_1, \ldots, X_r) \ll (X_1', \ldots, X_r')$, if $X_i \subseteq X_i'$ for all $i = 1, r$.*

We are now able to prove the following result:

Lemma 3.4.1 *$(D, \ll)$ is a finite lattice.*

Proof. D is finite since for a given method body we have a finite number of equations and local variables. D is a lattice since it is a product of lattices: $(\mathcal{P}(localVars(mref)), \subseteq)$ is a *poset*, since the set inclusion is a partial order, and for every two sets $X, Y \in \mathcal{P}(localVars(mref))$, there exists a lower bound (*e.g.*, $X \cap Y$) and an upper bound (*e.g.*, $X \cup Y$). □

The following result will help us conclude the existence of the maximal fixed point.

Lemma 3.4.2 *The function F is monotonic on $(D, \ll)$.*

Proof. To prove the monotonicity of $F = (F_1, \ldots, F_r)$, it suffices to observe that the components F_i are monotonic functions. This holds since they only consist of set intersections and unions, which are monotonic (see the form of the equations). $\square$

The next lemma guarantees the existence of the maximal fixed point solution for our data flow equations.

Lemma 3.4.3 *The function F has a unique maximal fixed point MFP* $\in D$.

Proof. By Lemma 3.4.1, we know that $(D, \ll)$ is a finite lattice, and therefore a complete lattice. But in a complete lattice, every monotonic function has a unique maximal fixed point, also known as *the greatest fixed point*). In our case, F is monotonic (by Lemma 3.4.2), and the maximal fixed point *MFP* is given by $\bigcap_k F^{(k)}(1_D)$. Here, 1_D is the r-tuple $(\mathit{localVars}(\mathit{mref}), \ldots, \mathit{localVars}(\mathit{mref}))$, *i.e.*, the *top element* of the lattice D. $\square$

From now on, for an expression or statement α, we denote by $\mathrm{MFP}_b(\alpha)$, $\mathrm{MFP}_a(\alpha)$, $\mathrm{MFP}_t(\alpha)$, and $\mathrm{MFP}_f(\alpha)$ the components of the maximal fixed point *MFP* corresponding to *before*(α), *after*(α), *true*(α), and *false*(α), respectively.

Struct type variables So far, we have not considered struct type variables. However, our analysis can easily be extended to struct type variables. Firstly, we observe that a struct type variable is considered definitely assigned if and only if all its instance fields are definitely assigned [74, §5.3] (this is because a struct value can be seen as a tuple consisting of its instance fields).

If a local variable *loc*, whose declared type is a struct, gets assigned, then all its instance fields are considered definitely assigned, and if there are struct type instance fields, then their instance fields get assigned as well, and so on.

Example 3.4.8 *Let us consider an instance method C::M, which takes an* `out` *parameter of type the struct S. The instance field y of p.x is definitely assigned before it is printed, since p gets definitely assigned after the method call.*

```
struct S {                    class Test {
    public S' x;                  static void Main() {
}                                         S p;
                                          new C().M(out p);
struct S' {                               Console.WriteLine(p.x.y);
    public int y;                 }
}                             }
```

According to our formalization, the C$^\sharp$ *compiler relies on the set* $\mathrm{MFP}_a(\alpha) = \mathit{after}(\alpha) = \{p\}$, *with* ${}^{\alpha}$(`new` *C*().*M*(`out` *p*)), *when checking the status of the variable p.x.y. But, as observed above, after* α *is evaluated, p.x and p.x.y are considered definitely assigned as well. So, allowing struct type variables requires the compiler to rely on "expanded" sets: The set* $\{p\}$ *is "expanded" to* $\{p, p.x, p.x.y\}$, *to also include the instance fields of p.*

Further, if a local variable *loc'*, which is an instance field of a struct type variable *loc*, gets assigned, and each instance field of *loc* except *loc'* either is already considered definitely assigned or gets assigned at the same time as *loc'* (this happens, for example, in case of the `out` arguments following a method call), then the variable *loc* gets assigned as well, and this "lookup" procedure is repeated with *loc* instead of *loc'*.

Example 3.4.9 *In the following example, the struct type field p.y gets assigned when its instance field v gets assigned. Next, the instance variable p gets assigned when its instance field p.x gets assigned since p.y is already assigned.*

```
struct S {                        class Test {
    public int x;                     static void Main() {
    public S' y;                          S p;
}                                         p.y.u = 1;
                                          p.y.v = 1;
struct S' {                               p.x = 1;
    public int u;                         S r = p;
    public int v;                     }
}                                 }
```

According to the formalization, the compiler relies on the set $\mathrm{MFP}_a(\alpha) = \mathit{after}(\alpha) = \{p.y.u, p.y.v, p.x\}$ *with* ${}^\alpha(p.x = 1)$ *when verifying whether p is definitely assigned before evaluating the assignment to r. Considering also struct type variables makes the compiler rely on the expanded set of* $\{p.y.u, p.y.v, p.x\}$*, i.e.,* $\{p.y.u, p.y.v, p.y, p.x, p\}$.

Hence, we conclude that after the C$^\sharp$ compiler determines the MFP sets, it relies on the expanded MFP sets in order to reject/accept programs. We will say that an "expansion" function is applied to the MFP sets to *propagate* the *definitely assigned* status.

3.4.3 The Control Flow Graph

So far, we have seen the equations used for the analysis and we have proven that the fixed point iteration for these equations is well-defined. The result we want to prove is that the outcome of the analysis is correct: For an arbitrary expression or statement, the sets of local variables MFP_b, MFP_a (and MFP_t, MFP_f for boolean expressions) correspond indeed to sets of *definitely assigned* variables, *i.e.*, variables which are assigned on every possible execution path to the appropriate point.

The considered paths are based on the control flow graph *CFG*. The nodes of the graph are actually points associated with every expression and statement. We suppose that every expression or statement α is characterized by an *entry point* $\mathcal{B}(\alpha)$ and an *end point* $\mathcal{A}(\alpha)$. Beside these two points a boolean expression α has two more points: a *true point* $\mathcal{T}(\alpha)$ (used when α evaluates to `true`) and a *false point* $\mathcal{F}(\alpha)$ (used when α evaluates to `false`). The edges of the graph are given by the *control transfer* defined in [74, §8]. Tables 3.13 and 3.14 show the edges specific to each boolean and arbitrary expression, respectively. If the expression α is not an instance of one expression in these tables (*e.g.*, *exp* `|` *exp'*) and has the *direct subexpressions* $\beta_1, \ldots, \beta_n$, then the left-to-right evaluation scheme also adds the following edges to the flow graph:

Expression α	Edges
`true`	$(\mathcal{B}(\alpha), \mathcal{T}(\alpha))$
`false`	$(\mathcal{B}(\alpha), \mathcal{F}(\alpha))$
$({}^{\beta}exp$ `?` ${}^{\gamma}exp'$ `:` ${}^{\delta}exp'')$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta)),$ $(\mathcal{T}(\gamma), \mathcal{T}(\alpha)), (\mathcal{T}(\delta), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha)),$ $(\mathcal{F}(\delta), \mathcal{F}(\alpha))$
$({}^{\beta}exp$ `&&` ${}^{\gamma}exp')$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{F}(\alpha)),$ $(\mathcal{T}(\gamma), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha))$
$({}^{\beta}exp$ `||` ${}^{\gamma}exp')$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{T}(\alpha)), (\mathcal{F}(\beta), \mathcal{B}(\gamma)),$ $(\mathcal{T}(\gamma), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha))$

Table 3.13: Control flow for boolean expressions.

Expression α	Edges
loc	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
lit	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$(loc = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$(loc\ \ bop = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$({}^{\beta}exp$ `?` ${}^{\gamma}exp'$ `:` ${}^{\delta}exp'')$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta)),$ $(\mathcal{A}(\gamma), \mathcal{A}(\alpha)), (\mathcal{A}(\delta), \mathcal{A}(\alpha))$
$T{::}F$	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
`ref` ${}^{\beta}exp$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
`out` ${}^{\beta}exp$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$T{::}M({}^{\beta_1}arg_1, \ldots, {}^{\beta_k}arg_k)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_k), \mathcal{A}(\alpha)),$ $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = 1, k-1$

Table 3.14: Control flow for arbitrary expressions.

$$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), \;\; (\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), \; i = 1, n-1, \text{ and } (\mathcal{A}(\beta_n), \mathcal{A}(\alpha))$$

For every boolean expression α in Table 3.13, we define the supplementary edges $(\mathcal{T}(\alpha), \mathcal{A}(\alpha))$ and $(\mathcal{F}(\alpha), \mathcal{A}(\alpha))$, which connect the boolean points of α to the *end point* of α. These edges are necessary for control transfer in cases when it does not matter whether α evaluates to `true` or `false`.

Example 3.4.10 *If β is the method call $T{::}M($*`true`*$)$ and α is the argument* `true`*, then control is transferred from the* end point *of the last argument — that is, $\mathcal{A}(\alpha)$ — to the* end point *of the method call — that is, $\mathcal{A}(\beta)$. But, since in Table 3.13 we have no edge leading to $\mathcal{A}(\alpha)$, we need to define the supplementary edge $(\mathcal{T}(\alpha), \mathcal{A}(\alpha))$.*

For a boolean expression α which is not an instance of any expression from Table 3.13, we add the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$ to the graph. They are needed if control is transferred from a boolean expression α to different points depending on whether α evaluates to `true` or `false`.

Example 3.4.11 *If α is of the form exp `|` exp′ and occurs in ((exp `|` exp′) `?` $^{\beta}$exp″ `:` $^{\gamma}$exp‴), then control is transferred from $\mathcal{F}(\alpha)$ to $\mathcal{B}(\beta)$ (if α evaluates to `false`) or from $\mathcal{T}(\alpha)$ to $\mathcal{B}(\gamma)$ (if α evaluates to `true`). The necessity of the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$ arises since so far we have defined for exp `|` exp′ only edges to $\mathcal{A}(\alpha)$.*

Table 3.15 introduces the edges of the control flow graph for each statement. We assume that the boolean constant expressions are replaced by `true` or `false` in the abstract syntax tree.

Example 3.4.12 *Thus, we consider that `true ||` b is replaced by `true` in the following `if` statement:*

```
αif β(true || b) δi = 1;
else γ{ int j = i;}
```

Although the new test, i.e., `true`, cannot evaluate to `false`, we still add to the graph the edge $(\mathcal{F}(\beta), \mathcal{B}(\gamma))$: The false point *of `true` is, however, never reachable (see Table 3.13).*

In the presence of `finally` blocks, the jump statements `goto`, `continue` and `break` bring more complexity to the graph. When a jump statement exits a `try` block, control is transferred first to the innermost `finally` block. If control reaches the *end point* of that `finally` block, then it is transferred to the next innermost `finally` block and so on. If control reaches the *end point* of the outermost `finally` block, then it is transferred to the target of the jump statement. For these control transfers, we have special edges in our graph. But one needs to take care of some important details: These special edges cannot be used for paths other than those which connect the jump statement with its target. In other words, if a path uses such an edge, then the path necessarily contains the *entry point* of the jump statement. For this reason, we say that an edge e is *conditioned* by a point *i* with the meaning that e can be used only in paths that contain *i*.

Example 3.4.13 *Let us assume that we do not institute the above restriction. Then the list $[\mathcal{B}(mb)\mathcal{B}(\alpha_1)\mathcal{B}(\alpha_2)\mathcal{B}(\alpha_3)\mathcal{B}(\alpha_4)\mathcal{B}(\alpha_5)\mathcal{A}(\alpha_5)\mathcal{B}(\alpha_6)]$ would be a possible execution path to the labelled statement in the following method body*

```
α1try α2 {
        α3(α4 (i = 3 / j);)
        αgoto L;
} finally α5 {}
α6 L : Console.WriteLine(i);
```

in case the evaluation of α_4 would throw an exception. But, this does not match the control transfer described in [74].

Statement α	Edges
;	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$({}^{\beta}exp;)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$\{{}^{\beta_1}stm_1 \ldots {}^{\beta_n}stm_n\}$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_n), \mathcal{A}(\alpha))$, $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = 1, n-1$
`if` $({}^{\beta}exp)\ {}^{\gamma}stm$ `else` ${}^{\delta}stm'$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta))$, $(\mathcal{A}(\gamma), \mathcal{A}(\alpha)), (\mathcal{A}(\delta), \mathcal{A}(\alpha))$
`while` $({}^{\beta}exp)\ {}^{\gamma}stm$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{A}(\alpha))$, $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$
$L\ :\ {}^{\beta}stm$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
`goto` L;	$ThroughFin_b(\alpha, \beta)$, where ${}^{\beta}L\ :\ stm$ is the statement to which α points
`break`;	$ThroughFin_a(\alpha, \beta)$, where β is the nearest enclosing `while` with respect to α
`continue`;	$ThroughFin_b(\alpha, \beta)$, where β is the nearest enclosing `while` with respect to α
`return`;	–
`return` ${}^{\beta}exp$;	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$
`throw`;	–
`throw` ${}^{\beta}exp$;	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$
`try` ${}^{\beta}block$ `catch`$(E_1\ x_1)\ {}^{\gamma_1}block_1$ $\vdots$ `catch`$(E_n\ x_n)\ {}^{\gamma_n}block_n$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$ $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_i)), (\mathcal{A}(\gamma_i), \mathcal{A}(\alpha)),\ i = 1, n$
`try` ${}^{\beta}block$ `finally` ${}^{\gamma}block'$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$,$(\mathcal{B}(\alpha), \mathcal{B}(\gamma))$, $(\mathcal{A}(\beta), \mathcal{B}(\gamma))$ and $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ conditioned by $\mathcal{A}(\beta)$

Table 3.15: Control flow for statements.

The following sets introduce the above described edges. If α and β are two statements and $Fin(\alpha, \beta)$ is the list $[\gamma_1, \ldots, \gamma_n]$, then the set $ThroughFin_b(\alpha, \beta)$ consists of the edges $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_1))$, $(\mathcal{A}(\gamma_n), \mathcal{B}(\beta))$, $(\mathcal{A}(\gamma_i), \mathcal{B}(\gamma_{i+1}))$, $i = 1, n-1$, all conditioned by $\mathcal{B}(\alpha)$, while the set $ThroughFin_a(\alpha, \beta)$ contains the edges $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_1))$, $(\mathcal{A}(\gamma_n), \mathcal{A}(\beta))$, $(\mathcal{A}(\gamma_i), \mathcal{B}(\gamma_{i+1}))$, $i = 1, n-1$, all conditioned by $\mathcal{B}(\alpha)$. If $Fin(\alpha, \beta)$ is empty, then the set $ThroughFin_b(\alpha, \beta)$ contains only the edge $(\mathcal{B}(\alpha), \mathcal{B}(\beta))$, while $ThroughFin_a(\alpha, \beta)$ refers to the edge $(\mathcal{B}(\alpha), \mathcal{A}(\beta))$.

Example 3.4.14 *Consider again the* `try-finally` *statement from Example 3.4.13. The list*

Fin(α, α_6) *is given by* $[\alpha_5]$*, while the set ThroughFin*$_b(\alpha, \alpha_6)$ *contains the edges* $(\mathcal{B}(\alpha), \mathcal{B}(\alpha_5))$*,* $(\mathcal{A}(\alpha_5), \mathcal{B}(\alpha_6))$ *conditioned by* $\mathcal{B}(\alpha)$*.*

Note that in Table 3.15, for `goto` and `continue`, the set of edges *ThroughFin*$_b$ is added to the graph, since after executing the `finally` blocks control is transferred to the *entry point* of the labelled statement and `while` statement, respectively. However, in case of `break` the set *ThroughFin*$_a$ is considered, since at the end, control is transferred to the *end point* of the `while` statement.

There are two more remarks concerning the `try` statement. First of all, since a reason for abruption (*e.g.*, an exception) can occur anytime in a `try` block, we should have edges from every point in a `try` block to: every associated `catch` block, every `catch` of enclosing `try` statements (if the `catch` clause matches the type of the exception) and to every associated `finally` block (if none of the `catch` clauses matches the type of the exception). We do not consider all these edges since the definite assignment analysis is an "over all paths" analysis. It is equivalent to consider only one edge to the *entry points* of the `catch` and `finally` blocks — from the *entry point* of the `try` block (see Table 3.15).

The following remark concerns the *end point* $\mathcal{A}(\alpha)$ of a `try`-`finally` statement α. [74, §8.10] states that $\mathcal{A}(\alpha)$ is reachable only if both *end points* of the `try` block β and `finally` block γ are reachable. The only edge to $\mathcal{A}(\alpha)$ is $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$, and we know that the `finally` block can be reached either through a jump or through a normal completion of the `try` block. In the case of a jump, if control reaches the *end point* $\mathcal{A}(\gamma)$ of the `finally`, then it is transferred further to the target of the statement which generated the jump and not to $\mathcal{A}(\alpha)$. This means that all paths to $\mathcal{A}(\alpha)$ contain also the *end point* $\mathcal{A}(\beta)$ of the `try` block. That is why we require that the edge $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ is *conditioned* by $\mathcal{A}(\beta)$ (see Table 3.15).

Example 3.4.15 *If the edge* $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ *is* not *conditioned by* $\mathcal{A}(\beta)$*, then* $\mathcal{A}(\alpha)$ *would be reachable in our specifications in the following example (under the assumption that* $\mathcal{B}(\alpha)$ *is reachable):*

$${}^{\alpha}\,\mathbf{try}\ {}^{\beta}\{\mathbf{goto}\ L;\ \}\ \mathbf{finally}\ {}^{\gamma}\ \{\}$$

We will not consider all the paths in the graph, but only the *valid* paths, *i.e.*, the paths p for which the following condition is fulfilled: If p uses a *conditioned* edge, then p also includes the point which conditions the edge.

Definition 3.4.6 *For a path* $[\alpha_1, \ldots, \alpha_n]$*, we define*

$$\begin{array}{l} \mathit{valid}([\alpha_1, \ldots, \alpha_n]) :\Leftrightarrow \\ \quad \forall \text{ conditioned edge } (\alpha_i, \alpha_{i+1})\ \exists\, j < i\ :\ (\alpha_i, \alpha_{i+1}) \text{ is } \textit{conditioned} \text{ by } \alpha_j \end{array}$$

If α is an expression or a statement, then $path_b(\alpha)$ is the set of all valid paths from the *entry point* of the method body $\mathcal{B}(mb)$ to the *entry point* $\mathcal{B}(\alpha)$ of α:

$$\begin{array}{rcl} path_b(\alpha) & = & \{[\alpha_1, \ldots, \alpha_n] \mid \alpha_1 = \mathcal{B}(mb), \alpha_n = \mathcal{B}(\alpha), (\alpha_i, \alpha_{i+1}) \in CFG, i = 1, n-1 \\ & & \text{and } \mathit{valid}([\alpha_1, \ldots, \alpha_n])\} \end{array}$$

Similarly, $path_a(\alpha)$ is defined to be the set of all valid paths from the *entry point* of the method body $\mathcal{B}(mb)$ to the *end point* $\mathcal{A}(\alpha)$ of α, while if α is a boolean expression, $path_t(\alpha)$ and $path_f(\alpha)$ are the sets of all valid paths from $\mathcal{B}(mb)$ to the *true point* $\mathcal{T}(\alpha)$ and to the *false point* $\mathcal{F}(\alpha)$ of α, respectively.

In the proofs in the next section, we use the following two notations. If p is a path, then $p[i,j]$ is the subpath of p, which connects the point i with the point j. Over the set of all paths, we consider the operation $\oplus$ to be the path concatenation (also defined for infinite paths).

3.4.4 The Correctness of the Analysis

We prove that when a C$^\sharp$ compiler relies on the sets MFP_b, MFP_a, MFP_t, and MFP_f, derived from the maximal fixed point of the data flow equations in Section 3.4.1 (or on their expanded sets if we allow struct type variables), then all accesses to the value of a local variable occur after the variable is initialized. In other words, the correctness of the analysis means that if a local variable is in one of the four sets — that is the analysis infers the variable as definitely assigned at a certain program point — then this variable will actually be assigned at that point during every execution path of the program. A variable *loc* is *assigned* on a path if the path contains an *initialization* of *loc* or a `catch` clause, whose exception variable is *loc*. We describe in the following definition what we mean by *initialization*.

Definition 3.4.7 *A path p contains an* initialization *of a local variable loc if at least one of the following conditions is true:*

- *p contains a simple assignment (as opposed to a compound assignment) to loc, or*
- *p contains a method call, for which loc is an* `out` *argument.*

Struct type variables The definition above has to be refined if we want to allow variables of struct types. Thus, a path p contains an *initialization* of a variable *loc also* in one of the following cases:

- *loc* is an instance field of a struct type variable *loc*$'$, and p contains an *initialization* of *loc*$'$, or
- *loc* is of a struct type, and p contains *initializations* for each instance field of *loc*.

We actually prove more than the correctness. We show that the components of the maximal fixed point *MFP* are exactly the sets of variables which are assigned on *every possible execution path* to the appropriate point (and not only a *safe-approximation*). To specify this, we define the following sets. If α is an arbitrary expression or statement, then $\mathrm{AP}_b(\alpha)$ denotes the set of local variables in $vars(\alpha)$ (the variables in the scope of which α is) assigned on every path in $path_b(\alpha)$:

$$\mathrm{AP}_b(\alpha) = \{x \in \mathit{vars}(\alpha) \mid x \text{ is } \mathit{assigned} \text{ on every path } p \in \mathit{path}_b(\alpha)\}$$

$\mathrm{AP}_a(\alpha)$ is the set of variables in $\mathit{vars}(\alpha)$, which are assigned on every path in $\mathit{path}_a(\alpha)$, while for a boolean expression α, the sets $\mathrm{AP}_t(\alpha)$ and $\mathrm{AP}_f(\alpha)$ are defined similarly as above, but with respect to paths in $\mathit{path}_t(\alpha)$ and $\mathit{path}_f(\alpha)$, respectively.

Struct type variables If we also consider variables of struct types, the definition of "is *assigned* on" is extended as pointed out above. The definitions of the sets AP_b, AP_a, AP_t and AP_f are also adapted. But considering the new definition for "is *assigned*", one can easily observe that the definitions of the AP sets to include struct type variables are nothing else than their expanded sets. So, the same "expansion" function mentioned in Section 3.4.2 is also applied to the AP sets in order to include struct type variables.

The following result is used to prove Lemma 3.4.5.

Lemma 3.4.4 *For every expression or statement α, if* $\mathrm{MFP}_b(\alpha) \subseteq \mathit{vars}(\alpha)$ *holds, then we have* $\mathrm{MFP}_a(\alpha) \subseteq \mathit{vars}(\alpha)$. *Moreover, if α is a boolean expression, then we have also* $\mathrm{MFP}_t(\alpha) \subseteq \mathit{vars}(\alpha)$ *and* $\mathrm{MFP}_f(\alpha) \subseteq \mathit{vars}(\alpha)$.

Proof. The proof proceeds by induction over the structure of expressions and statements. Thus, we first prove the base cases of the induction, *i.e.*, the above stated implications for all possible leaves of the AST of our method body. The expressions which are leaves in the AST are the following: `true`, `false`, *loc*, *lit*, and $T{::}F$. Since MFP is in particular a solution of the data flow equations, it is obvious that the implications stated in our lemma are satisfied. The statements considered leaves in the AST are the *empty-statement*, `goto` *L*, `break`, `continue`, `return`, and `throw`. For the last five, from the equations above we obviously have $\mathrm{MFP}_a(\alpha) \subseteq \mathit{vars}(\alpha)$. For the *empty-statement*, this is true as well since our hypothesis is $\mathrm{MFP}_b(\alpha) \subseteq \mathit{vars}(\alpha)$.
In the induction step, the implications for each expression and statement are proved under the assumption that their "children" (subexpressions/substatements) satisfy the implications. □

The next lemma is used in the proof of the correctness theorem (Theorem 3.4.1). It claims that the MFP sets of an expression or statement α consist of variables in the scope of which α is located.

Lemma 3.4.5 *For every expression or statement α, we have* $\mathrm{MFP}_b(\alpha) \subseteq \mathit{vars}(\alpha)$ *and* $\mathrm{MFP}_a(\alpha) \subseteq \mathit{vars}(\alpha)$. *Moreover, if α is a boolean expression, then we have* $\mathrm{MFP}_t(\alpha) \subseteq \mathit{vars}(\alpha)$ *and* $\mathrm{MFP}_f(\alpha) \subseteq \mathit{vars}(\alpha)$.

Proof. We show the above inclusions for all expressions and statements by an induction over the AST, starting at the root, *i.e.*, the method body (the base case of the induction). Notice that the induction schema is in the opposite direction compared to that in Lemma 3.4.4. Therefore, the induction step is: Under the assumption that a node of the AST satisfies the inclusions, all its "children" (subexpressions/substatements) satisfy the inclusions as well.

According to Lemma 3.4.4, it is enough to prove for all labels α: $\mathrm{MFP}_b(\alpha) \subseteq \mathit{vars}(\alpha)$. For our method body, this is trivial: The relation $\mathrm{MFP}_b(\mathit{mb}) \subseteq \mathit{vars}(\mathit{mb})$ holds since $\mathrm{MFP}_b(\mathit{mb}) = \mathit{vars}(\mathit{mb}) = \emptyset$. Lemma 3.4.4 is applied again in the next step of the proof, which consists in showing for each expression and statement that under the assumption $\mathrm{MFP}_b(\alpha) \subseteq \mathit{vars}(\alpha)$, each of its direct subexpressions/substatements β satisfies $\mathrm{MFP}_b(\beta) \subseteq \mathit{vars}(\beta)$. □

The correctness of the definite assignment analysis in C$^\sharp$ is proved in the following theorem, which claims that the analysis is a *safe approximation*.

Theorem 3.4.1 (Definite assignment analysis correctness) *If α is an expression or a statement, then the following relations are true:* $\mathrm{MFP}_b(\alpha) \subseteq \mathrm{AP}_b(\alpha)$ *and* $\mathrm{MFP}_a(\alpha) \subseteq \mathrm{AP}_a(\alpha)$. *Moreover, if α is a boolean expression, then* $\mathrm{MFP}_t(\alpha) \subseteq \mathrm{AP}_t(\alpha)$ *and* $\mathrm{MFP}_f(\alpha) \subseteq \mathrm{AP}_f(\alpha)$.

Proof. We consider the following definitions. The set $\mathrm{AP}_b^n(\alpha)$ is defined in the same way as $\mathrm{AP}_b(\alpha)$, except that we consider only paths of length less than n. Similarly, we also define the sets $\mathrm{AP}_a^n(\alpha)$, $\mathrm{AP}_t^n(\alpha)$, $\mathrm{AP}_f^n(\alpha)$ (analogously, we have definitions for the sets of paths path_b^n, path_a^n, path_t^n, path_f^n). According to these definitions, the following set equalities hold for every α:

$$\mathrm{AP}_b(\alpha) = \bigcap_n \mathrm{AP}_b^n(\alpha), \ \mathrm{AP}_a(\alpha) = \bigcap_n \mathrm{AP}_a^n(\alpha)$$

If α is a boolean expression, then similar equalities hold for $\mathrm{AP}_t(\alpha)$ and $\mathrm{AP}_f(\alpha)$.

Therefore, to complete the proof, it suffices to show for every n: If α is an expression or statement, then $\mathrm{MFP}_b(\alpha) \subseteq \mathrm{AP}_b^n(\alpha)$ and $\mathrm{MFP}_a(\alpha) \subseteq \mathrm{AP}_a^n(\alpha)$. In addition, if α is a boolean expression, then $\mathrm{MFP}_t(\alpha) \subseteq \mathrm{AP}_t^n(\alpha)$ and $\mathrm{MFP}_f(\alpha) \subseteq \mathrm{AP}_f^n(\alpha)$. This is done by induction on n.

Base case $[\mathcal{B}(\mathit{mb})]$ is the only path of length 1 (the *entry point* of the method body). Obviously, no local variable is assigned on this path and therefore we have $\mathrm{AP}_b^1(\mathit{mb}) = \emptyset$ which satisfies $\mathrm{MFP}_b(\mathit{mb}) \subseteq \mathrm{AP}_b^1(\mathit{mb})$ since from the equations $\mathrm{MFP}_b(\mathit{mb}) = \emptyset$. From the definition of AP_a^1, we get $\mathrm{AP}_a^1(\mathit{mb}) = \mathit{vars}(\mathit{mb}) = \emptyset$ and from the equations of a block, we derive also $\mathrm{MFP}_a(\mathit{mb}) \subseteq \mathit{vars}(\mathit{mb})$ and implicitly $\mathrm{MFP}_a(\mathit{mb}) \subseteq \mathrm{AP}_a^1(\mathit{mb})$. If $\alpha \neq \mathit{mb}$, then $\mathrm{AP}_b^1(\alpha) = \mathrm{AP}_a^1(\alpha) = \mathit{vars}(\alpha)$ and if α is a boolean expression $\mathrm{AP}_t^1(\alpha) = \mathrm{AP}_f^1(\alpha) = \mathit{vars}(\alpha)$. If we apply Lemma 3.4.5, then the basis of the induction is complete.

Induction step The proof has the following pattern: We show for every expression or statement α from Tables 3.10, 3.11, and 3.12 the relation for $\mathrm{MFP}_a(\alpha)$, and where applicable for $\mathrm{MFP}_t(\alpha)$ and $\mathrm{MFP}_f(\alpha)$ and, for every direct subexpression/substatement of α, the relations for MFP_b. In this way, all the relations for all expressions/statements are proved except $\mathrm{MFP}_b(\mathit{mb}) \subseteq \mathrm{AP}_b^{n+1}(\mathit{mb})$ (since *mb* has no "superstatement") which holds anyway, since $\mathrm{MFP}_b(\mathit{mb}) = \emptyset$.

We only consider two critical cases: the block of statements and the `try`-`finally` statement.

Case 1 α is $\{{}^{\beta_1}\mathit{stm}_1 \ldots {}^{\beta_n}\mathit{stm}_n\}$.

We prove $\mathrm{MFP}_b(\beta_{i+1}) \subseteq \mathrm{AP}_b^{n+1}(\beta_{i+1})$ for an embedded statement β_{i+1}. If we arbitrarily choose a local variable *loc* in $\mathrm{MFP}_b(\beta_{i+1})$, then we obtain $\mathit{loc} \in \mathrm{MFP}_a(\beta_i)$ and $\mathit{loc} \in \mathit{goto}(\beta_{i+1})$ (MFP is a solution of the flow equations). Note that, at this point, $\mathit{goto}(\beta_{i+1})$ depends only on MFP sets and on the control flow graph. From the induction hypothesis, we get $\mathit{loc} \in \mathrm{AP}_a^n(\beta_i)$. In particular, this means $\mathit{loc} \in \mathit{vars}(\beta_i) = \mathit{vars}(\beta_{i+1})$, *i.e.*, β_{i+1} is in the scope of a declaration of *loc*.

Subcase 1.1 β_{i+1} is *not* a labelled statement.

In this case, we have $\mathit{goto}(\beta_{i+1}) = \mathit{vars}(\beta_{i+1})$ from the definition of the *goto* set.

Subcase 1.1.1 β_{i+1} is *not* a `while` statement.

There exists in the *CFG* only one edge to $\mathcal{B}(\beta_{i+1})$, namely $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$. This means that

$$\mathit{path}_b^{n+1}(\beta_{i+1}) = \mathit{path}_a^n(\beta_i) \oplus \mathcal{B}(\beta_{i+1})$$

and implicitly $\mathit{loc} \in \mathrm{AP}_b^{n+1}(\beta_{i+1})$ since $\mathit{loc} \in \mathrm{AP}_a^n(\beta_i)$ (induction hypothesis). Remember that we derived earlier that β_{i+1} is in the scope of *loc*, *i.e.*, $\mathit{loc} \in \mathit{vars}(\beta_{i+1})$.

Subcase 1.1.2 β_{i+1} is a `while` statement.

There could be many edges to $\mathcal{B}(\beta_{i+1})$ in the *CFG* (if the `while` has associated `continue` statements). If there are no `continue` statements corresponding to our `while` statement, then the proof is as in **Subcase 1.1.1**.

If there are `continue` statements, we would like to show that *loc* is assigned on every path *p* to $\mathcal{B}(\beta_{i+1})$ of length at most $n + 1$. If *p* contains no `continue` statements, then *p* has the last edge $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$, *i.e.*, *p* passes through $\mathcal{A}(\beta_i)$. If *p* contains a `continue` statement associated to our `while` and eventually passes through `finally` blocks associated to `try` blocks that contain the `continue` statement, then *p* necessarily passes through $\mathcal{A}(\beta_i)$ because the *CFG* shows that it is not possible to "jump" into the `while` body (in which our `continue` is embedded). On the other hand, since $\mathit{loc} \in \mathrm{MFP}_a(\beta_i)$, we get $\mathit{loc} \in \mathrm{AP}_a^n(\beta_i)$ from the induction hypothesis, *i.e.*, *loc* is assigned on every path to $\mathcal{A}(\beta_i)$ of length at most *n*. Consequently, *p* assigns *loc* since *p* passes through $\mathcal{A}(\beta_i)$. We are now sure that *loc* is assigned on every path to $\mathcal{B}(\beta_{i+1})$ of length at most $n + 1$, and therefore we can conclude $\mathit{loc} \in \mathrm{AP}_b^{n+1}(\beta_{i+1})$.

Subcase 1.2 β_{i+1} is a labelled statement $L:$ *stm*.

As in the case of a `while` statement, there could be many edges to $\mathcal{B}(\beta_{i+1})$ in the *CFG* (if there are `goto` statements pointing to our labelled statement). If there are no associated `goto` statements, then the proof is the same as in the case of a `while` statement with no associated `continue` statements.

We want to show that *loc* is assigned on every path *p* to $\mathcal{B}(\beta_{i+1})$ of length at most $n + 1$. If *p* contains no `goto` *L* statements, then *p* necessarily passes through $\mathcal{A}(\beta_i)$. And since *loc* is assigned on every path to $\mathcal{A}(\beta_i)$ of length at most *n* (because of $\mathit{loc} \in \mathrm{AP}_a^n(\beta_i)$), we are sure that *p* assigns *loc*.

Suppose that *p* passes through a $^\gamma$`goto` *L* statement and eventually through `finally` blocks associated to `try` blocks in which γ is embedded. Considering the definition of the *goto* set, we can derive $\mathit{loc} \in \mathrm{MFP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$ since $\mathit{loc} \in \mathit{goto}(\beta_{i+1})$.

If there are no `finally` blocks in the list $\mathit{Fin}(\gamma, \beta_{i+1})$, then $\mathit{JoinFin}(\gamma, \beta_{i+1}) = \emptyset$ and implicitly $\mathit{loc} \in \mathrm{MFP}_b(\gamma)$. By the induction hypothesis, we obtain $\mathit{loc} \in \mathrm{AP}_b^n(\gamma)$. It means that *p*, which is of length at most $n + 1$ and contains $\mathcal{B}(\gamma)$, assigns *loc*.

Let us now suppose that the list $\mathit{Fin}(\gamma, \beta_{i+1})$ is non-empty: $\mathit{Fin} = [\gamma_1, \ldots, \gamma_k]$. By the definition of the set $\mathit{JoinFin}(\gamma, \beta_{i+1})$, we get

$$\mathit{loc} \in \mathrm{MFP}_b(\gamma) \cup \bigcup_{j=1}^{k} \mathrm{MFP}_a(\gamma_j)$$

If $\mathit{loc} \in \mathrm{MFP}_b(\gamma)$, then, by the induction hypothesis, we derive $\mathit{loc} \in \mathrm{AP}_b^n(\gamma)$, and we are sure that *p*, which passes through $\mathcal{B}(\gamma)$, assigns *loc*.

If there is a `finally` block γ_j such that $\mathit{loc} \in \mathrm{MFP}_a(\gamma_j)$, then the induction hypothesis implies $\mathit{loc} \in \mathrm{AP}_a^n(\gamma_j)$. And since *p* necessarily passes through $\mathcal{A}(\gamma_j)$, we are sure that *p* assigns *loc*.

Thus, we have analyzed every possible path to $\mathcal{B}(\beta_{i+1})$ of length at most $n + 1$ and we showed that each such a path assigns *loc*, *i.e.*, $\mathit{loc} \in \mathrm{AP}_b^{n+1}(\beta_{i+1})$.

Case 2 α is `try` ${}^{\beta}$*block* `finally` ${}^{\gamma}$*block*$'$.

Here we do the proof for $\mathrm{MFP}_b(\gamma) \subseteq \mathrm{AP}_b^{n+1}(\gamma)$. Let *loc* be a local variable in the set $\mathrm{MFP}_b(\gamma)$. Following the data flow equations, we get $\mathit{loc} \in \mathrm{MFP}_b(\alpha)$. The induction hypothesis implies then $\mathit{loc} \in \mathrm{AP}_b^n(\alpha)$ and, in particular, $\mathit{loc} \in \mathit{vars}(\alpha)$.

It is important to notice that in the *CFG* there could be many edges to $\mathcal{B}(\gamma)$: from the *entry point* of the `try`-`finally` statement $(\mathcal{B}(\alpha), \mathcal{B}(\gamma))$, from the *end point* of the `try` block $(\mathcal{A}(\beta), \mathcal{B}(\gamma))$, from a `goto`, `break` or `continue` statement $(\mathcal{B}(\delta), \mathcal{B}(\gamma))$ (within a *conditioned* path), and from the *end point* of another `finally` block $(\mathcal{A}(\omega), \mathcal{B}(\gamma))$ (within a *conditioned* path). We claim that independent of the last edge of a path *p* to $\mathcal{B}(\gamma)$, *p* passes through the *entry point* $\mathcal{B}(\alpha)$ of the `try`-`finally` statement.

- If the last edge of *p* is $(\mathcal{B}(\alpha), \mathcal{B}(\gamma))$, then there is nothing to prove.
- If the last edge is $(\mathcal{A}(\beta), \mathcal{B}(\gamma))$, then the claim holds since the *end point* $\mathcal{A}(\beta)$ can be reached only through $\mathcal{B}(\alpha)$ (according to the *CFG*, it is not possible to "jump" into the `try` block).
- If the last edge is $(\mathcal{B}(\delta), \mathcal{B}(\gamma))$, then the claim can be justified in the same way as above, because the respective jump statements are supposed to be embedded in the `try` block.
- If the last edge of *p* is a *conditioned* edge $(\mathcal{A}(\omega), \mathcal{B}(\gamma))$, then necessarily the `finally` block ω (as well as the jump statement which triggered the conditioning) is embedded in our `try` block. This means that in order to justify the claim, we can apply the same argument as above.

So, all the paths to $\mathcal{B}(\gamma)$ should pass through $\mathcal{B}(\alpha)$, and since $\mathit{loc} \in \mathrm{AP}_b^n(\alpha)$, we can be sure that $\mathit{loc} \in \mathit{vars}(\gamma) = \mathit{vars}(\alpha)$ is assigned on every path to $\mathcal{B}(\gamma)$ of length at most $n + 1$, *i.e.*, $\mathit{loc} \in \mathrm{AP}_b^{n+1}(\gamma)$, and the proof of the considered relation is complete. □

As explained above, we can actually prove more: The *MFP* solution is not only an approximation of *AP*, but it is a perfect solution (Theorem 3.4.3). For this, we also use the following theorem that states that the *MFP* solution contains the local variables which are initialized over *all possible paths*.

Theorem 3.4.2 *If α is an expression or a statement, then the following relations are true:* $\text{AP}_b(\alpha) \subseteq \text{MFP}_b(\alpha)$ *and* $\text{AP}_a(\alpha) \subseteq \text{MFP}_a(\alpha)$*. Moreover, if α is a boolean expression, then* $\text{AP}_t(\alpha) \subseteq \text{MFP}_t(\alpha)$ *and* $\text{AP}_f(\alpha) \subseteq \text{MFP}_f(\alpha)$*.*

Proof. Tarski's fixed point theorem [119] states that *MFP* is the lowest upper bound (with respect to $\ll$) of the set $\mathit{Ext}(F) = \{X \in D \mid X \ll F(X)\}$. It then suffices to show that the r-tuple consisting of the *AP* sets is an element of $\mathit{Ext}(F)$ since *MFP* is in particular an upper bound of this set. Since $\ll$ is the pointwise subset relation, the idea is to prove the left-to-right subset relations for the data flow equations in Tables 3.10, 3.11, and 3.12, where instead of the sets *before*, *after*, *true* and *false*, we have the sets AP_b, AP_a, AP_t, and AP_f, respectively.

We only consider here one critical case, namely the case of a block of statements: Assuming that β_{i+1} is a labelled statement L : *stm* in a block α given by $\{{}^{\beta_1}\mathit{stm}_1 \ldots {}^{\beta_n}\mathit{stm}_n\}$, we want to prove $\text{AP}_b(\beta_{i+1}) \subseteq \text{AP}_a(\beta_i) \cap \mathit{goto}(\beta_{i+1})$. Note that here, $\mathit{goto}(\beta_{i+1})$ depends only on the AP sets and on the control flow graph *CFG*.

Let *loc* be a variable in $\text{AP}_b(\beta_{i+1})$, *i.e.*, *loc* is assigned on every path to $\mathcal{B}(\beta_{i+1})$. An immediate consequence is $\mathit{loc} \in \text{AP}_a(\beta_i)$, since all the paths to $\mathcal{A}(\beta_i)$ are — "modulo" the edge $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$ — also paths to $\mathcal{B}(\beta_{i+1})$, and no variable is assigned on this edge.

To show $\mathit{loc} \in \mathit{goto}(\beta_{i+1})$, we need to prove that the variable *loc* is in the set $\text{AP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$ for every ${}^{\gamma}$`goto` L whose target is our labelled statement.

If there is no such `goto` statement, then we obviously have $\mathit{loc} \in \mathit{goto}(\beta_{i+1})$ since in this case $\mathit{goto}(\beta_{i+1}) = \mathit{vars}(\beta_{i+1})$ and $\mathit{loc} \in \text{AP}_b(\beta_{i+1}) \subseteq \mathit{vars}(\beta_{i+1})$. Let us now suppose that there exists at least one `goto` statement γ pointing to β_{i+1}.

Case 1 $\mathcal{B}(\gamma)$ is not reachable.

Then $\mathit{path}_b(\gamma)$ is empty, and consequently, we get $\mathit{loc} \in \text{AP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$ because $\text{AP}_b(\gamma) = \mathit{vars}(\gamma) \supseteq \mathit{vars}(\beta_{i+1})$ and $\mathit{loc} \in \mathit{vars}(\beta_{i+1})$. The last subset relation holds because, in C$^\sharp$, a `goto` statement should be always in the scope of the corresponding labelled statement.

Case 2 $\mathcal{B}(\gamma)$ is reachable.

Let p be an arbitrary path to $\mathcal{B}(\gamma)$. Here we will only consider the case when there are `finally` blocks from γ to β_{i+1}, *i.e.*, $\mathit{Fin}(\gamma, \beta_{i+1}) = [\gamma_1, \ldots, \gamma_k]$ (the proof of the case where there are no `finally` blocks is much simpler). Accordingly, also the edges

$$(\mathcal{B}(\gamma), \mathcal{B}(\gamma_1)), (\mathcal{A}(\gamma_k), \mathcal{B}(\beta_{i+1})), (\mathcal{A}(\gamma_j), \mathcal{B}(\gamma_{j+1})), j = 1, k-1$$

defined by the set $\mathit{ThroughFin}_b(\gamma, \beta_{i+1})$ are added to the *CFG*.

We prove $\mathit{loc} \in \text{AP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$ by *reductio ad absurdum*. Let us assume that $\mathit{loc} \notin \text{AP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$. This is equivalent to $\mathit{loc} \notin \text{AP}_b(\gamma)$ and $\mathit{loc} \notin \text{AP}_a(\gamma_j)$ for all $j = 1, k$. This means that the paths $p_0 \in \mathit{path}_b(\gamma)$ and $p_j \in \mathit{path}_a(\gamma_j)$ for $j = 1, k$ exist, such that *loc* is not assigned on any of these paths. A simple inspection of the *CFG* shows that the point $\mathcal{B}(\gamma_j)$ necessarily occurs on the path p_j for every $j = 1, k$ since it is not possible to "jump" into a `finally` block. We now want to prove that the following list

$$q := p_0 \oplus p_1[\mathcal{B}(\gamma_1), \mathcal{A}(\gamma_1)] \oplus \ldots \oplus p_k[\mathcal{B}(\gamma_k), \mathcal{A}(\gamma_k)] \oplus \mathcal{B}(\beta_{i+1})$$

represents a valid path to $\mathcal{B}(\beta_{i+1})$. The only problem that could arise is concerning the *conditioned* edges. Remember that the edges *conditioned* by a certain `goto`, `break` or `continue` statement can only be used in paths (or subpaths) that contain the *entry point* of the respective jump statement. The use of edges $(\mathcal{B}(\gamma), \mathcal{B}(\gamma_1)), \ldots (\mathcal{A}(\gamma_k), \mathcal{B}(\beta_{i+1}))$ is correct as long as our path q contains $\mathcal{B}(\gamma)$.

Let us consider one of the subpaths $p_j[\mathcal{B}(\gamma_j), \mathcal{A}(\gamma_j)]$ used in q. If this subpath contains a *conditioned edge*, then since the *conditioned* edges connect jump statements with `finally` blocks we would be sure that these `finally` blocks are embedded in our `finally` block γ_j. The respective jump statement is embedded into the `try` blocks (associated to the *conditioned* "connected" `finally` blocks) which necessarily should be in γ_j (this is an immediate consequence of the C$^\sharp$ grammar). So, the jump statement is necessarily embedded in the `finally` block γ_j. Considering that the subpath $p_j[\mathcal{B}(\gamma_j), \mathcal{A}(\gamma_j)]$ contains *conditioned* edges, we get that also p_j uses the same *conditioned* edges and since we assumed that p_j is a valid path, necessarily p_j should contain the *entry point* of the respective jump statement which, as we proved above, is embedded in our `finally` block, and consequently appears in the subpath $p_j[\mathcal{B}(\gamma_j), \mathcal{A}(\gamma_j)]$. It means that this subpath is valid. Obviously, this is true for all the considered subpaths in q.

The above defined path q is a valid path to $\mathcal{B}(\beta_{i+1})$ which does not assign *loc*. Obviously, this contradicts *loc* $\in \text{AP}_b(\beta_{i+1})$ and therefore our assumption is wrong. Hence, we obtain the desired *loc* $\in \text{AP}_b(\gamma) \cup \textit{JoinFin}(\gamma, \beta_{i+1})$. $\square$

The following result is then an obvious consequence of Theorems 3.4.1 and 3.4.2:

Theorem 3.4.3 *The maximal fixed point solution of the data flow equations in Tables 3.10, 3.11, and 3.12 represents the sets of local variables which are assigned over all possible execution paths. More exactly, for every expression or statement α, the following are true:* $\text{AP}_b(\alpha) = \text{MFP}_b(\alpha)$ *and* $\text{AP}_a(\alpha) = \text{MFP}_a(\alpha)$. *Moreover, if α is a boolean expression:* $\text{AP}_t(\alpha) = \text{MFP}_t(\alpha)$ *and* $\text{AP}_f(\alpha) = \text{MFP}_f(\alpha)$.

Struct type variables Suppose now that we include for the analysis also variables of struct types. In this case, the C$^\sharp$ compiler relies on the expanded MFP sets and the correctness of the analysis would mean that the expanded MFP sets are a *safe approximation* of the expanded AP sets. On the other hand, we proved in Theorem 3.4.3, that the MFP and AP sets coincide. Then, also the expanded MFP sets will coincide with the expanded AP sets; so they are, in particular, a *safe approximation*. This means that, allowing variables of struct types does not affect the correctness of the definite assignment analysis. One can justify the correctness also by applying Theorem 3.4.1 and by observing that the "expansion" function is monotonic.

3.5 The Semantics of C$^\sharp{}_S$ Programs

In this section, the static and dynamic semantics of C$^\sharp{}_S$ are formally specified, along the lines of [24].

Function definition	Function name
$opResVal : Map(Bop \times Val \times Val, Val)$	operation result value
$convertVal : Map(PrimitiveType \times SimpleVal, SimpleVal)$	conversion result value

Table 3.16: The basic static functions.

3.5.1 The Static Semantics

Besides the execution environment and the grammar defined in Section 3.2, the static semantics of $C^\sharp{}_S$ also comprises the *static functions* in Table 3.16.

For any two values val and val' and binary operator bop, $opResVal(bop, val, val')$ returns the result of applying bop to operands val and val' (assuming that the operation is valid). For any primitive type T and simple value val, $convertVal(T, val)$ yields the result of converting val to T.

Given a type T, $defVal(T) : Map(Type, Val)$ gives the *default value*, *i.e.*, the ”zero”, of type T. It is worth mentioning that the default value of a struct is the same as the value computed by the struct's default constructor, namely the tuple with all the components initialized to default values.

Definition 3.5.1 (Default value) *For a type T, we define*

$$defVal(T) = \begin{cases} False, & \text{if } T = \texttt{bool}; \\ 0, & \text{if } T = \texttt{int}; \\ 0l, & \text{if } T = \texttt{long}; \\ 0d, & \text{if } T = \texttt{double}; \\ \texttt{null}, & \text{if } T \in RefType; \\ \{T{::}F \mapsto defVal(fieldType(T{::}F)) \mid T{::}F \in instFields(T)\}, & \text{if } T \in Struct. \end{cases}$$

Remark 3.5.1 *The definition of defVal is indeed recursive, but it is* well-founded *in any case, in the sense that there is no type T such that defVal(T) is defined in terms of defVal(T) itself. This is justified in particular by the key fact that a struct cannot be* recursive*, i.e., the declaration of a struct S is* not *allowed to use the name S in the types of its instance fields.*

A virtual method call is resolved dynamically, *i.e.*, at run-time. More exactly, the method to be *invoked* at run-time is determined based on the type of the object the method is *called* on. This is known as *dynamic method binding* [74, §10.5.3], and Definition 3.5.2 specifies it as a derived function $lookUp : Map(Type \times MRef, MRef)$. Thus, given the run-time type T (see the definition in Section 3.5.2) of an object and a virtual method $T'{::}M$, $lookUp(T, T'{::}M)$ yields the method $T''{::}M$, where $T''{::}M$ is the *first* implementation of $T'{::}M$ declared by a supertype of T, starting with T itself.

Definition 3.5.2 (Method lookup) *For a type T and a method T′::M, we define:*

```
$lookUp(T, T'::M) =$
    **if** $T$ declares a non-abstract method $M \wedge$
        $((T = T') \vee (T' \in Class \Rightarrow T'::M$ overrides $T::M)) \wedge$
        $(T' \in Interface \Rightarrow T::M$ implements $T'::M)$
    **then** $T::M$
    **elseif** $T =$ object **then** $undef$
    **else** $lookUp(T'', T'::M)$ where $T''$ is the direct base class of $T$
```

3.5.2 The Dynamic Semantics

The dynamic semantics does not work directly on $C^\sharp{}_S$ syntax, but on the abstract terms produced by the grammar in Figure 3.1. The dynamic *state* of $C^\sharp{}_S$ programs is given by the *dynamic functions* in Table 3.18. Some *operational semantics rules* (that enable the transition between states) are explained in Section 3.5.2.1.

The dynamic semantics specification assumes that the static semantics is correct. Its definition proceeds by walking through the AST of the current method *meth* and determining at each node the effect of the $C^\sharp{}_S$ construct corresponding to the node. This walk is formalized by means of a cursor ►, for example, as in $^{\blacktriangleright}exp$ or $^{\blacktriangleright}stm$, whose position in the tree – represented by an *abstract program counter pos* – is updated using the static tree functions defined in Table 3.6. The counter *pos* points to the current expression or statement to be executed. The moves of *pos* implicitly contain the $C^\sharp{}_S$'s control flow graph.

The addresses of the local variables and parameters are maintained in the function *locAdr*. Thus, for a local variable or parameter *loc*, $locAdr(loc)$ is the address *allocated* for *loc* in the current method (if *loc* is a local variable or a value parameter) or the address *passed* to the current method for *loc* (if *loc* is a `ref`/`out` parameter). The function *values* is used to store intermediate evaluation results, elements of the universe *Result* defined in Table 3.17. Thus, given the node at a position α, $values(\alpha)$ is eventually assigned a value, an address, a reason for abruption (element of the universe *Abr* defined in Table 3.17), *undef*, or the constant `Norm` (used to denote the *normal completion* of statements). Unlike [24], we do not consider the addresses to be values. For the clarity of the type safety proof, the universes *Adr* and *Val* are regarded as disjoint in our semantics model.

A *reason for abruption*, as opposed to normal completion, represents an abruption of the control flow, which can happen due to one of the statements `break`, `continue`, `goto`, `return`, and `throw`.

Method invocations produce (activation) *frames*, elements of the universe *Frame* defined in Table 3.17. The *current frame* is given by the 4-tuple $(meth, pos, locAdr, values)$. The currently still to be executed frames on the stack are maintained by the function *frameStack*.

For accessing/updating the values of (managed) memory locations, *e.g.*, local variables, parameters, fields, the function *mem* is considered. This function assigns a simple value or object reference to every memory address[8].

The function *fieldAdr* assigns to every instance field of an object reference or of a struct

[8]In our abstract memory model, we assume that a single address can hold either a simple value or an object reference, whereas a struct value is held by a block of addresses.

Universe			Typical use
Adr			adr, adr', adr''
Result	=	$Val \cup Adr \cup Abr \cup \{undef, \texttt{Norm}\}$	*res*
Abr	=	$\{\texttt{Break}, \texttt{Continue}, \texttt{Goto}(Lab), \texttt{Return}, \texttt{Return}(Val), \texttt{Exc}(ObjRef)\}$	*abr*
Frame	=	$MRef \times Pos \times Map(Loc, Adr) \times Map(Pos, Result)$	$fr, fr', fr_1, \ldots, fr_n$

Table 3.17: Semantics specific universes.

instance stored at a given address its allocated address. More exactly, the address of an instance field $C::F$ of an object pointed to by a reference *ref* is given by $fieldAdr(ref, C::F)$. Also, the address of an instance field $S::F$ of a struct instance stored at the address *adr* is $fieldAdr(adr, S::F)$. For a reference *ref* pointing to a boxed value, $addressOf(ref)$ returns the address of the value being boxed.

Every object, *i.e.*, class instance or boxed value type, has an initial part that represents its type, usually called the *run-time type*, by opposition to the *compile-time type* $\mathcal{ST}$. For every object reference *ref*, $actualTypeOf(ref)$ returns the run-time type of *ref*. The run-time type of an instance of a class C is set to C upon the instance creation, whereas the run-time type of a boxed value of type T is set to T upon the boxing. Note that, although the boxing returns objects, the dynamic type checks, *e.g.*, triggered by the `is`/`as` operators, can directly reference the value type T.

A `throw` statement without expression, allowed in a `catch` block only, is applied to rethrow the exception handled in the corresponding `catch` block. To specify the semantics of these statements, a stack *excStack* of exception references is required to record exceptions which are to be re-thrown.

The invocation list of a delegate is maintained in *invocationList*. So, for a delegate reference d, $invocationList(d)$ keeps the list of pairs consisting of a target object and a target method pointed to by the delegate d.

Initial State The following conditions should be satisfied in the initial state of the operational semantics model by the dynamic functions in Table 3.18:

meth	=	`Main`	*fieldAdr*	=	$\emptyset$
pos	=	α, where $\alpha \in Pos$ with ${}^{\alpha}body(\texttt{Main})$	*addressOf*	=	$\emptyset$
locAdr	=	$\emptyset$	*actualTypeOf*	=	$\emptyset$
values	=	$\emptyset$	*excStack*	=	[ ]
frameStack	=	[ ]	*invocationList*	=	$\emptyset$
$mem(adr)$	=	*undef*, for every $adr \in Adr$			

As *mem* is applied to ”read” from the memory *only* simple values and object references, it is considered a (derived) function $memVal : Map(Adr \times Type, Val \cup \{undef\})$, which in addition ”reads” struct values. The definition below of *memVal* relies on the fact that the struct values

Function definition			Function name
meth	:	*MRef*	current method
pos	:	*Pos*	current position
locAdr	:	$Map(Loc, Adr)$	local variable address
values	:	$Map(Pos, Result)$	evaluation results
frameStack	:	$List(Frame)$	stack of call frames
mem	:	$Map(Adr, SimpleVal \cup ObjRef \cup \{undef\})$	memory function
fieldAdr	:	$Map((ObjRef \cup Adr) \times Type{::}Field, Adr)$	instance field addresses
addressOf	:	$Map(ObjRef, Adr)$	boxed value address
actualTypeOf	:	$Map(ObjRef, Class \cup ValueType)$	reference run-time type
excStack	:	$List(ObjRef)$	exception stack
invocationList	:	$Map(ObjRef, List(ObjRef \times MRef))$	delegate invocation list

Table 3.18: The basic dynamic functions.

are defined as mappings. Thus, for an address *adr* and a type *T*, $memVal(adr, T)$ is the value stored in the memory block at *adr*, *sufficient to hold values of type T*.

Definition 3.5.3 *For an address adr and a type T, we define:*

```
$memVal(adr, T) =$
  **if** $T \in PrimitiveType \cup RefType$ **then** $mem(adr)$
  **elseif** $T \in Struct$ **then**
    $\{T{::}F \mapsto memVal(fieldAdr(adr, T{::}F), fieldType(T{::}F)) \mid T{::}F \in instFields(T)\}$
```

The "write" in memory of values, including struct values, is accomplished with the below defined macro WRITEMEM. Thus, given an address *adr*, a type *T* and a value *val*, $\text{WRITEMEM}(adr, T, val)$ stores *val* in the memory block at *adr*, sufficient to hold values of type *T*.

Definition 3.5.4 *Given an address adr, a type T and a value val, we define:*

```
$\text{WRITEMEM}(adr, T, val) \equiv$
  **if** $T \in PrimitiveType \cup RefType$ **then** $mem(adr) := val$
  **elseif** $T \in Struct$ **then**
    **forall** $T{::}F \in instFields(T)$ **do**
      $\text{WRITEMEM}(fieldAdr(adr, T{::}F), fieldType(T{::}F), val(T{::}F))$
```

Note that the definitions of *memVal* and WRITEMEM are *well-founded* since a struct cannot be *recursive*, *i.e.*, the declaration of a struct type is *not* allowed to use the struct name in the types of its instance fields.

Example 3.5.1 *In order to aid the reader's understanding of the definitions of memVal and* WRITEMEM*, we consider the following example. Let S and S' be two structs. We assume that S* *declares two instance fields $S::F$ and $S::F'$ of declared types S' and* `object`*, respectively. Moreover, S' has two instance fields $S'::F''$ and $S'::F'''$ of declared types* `object` *and* `int`*, respectively.*

When a value val of type S is created (upon evaluating a `new` *expression), a memory block at an address adr, sufficient to hold values of type S, is allocated for val. As val is a composite value, addresses adr′ and adr″ are also allocated for the values which compose val, i.e., for val's instance fields $S::F$ and $S::F'$ (see the evaluation rule of the* `new` *expressions in Section 3.5.2.1):*

$$fieldAdr(adr, S::F) = adr' \qquad fieldAdr(adr, S::F') = adr''$$

Similarly, as the field $S::F$ is of struct type S', addresses adr''' and $adr^{(4)}$ are allocated for the instance fields $S'::F''$ and $S'::F'''$ of the struct instance stored at adr':

$$fieldAdr(adr', S'::F'') = adr''' \qquad fieldAdr(adr', S'::F''') = adr^{(4)}$$

The value val can then be recursively computed as the value of type S stored at the address adr:

$$\begin{aligned}
&memVal(adr, S)\\
&\quad = \{S::F \mapsto memVal(fieldAdr(adr, S::F), S'),\\
&\qquad S::F' \mapsto memVal(fieldAdr(adr, S::F'), \texttt{object})\}\\
&\quad = \{S::F \mapsto memVal(adr', S'), S::F' \mapsto memVal(adr'', \texttt{object})\}\\
&\quad = \{S::F \mapsto \{S'::F'' \mapsto memVal(fieldAdr(adr', S'::F''), \texttt{object}),\\
&\qquad\qquad S'::F''' \mapsto memVal(fieldAdr(adr', S'::F'''), \texttt{int})\},\\
&\qquad S::F' \mapsto memVal(adr'', \texttt{object})\}\\
&\quad = \{S::F \mapsto \{S'::F'' \mapsto memVal(adr''', \texttt{object}), S'::F''' \mapsto memVal(adr^{(4)}, \texttt{int})\},\\
&\qquad S::F' \mapsto memVal(adr'', \texttt{object})\}\\
&\quad = \{S::F \mapsto \{S'::F'' \mapsto mem(adr'''), S'::F''' \mapsto mem(adr^{(4)})\}, S::F' \mapsto mem(adr'')\}
\end{aligned}$$

If one has, for example, to update val's instance field $S::F$ with the value

$$val' = \{S'::F'' \mapsto val'', S'::F''' \mapsto val'''\}$$

then val' is stored at adr'. This is accomplished through WRITEMEM(adr', S', val'). *This macro simply updates the instance fields $S'::F''$ and $S'::F'''$ with val'' and val''', respectively. Concerning type safety, the following question arises: Is the updated value val still of type S?*

Not surprisingly, if a value *val* is stored in a memory block, sufficient to hold values of a type T, then the value "read" from that block is *val*.

Lemma 3.5.1 *After* WRITEMEM(adr, T, val) *is executed,* $memVal(adr, T) = val$.

Proof. By Definitions 3.5.3 and 3.5.4. □

Variable expressions vs. Address positions One key aspect in the modelling is represented by the "indirection through memory addresses". The idea of separately defining the universe *Vexp* of variable expressions is to isolate the expressions whose evaluation *can* produce addresses. The addresses are needed for the call-by-reference mechanism with `ref` and `out` parameters. Thus, as we will see in Section 3.5.2.1 and in the Appendix, if a variable expression is passed "by reference" in a method call, it is the address of the variable expression which is passed as argument to the method. This means that, depending on the context, the evaluation of a variable expression *vexp* is required to yield its address or its value. In other words, if *vexp* is at an *address position*, then the address of *vexp* is needed, otherwise the value of *vexp*. An address position (relative to a given attributed syntax tree) is the position in a `ref`/`out` argument, or the position which is assigned to in a compound assignment, or the position of an access (other than in a delegate creation) to a variable expression of a struct type. Definition 3.5.5 makes precise what we mean by an address position.

Definition 3.5.5 *For a position $\alpha \in Pos$, we define:*

$$\begin{array}{l} isAddressPos(\alpha) :\Leftrightarrow \\ \quad first(up(\alpha)) = \alpha \wedge \\ \quad (label(up(\alpha)) \in \{\texttt{ref}, \texttt{out}\} \cup Aop \vee \\ \quad (label(up(\alpha)) = \text{'.'} \wedge up(\alpha) \notin Dexp \wedge \alpha \in Vexp \wedge \mathcal{ST}(\alpha) \in Struct)) \end{array}$$

Remark 3.5.2 *The definition of isAddressPos in* [24] *considers an address position the position of the target object expression in a delegate creation expression, i.e., when $up(pos) \in Dexp$, though the target object cannot be an address – see also the description of delegate creations in Section 3.5.2.1.*

The following lemma says that only variable expressions can occur at address positions.

Lemma 3.5.2 *If $isAddressPos(\alpha)$, then $\alpha \in Vexp$.*

Proof. By Definition 3.5.5 and $C^{\sharp}{}_{S}$'s grammar in Figure 3.1. □

3.5.2.1 The Evaluation of the $C^{\sharp}{}_{S}$ Expressions and Statements

In this section, we briefly describe the evaluation rules for a few expressions, whose formal specifications are defined in [24]. For the reader's convenience, we include these rules in the Appendix. The evaluation rules extensively make use of the macros defined and informally described in Table 3.19. The definitions of these macros assume an infinite memory space *Adr*, assumption that is often made when specifying the semantics of programming languages.

Compound assignment In *vexp bop* $=$ *exp*, *vexp* is evaluated before *exp*. Since *vexp* occurs at an address position (see Definition 3.5.5), its evaluation should produce an address, say *adr*. Let *val* be the value *exp* evaluates to. After *vexp* and *exp* are evaluated, the assignment is in the form *adr bop* $=$ *val*, and *pos* points to the position of *val*. The result val'' of *bop* having as operands the value stored at *adr* and *val* is stored with WRITEMEM at *adr* and yielded at the parent position of *pos*. To yield a result at a given position, the macro YIELD is used:

let $ref = new(ObjRef, C)$ **in** $P \equiv$
  import ref **do**
    $ObjRef(ref) := True$
    ALLOCFIELDS$(ref, instFields(C))$
  seq P
	Returns a reference to a newly allocated object of a given type.

let $ref = new(ObjRef)$ **in** $P \equiv$
  import ref **do**
    $ObjRef(ref) := True$
  seq P
	Returns a new object reference.

let $adr = new(Adr, T)$ **in** $P \equiv$
  import adr **do**
    ALLOCADR(adr, T)
  seq P
	Returns the address of a newly allocated memory block, *sufficient to hold values a given type*.

ALLOCFIELDS$(x, A) \equiv$
  forall $T{::}F \in A$ **do**
    import adr **do**
      $fieldAdr(x, T{::}F) := adr$
      ALLOCADR$(adr, fieldType(T{::}F))$
	Allocates the instance fields in a given set.

ALLOCADR$(adr, T) \equiv$
  $Adr(adr) := True$
  if $T \in Struct$ **then**
    ALLOCFIELDS$(adr, instFields(T))$
	Allocates the instance fields of a value of a given type, stored at a given address.

Table 3.19: Allocating object references and addresses.

```
YIELD$(res, \alpha) \equiv$
  $values(\alpha) := res$
  $pos := \alpha$
```

This macro is applied in two forms:

YIELD$(res) \equiv$ YIELD(res, pos) YIELDUP$(res) \equiv$ YIELD$(res, up(pos))$

If the binary operation is a division by zero, then a `DivideByZeroException` is raised. This case is omitted in [24]'s semantics model.

Boxing and unboxing The boxing occurs, for example, when evaluating a type cast $(T)\ exp$, where T is a reference type and the compile-time type $\mathcal{ST}$ of exp is a value type. Note that the type constraints in Table 3.7 guarantee that the compile-time type of exp is a subtype of T.

For boxing, the macro *NewBox* is defined. Thus, for a type T, a value type *val*, $NewBox(T, val)$ returns a reference to a newly allocated object which embeds *val*:

```
**let** $ref = NewBox(T, val)$ **in** $P \equiv$
  **let** $ref = new(ObjRef)$ **and** $adr = new(Adr, T)$ **in**
    $actualTypeOf(ref) := T$
    $addressOf(ref) \quad := adr$
    $\textsc{WriteMem}(adr, T, val)$
  **seq** $P$
```

The unboxing is triggered by the evaluation of an expression of the form (T) *exp*, where T is a value type and the compile-time type of *exp* is a reference type. The value embedded into the boxed object is yielded up if its type is T. Otherwise, an `InvalidCastException` is raised.

Field access Assume that in $exp.T{::}F$, *exp* is evaluated to *valadr*[9], possibly an address due to the position of *exp*. If *valadr* is a struct value, as opposed to an address where a struct value is stored, then its field $T{::}F$ is computed as $valadr(T{::}F)$ (remember that the struct values are defined as mappings). If *valadr* is `null`, a `NullReferenceException` is thrown[10]. If *valadr* is not a struct value or `null`, then *valadr* is either an address or an object reference. In either case, the address of $T{::}F$ is $fieldAdr(valadr, T{::}F)$. Depending on whether $up(pos)$ is an address position, the address of $T{::}F$ or the value stored at this address is yielded up, respectively. For this purpose, the macro YIELDINDIRECT is defined:

```
$\textsc{YieldIndirect}(adr, \alpha) \equiv$
  **if** $isAddressPos(\alpha)$ **then** $\textsc{Yield}(adr, \alpha)$
  **else** $\textsc{Yield}(memVal(adr, \mathcal{ST}(\alpha)), \alpha)$
```

Similarly to YIELD, we also have two forms for YIELDINDIRECT: $\textsc{YieldIndirect}(adr)$ and $\textsc{YieldUpIndirect}(adr)$.

Method call In the expression $exp.T{::}M(args)$, the expression *exp* is evaluated before the list of arguments *args*. Let *valadr* and *valadrs* be the value/address and the list of values/addresses *exp* and *args* evaluate to, respectively. If *valadr* is a `null` reference, then a `NullReferenceException` is raised. Otherwise, the method $T{::}M$ is called with the arguments *valadr* (as the `this` pointer) and *valadrs*. To call methods, we define the macro VIRTCALL. Thus, for a method $T{::}M$ called with the arguments *valadr* (as the `this` pointer) and *valadrs*, $\textsc{VirtCall}(T{::}M, valadr, valadrs)$ is defined as follows[11]:

[9]We typically use the name *valadr* for an evaluation result from $Val \cup Adr$, which, depending on the context, is a value or an address.

[10]As observed in [55], the semantics (related to the timing of the `null` check) of field accesses involved in assignments is violated by an optimization of the C$\sharp$ compiler of [3].

[11]The macro INVOKEMETHOD, defined in the Appendix, is used to invoke a method with a given list of arguments.

```
$\textsc{VirtCall}(T{::}M, valadr, valadrs) \equiv$
  if $callKind(T{::}M) = \texttt{Virtual}$ then
    let $T''{::}M = lookUp(actualTypeOf(valadr), T{::}M)$ in
      let this $=$ if $T'' \in Struct$ then $addressOf(valadr)$ else $valadr$ in
        $\textsc{InvokeMethod}(T''{::}M, [\texttt{this}] \cdot valadrs)$
  else $\textsc{InvokeMethod}(T{::}M, [valadr] \cdot valadrs)$
```

In $\textsc{VirtCall}(T{::}M, valadr, valadrs)$, if $T{::}M$ is virtual, the method to be invoked at run-time is determined with *lookUp*. The type of *valadr* considered by *lookUp* is $actualTypeOf(valadr)$. Depending on whether the method to be invoked at run-time is declared by a class or by a struct, the `this` pointer of the method to be called is *valadr* or the address $addressOf(valadr)$ of the value inside the boxed object (remember that the `this` pointer of a struct method call is passed ”by reference”), respectively.

Delegate creation We only describe here creations of the form `new` $D(exp.T{::}M)$. Let *val* be the value to which *exp* evaluates. If *val* is `null`, then a `NullReferenceException` is thrown. Otherwise, it is proceeded according to the following case distinction.

If $T{::}M$ is virtual, the target method to be inserted in $invocationList(d)$, where d is the newly allocated delegate, is determined via *lookUp*, exactly in the same way as the method to be invoked in the case of a method call. The target object is set to *val*.

Remark 3.5.3 [24]*'s semantics model does not perform a method lookup at the delegate creation time, but at the time when the delegate is called. This is, however, not according to* [74]*. Therefore, we changed the model to correctly reflect the Microsoft implementation.*

If $T{::}M$ is not virtual, $T{::}M$ is inserted in $invocationList(d)$. If T is a struct, then *val* (assumed to be a struct value) is boxed and considered as the target object. Otherwise, *val* becomes the target object.

Remark 3.5.4 *The case when the target object is a boxed struct value is omitted by* [24]*'s semantics model. Note that, this case is also possible if T::M is virtual.*

3.6 Type Safety

In this section, we discuss our type safety theorem together with its crucial lemmas. The properties ensured for legal, well-typed methods accepted by the definite assignment analysis are the following:

Type safety If an expression evaluates to a value, then the value is compatible with the compile-time type of the expression's position. The value parameters, local variables, instance fields always contain values of the declared types. The values inside boxed objects are compatible with the corresponding value types.

Lvalue evaluation consistency If an expression evaluates to an address, then the address is a valid address for the compile-time type of the expression's position. The `ref`/`out` parameters always point to valid addresses for the declared types. The addresses always occur at address positions, and the expressions in address positions always evaluate to addresses.

Statement execution consistency Run-time abruptions always occur in accordance with the corresponding compile-time constraints for jumps, method returns, or exceptions.

Any address appearing in the evaluation tree should be a *valid* (managed) *address*, *i.e.*, the address of a managed memory block as opposite to the address of a random memory block. In other words, the addresses occurring in the evaluation tree should only be memory locations the program is authorized to access. To ensure that, we have to answer the question: "*If an address adr appears in the evaluation tree at a position* α*, where it originates from*?". Intuitively, in general, there should be a variable expression *vexp* at α such that *adr* is "the address" of *vexp*, where "the address" of *vexp* might have been previously allocated for a variable (not necessarily *vexp*) to which *vexp* points (this is possible because of the call-by-reference mechanism). This natural answer is, however, not simple to formally reason about, since several evaluation steps might be required to produce the address of a variable expression. Moreover, it is *not* always the case that, if an address appears in the tree, then there exists an expression at the same position whose evaluation produced the address. A counterexample for this is given by the `ref`/`out` arguments: The argument `ref` *loc* at a position α is evaluated in two steps to the address of *loc*.

Due to the call-by-reference mechanism, it is possible that updating a variable *vexp* in a frame *fr* implies another variable *vexp*′ in another frame *fr*′ (below *fr* in the frame stack) to be updated. One has to carefully analyze these updates since *vexp*′ does not necessarily point directly to *vexp*, but it could also point to its "interior", *e.g.*, *vexp*′ is a field of a field of *vexp*. In both cases, one has to ensure that the update of *vexp*′ preserves the typing of *vexp* with respect to *fr*. Otherwise, C$\sharp$'s type safety would be violated when *fr* becomes (again) the current frame.

If a local variable *loc* is not defined in the local environment, then an access to *loc* yields the constant *undef*. This constant is then propagated in the evaluation tree, and eventually the operational semantics model stops, and the execution fails. Our type safety result ensures that the constant *undef* never occurs in the evaluation tree. This is because in the real C$\sharp$ implementation, there is no constant *undef*. In the real implementation, if the uninitialized *loc* is accessed, the execution proceeds with whatever value was in the memory at the address of *loc*. This value could be of any type, and thus C$\sharp$'s type safety would be violated. The fact that *undef* does not show up in the evaluation tree implies, in particular, that there no dangling pointers. To ensure that *undef* does not appear during the evaluation is not trivial. *Not* every address occurring in the evaluation tree holds a value. For example, consider an argument ${}^{\alpha}($`out` ${}^{\beta}$*loc*$)$, where the local variable is not definitely assigned. The address of the uninitialized *loc* is produced at β and then propagated to α. It is then crucial that the memory block pointed to by such an address is not "read". Note that, a memory block might be regarded as uninitialized, *i.e.*, the value "read" from the block is *undef*, though several memory blocks "in" the given block are initialized. Consider, for example, a local variable *loc* of type a struct *S*, where *S* has at least two instance fields. Let *S*::*F* be one of them. If *loc* is not initialized, then the value "read" from the memory block pointed to by *locAdr*(loc), sufficient to hold values of type *S*, is *undef*. However, if there was an assignment to $loc.S{::}F$, then the memory block pointed to by the address of $S{::}F$ holds a value though *loc* is regarded as uninitialized.

When defining the typing of values, one key point is how to distinguish between values of value types and boxed values of value types. Note that, when a value *val* of a value type T is boxed, we considered the type T as the run-time type used in dynamic type-checks. For this reason, we introduce *boxed types*, as reference types used *only* for the purpose of *verifying* type safety:

$$\boxed{BoxedType = \{boxed(T) \mid T \in ValueType\}}$$

Thus, we consider a boxed value of a value type T to be typable with the type $boxed(T)$.

To define the typing of values, we introduce the notion of run-time compatibility for C$^\sharp{}_S$'s types, boxed types, and *Null* (the type of the `null` references).

Definition 3.6.1 (Run-time compatibility) *The* run-time compatibility relation $\sqsubseteq$ *is the least reflexive and transitive relation such that:*

- *if* $T'' \in ValueType$ *extends / implements* T' *and* $boxed(T'') = T$*, or*
- $T, T' \in RefType$ *and* $T \preceq T'$*, or*
- $T = Null$ *and* $T' \in RefType$*,*

then $T \sqsubseteq T'$.

The following lemma is applied in the type safety proof:

Lemma 3.6.1 *If* $T \sqsubseteq T'$ *and* $T' \preceq T''$*, then* $T \sqsubseteq T''$.

Proof. By case distinction where Definitions 3.2.1 and 3.6.1 are applied. □

The typing of values takes advantage of the fact that the values carry their type at run-time. In other words, every value can be viewed as a *tagged value*. The *tag types* are the value types and `ref`. The tag `ref` is used for object references independently of their actual type. The values are constrained by the following well-formedness conditions:

[value well-formedness] For any tagged value $\langle val, T\rangle$, it holds:

$$\begin{array}{llll}(T \in PrimitiveType & \Rightarrow & val \in SimpleVal) & \wedge \\ (T \in Struct & \Rightarrow & val \in StructVal) & \wedge \\ (T = \texttt{ref} & \Rightarrow & val \in ObjRef) & \end{array}$$

To type values, we recursively define the function $type : Map(Val, Type \cup BoxedType)$, which, given a value, returns the *least* type (with respect to the run-time type compatibility $\sqsubseteq$) the value can be *typable* with. One crucial point is how to define what it means that a struct value is of that struct type. The idea is that, since a struct value is a tuple consisting of the values of its instance fields, every instance field should hold a value typed according to the corresponding declaration type. Another point is that the function *type* should also distinguish between a boxed object and the value inside a boxed object.

Definition 3.6.2 *For a tagged value $\langle val, T\rangle$, we define:*

$$type(\langle val, T\rangle) = \begin{cases} T, & \text{if } T \in PrimitiveType; \\ T, & \text{if } T \in Struct \text{ and} \\ & type(val(T{::}F)) \sqsubseteq fieldType(T{::}F) \\ & \text{for every } T{::}F \in instFields(T); \\ actualTypeOf(val), & \text{if } T = \texttt{ref} \text{ and } actualTypeOf(val) \in Class; \\ boxed(actualTypeOf(val)), & \text{if } T = \texttt{ref} \text{ and} \\ & actualTypeOf(val) \in ValueType. \end{cases}$$

Note that Definition 3.6.2 is correct as the declared type of a struct instance field cannot be the struct type itself [74].

We assume that the results returned by the static semantics functions are typed correctly:

[static functions' correct typing]

1. for every $lit \in Lit$, $type(valueOfLiteral(lit)) = typeOfLiteral(lit)$;
2. for every $T \in Type$, $type(defVal(T)) = T$;
3. for every $T \in PrimitiveType$ and $val \in SimpleVal$, $type(convertVal(T, val)) = T$;

The following lemma determines the kind of each value depending on the types the value is typable with.

Lemma 3.6.2 *Assume $type(val) \sqsubseteq T$.*

1. *If $T \in PrimitiveType$, then $val \in SimpleVal$.*
2. *If $T \in Struct$, then $val \in StructVal$.*
3. *If $T \in RefType$, then $val \in ObjRef$.*

Proof. By the condition **[value well-formedness]** and Definitions 3.6.2 and 3.6.1. □

Lemma 3.6.3 is used to show that the fields of a struct value (so, a defined value) are defined.

Lemma 3.6.3 *If $val \neq undef$ and $type(val) \sqsubseteq S$ where $S \in Struct$, then $val(S{::}F) \neq undef$ for every $S{::}F \in instFields(S)$.*

Proof. By *reductio ad absurdum* and making use of Lemma 3.6.2 and Definition 3.6.2. $\square$

An interesting part consists in defining what addresses are *valid addresses*, *i.e.*, addresses of managed memory blocks. The key idea is that, if a memory block is allocated to store a struct value, then it is considered that also the memory blocks for storing the instance fields of the struct value are allocated. Therefore, we introduce the predicate *isAddressIn* with the following meaning. For two addresses *adr*, *adr'* and two types T, T', $\mathit{isAddressIn}(\mathit{adr}, T, \mathit{adr}', T')$ holds if a value of type T is stored at the address *adr* which ”is an address in” the block of memory pointed to by the address *adr'* where a value of type T is stored.

Definition 3.6.3 (”is address in”) *Given the addresses adr, adr' and the types T, T', we define:*

$$
\begin{array}{l}
\mathit{isAddressIn}(\mathit{adr}, T, \mathit{adr}', T') :\Leftrightarrow \\
\quad (\mathit{adr} = \mathit{adr}' \wedge T = T') \vee \\
\quad (T' \in \mathit{Struct} \wedge \\
\qquad \exists\, T'{::}F \in \mathit{instFields}(T') \ : \\
\qquad\quad \mathit{isAddressIn}(\mathit{adr}, T, \mathit{fieldAdr}(\mathit{adr}', T'{::}F), \mathit{fieldType}(T'{::}F)))
\end{array}
$$

If the address of a struct value ”is an address in” a memory block, then also the address of each instance field of the struct value ”is an address in” that memory block.

Lemma 3.6.4 *If $T \in \mathit{Struct}$ and $\mathit{isAddressIn}(\mathit{adr}, T, \mathit{adr}', T')$, then*

$$\mathit{isAddressIn}(\mathit{fieldAdr}(\mathit{adr}, T{::}F), \mathit{fieldType}(T{::}F), \mathit{adr}', T')$$

for every $T{::}F \in \mathit{instFields}(T)$.

Proof. By case distinction, where Definition 3.6.3 is applied. $\square$

Lemma 3.6.5 *There exists no types T, T', object reference ref, address adr and instance field $T''{::}F \in \mathit{instFields}(\mathit{actualTypeOf}(\mathit{ref}))$ such that*

$$\mathit{isAdddressIn}(\mathit{fieldAdr}(\mathit{ref}, T''{::}F), T, \mathit{adr}, T')$$

Proof. By applying Definition 3.6.3 and using the abstract memory allocation defined in Table 3.19. $\square$

Definition 3.6.4 makes precise what we mean by a *valid address*. An address *adr* is valid for a type T with respect to a stack of frames if *adr* is the address of a local variable or value parameter of declared type T of a frame in the frame stack, or the address of a class instance field of declared type T, or the address of a value of a value type T inside a boxed object, or it ”is an address in” a memory block pointed to by the addresses of one of the above locations.

Definition 3.6.4 (Valid address) *An address adr is* valid *for a type T with respect to a stack of frames* $[fr_1, \ldots, fr_n]$, *denoted* $[fr_1, \ldots, fr_n] \vdash validAddress(adr, T)$, *if one of the following conditions holds:*

(va-loc) *there exists* $i = 1, n$ *and* $loc \in localVars(meth(fr_i))$ *such that* $isAddressIn(adr, T, locAdr(fr_i)(loc), locType(meth(fr_i), loc))$ *holds;*

(va-arg) *there exists* $i = 1, n$ *and* $loc \in valueParams(meth(fr_i))$ *such that* $isAddressIn(adr, T, locAdr(fr_i)(loc), paramType(meth(fr_i), loc))$ *holds;*

(va-field) *there exists* $C \in Class$, $C'::F \in instFields(C)$ *and* $ref \in ObjRef$ *such that* $actualTypeOf(ref) = C$ *and* $isAddressIn(adr, T, fieldAdr(ref, C'::F), fieldType(C'::F))$ *holds;*

(va-box) *there exists* $T' \in ValueType$ *and* $ref \in ObjRef$ *such that* $actualTypeOf(ref) = T'$ *and* $isAddressIn(adr, T, addressOf(ref), T')$ *holds.*

If an address is valid for a struct type, then the address of each of its instance fields is valid for the declared type of the respective field.

Lemma 3.6.6 *If* $[fr_1, \ldots, fr_n] \vdash validAddress(adr, S)$ *and* $S \in Struct$, *then*

$$[fr_1, \ldots, fr_n] \vdash validAddress(fieldAdr(adr, S::F), fieldType(S::F))$$

for every $S::F \in instFields(S)$.

Proof. By Definition 3.6.4 and Lemma 3.6.4. □

The following lemma states that each valid address with respect to a stack of frames remains valid upon pushing a new frame onto the stack[12].

Lemma 3.6.7 *If* $[fr_1, \ldots, fr_n] \vdash validAddress(adr, T)$ *and* $fr \in Frame$, *then*

$$[fr_1, \ldots, fr_n, fr] \vdash validAddress(adr, T)$$

Proof. By Definition 3.6.4. □

Lemma 3.6.8 says that if an address *adr* "is an address in" the memory block pointed to by an address *adr*$'$ where a value of a type T' is stored, and *adr* is the address of a memory block sufficient to hold values of a type T, then the value read from the memory block pointed to by *adr* is of type T. This lemma is used to prove that if the evaluation of a struct field access $^{\blacktriangleright}exp.S::F$ in the case when $isAddressPos(pos)$ holds is required to produce a value (as opposite to an address), then this value is of the expected (compile-time) type.

[12]Note that, the converse implication is false. For example, the address of a local variable of a popped frame is not valid anymore with respect to the new frame stack.

Lemma 3.6.8 *If $isAddressIn(adr, T, adr', T')$ and $type(memVal(adr', T')) \sqsubseteq T'$, then*

$$type(memVal(adr, T)) \sqsubseteq T$$

Proof. By induction on the derivation of *isAddressIn* in Definition 3.6.3 and making use of Definition 3.6.2. □

Assume that adr'' points to a memory block and in this block adr' is an address where a value of a type T' is stored. Lemma 3.6.9 claims that if the value "read" at adr'' is of a type T'', then, after a value of type T' is stored at adr', the value in the memory block pointed to by adr'' is still of type T''. This lemma is applied, for example, to ensure that, following a `ref`/`out` parameter update, the variable to which or to whose "interior" the parameter points has still a value of the expected type.

Lemma 3.6.9 *If the types T, T', T'' satisfy $T' \sqsubseteq T$, $type(val) \sqsubseteq T'$, $type(memVal(adr'', T'')) \sqsubseteq T''$ and $isAddressIn(adr', T', adr'', T'')$, then $type(memVal(adr'', T'')) \sqsubseteq T''$ also holds after the macro* WRITEMEM(adr', T, val) *is executed.*

Proof. By induction on the definition of the *isAddressIn* predicate. The base case of the induction is proved by applying Lemmas 3.5.1 and 3.6.8. □

To prove type safety of method calls, in particular of delegate calls, the following lemma concerning the method lookup is required.

Lemma 3.6.10 *If $lookUp(T, T'::M) = T''::M$, then:*

- $T \preceq T'' \preceq T'$;
- *$length(paramTypes(T'::M)) = length(paramTypes(T''::M))$ and the following relation holds for every $i = 1, length(paramTypes(T'::M)) - 1$*

$$paramTypes(T''::M)(i) = paramTypes(T'::M)(i)$$

- $retType(T'::M) = retType(T''::M)$.

Proof. By Definition 3.5.2 and the condition **[override/implement]**. □

The soundness of the definite assignment analysis is crucial for proving that *undef* does not show up during the evaluation of C$^\sharp$ expressions. The following definition makes precise what variables are guaranteed by the definite assignment analysis to hold a value.

Definition 3.6.5 *The* variables tracked by the definite assignment analysis *are defined by the following grammar:*

$$\mathit{DefAssVar} \quad ::= \quad \mathit{Loc} \mid \mathit{StructDefAssVar}.\mathit{Field}$$

where StructDefAssVar represents a DefAssVar of a struct type.

The following lemma states that every variable tracked by the definite assignment analysis is in particular a *Vexp*.

Lemma 3.6.11 *If exp* $\in$ *DefAssVar, then exp* $\in$ *Vexp.*

Proof. By Definition 3.6.5 and the definition of *Vexp* in Figure 3.1. □

The recursively defined function $\mathit{adrOfVexp} : \mathit{Map}(\mathit{Frame} \times \mathit{DefAssVar}, \mathit{Adr})$ determines the addresses of the variables tracked by definite assignment analysis.

Definition 3.6.6 *Given vexp* $\in$ *DefAssVar in the method executed in a frame fr, we define:*

$$\mathit{adrOfVexp}\,(\mathit{fr}, \mathit{vexp}) = \begin{cases} \mathit{locAdr}(\mathit{fr})(\mathit{loc}), & \text{\textit{if vexp is a local variable or parameter loc};} \\ \mathit{fieldAdr}(\mathit{adrOfVexp}\,(\mathit{fr}, \mathit{vexp}'), \mathcal{ST}(\alpha)::F), & \text{\textit{if vexp is of the form} } {}^{\alpha}\mathit{vexp}'.F. \end{cases}$$

As the definite assignment analysis is sound (Section 3.4), the methods accepted by this analysis should satisfy the following condition:

[definite assignment analysis] If ${}^{\alpha}\mathit{vexp} \in \mathit{DefAssVar}$ and α occurs in the method executed in a frame *fr*, then

$$\neg\,\mathit{isAddressPos}(\alpha) \vee \mathit{label}(\mathit{up}(\alpha)) \in \{\mathit{Aop}, \texttt{ref}\} \;\Rightarrow\; \mathit{memVal}(\mathit{adrOfVexp}(\mathit{fr}, \mathit{vexp}), \mathcal{ST}(\alpha)) \neq \mathit{undef}$$

So, a variable is *definitely* assigned unless the variable occurs at an address position, and it is neither a variable which is assigned to in a compound assignment nor a `ref` argument. It worths mentioning that, although a variable which is assigned to in a compound assignment or is passed as a `ref` argument, is at an address position, the variable is required to have a value according to the definite assignment rules [74, §5.3].

Finally, we state the type safety result in Theorem 3.6.1. The invariants of this theorem are categorized into *frame invariants*, *global invariants*, and *dynamic method chain invariants*.

Frame invariants If an expression evaluates to a value, then the value is compatible with the compile-time type associated to the expression position **(val)**. If an expression evaluates to an address, then the address is a valid address for the compile-time type of the expression position with respect to the substack of frames under the current frame **(adr)**. The constant *undef* never occurs in the evaluation tree **(undef)**. Expressions and arguments (other than `ref`/`out` arguments) not occurring in address positions are ensured to evaluate to values (as opposite to addresses), whereas the `ref`/`out` arguments are guaranteed to evaluate to addresses **(label1)**. Expressions occurring in address positions evaluate to addresses **(label1)**. If a statement is executed, then its execution can only complete normally or with an abruption reason **(label2)**. The `ref`/`out` parameters can only point to valid addresses for their declaration type with respect to the substack of frames under the current frame **(arg1)**. The value parameters are guaranteed to have values of their declared type **(arg2)**. The local variables and the fields of struct type local variables are either uninitialized, *i.e.*, *undef*, or have values of their declared type **(loc)**. The `catch` parameters always hold values of types compatible with the type of exceptions the `catch` clauses are handling. Every position where a `Break` or a `Continue` occurs is within a `while` statement **(abr1)**. A `Goto`(lab) can only occur in the *scope* of the statement labelled with the label *lab*, *i.e.*, in the block in which the labelled statement occurs **(abr2)**. Every position where `Return` or `Return`(val) occurs is within a method body, with a return value of a type compatible with method return type **(abr3)**, **(abr4)**. The types of thrown exceptions are subtypes of `object`.

Global invariants The instance fields of an object (other than a boxed object) have values of their declared type **(field)**. The value inside a boxed value type is compatible with the value type **(box)**. Every target object in the invocation list of a delegate is an object of an appropriate type: (1) if it is a boxed object, then the value type being boxed is compatible with the type of the corresponding target method; (2) if it is not a boxed object, then the run-time type is compatible with the type that declares the corresponding target method. Every target method is consistent with the delegate type **(del)**. Every exception in the *excStack* is an object reference **(excstack)**.

Dynamic method chain invariants The current position of a caller frame is either the position of an instance creation or the position of a method call **(framestack1)**. If the return type of the method associated to a callee frame is not `void`, then the return type is compatible with the compile-time type expected by the caller frame **(framestack2)**. If a callee frame corresponds to an instance constructor call, then its `this` pointer is the newly allocated address (in case of a struct constructor) or the newly created reference (in case of a class constructor) **(framestack3)**.

Theorem 3.6.1 (Type safety) *Assume that frameStack is given by* $[fr_1, \ldots, fr_{n-1}]$ *and the current frame is a frame* fr_n. *Moreover, assume that* $meth(fr_i)$ *is well-typed for every* $i = 1, n$. *Then the following invariants hold for every frame* $fr_i = (meth^*, pos^*, locAdr^*, values^*)$, $i = 1, n$:

(val) *If* $label(\alpha) \in Exp$ *and* $values^*(\alpha) = val$ *where* $val \in Val$, *then* $type(val) \sqsubseteq \mathcal{ST}(\alpha)$.

(adr) *If* $label(\alpha) \in Exp$ *and* $values^*(\alpha) = adr$ *where* $adr \in Adr$, *then*

$$[fr_1, \ldots, fr_i] \vdash validAddress(adr, \mathcal{ST}(\alpha))$$

(undef) $values^*(\alpha) \neq undef$ *for every* $\alpha \in dom(values^*)$.

(label1) *If* $label(\alpha) \in Exp \cup Arg$ *and* $\alpha \in dom(values^*)$, *then*

$$\begin{array}{lll} (\neg isAddressPos(\alpha) & \Rightarrow & (label(\alpha) \in \{\texttt{ref}, \texttt{out}\} \Rightarrow values^*(\alpha) \in Adr) \wedge \\ & & (label(\alpha) \notin \{\texttt{ref}, \texttt{out}\} \Rightarrow values^*(\alpha) \in Val)) \quad \wedge \\ (isAddressPos(\alpha) & \Rightarrow & values^*(\alpha) \in Adr) \end{array}$$

(label2) *If* $label(\alpha) \in Stm$ *and* $\alpha \in dom(values^*)$, *then*

$$values^*(\alpha) \in \{\texttt{Norm}\} \cup Abr$$

(arg1) *If* $loc \in refParams(meth^*) \cup outParams(meth^*)$, *then*

$$[fr_1, \ldots, fr_i] \vdash validAddress(locAdr^*(loc), paramType(meth^*, loc))$$

(arg2) *If* $loc \in valueParams(meth^*)$, *then*

$$type(memVal(locAdr^*(loc), paramType(meth^*, loc))) \sqsubseteq paramType(loc, meth^*)$$

(loc) *If* $loc \in localVars(meth^*)$, *then*

$$memVal(adr, T) = undef \text{ or } type(memVal(adr, T)) \sqsubseteq T$$

for every $(adr, T) \in Adr \times Type$ *with* $isAddressIn(adr, T, locAdr^*(loc), locType(meth^*, loc))$.

(catch) *If* pos^* *is in the scope of a* `catch` *parameter loc of type T, then*

$$type(memVal(locAdr^*(loc), T)) \sqsubseteq T$$

(abr1) *If* $values^*(\alpha) \in \{\texttt{Break}, \texttt{Continue}\}$, *then* α *is in a* `while` *statement.*

(abr2) *If* $value^*(\alpha) = \texttt{Goto}(lab)$, *then* α *is in the scope of a statement labelled with lab.*

(abr3) *If* $values^*(\alpha) = \texttt{Return}$, *then* $retType(meth^*) = \texttt{void}$.

(abr4) *If* $values^*(\alpha) = \texttt{Return}(val)$, *then* $val \in Val$ *and* $type(val) \sqsubseteq retType(meth^*)$.

(abr5) *If* $values^*(\alpha) = \texttt{Exc}(ref)$, *then* $type(ref) \sqsubseteq \texttt{object}$.

The following global invariants are satisfied:

(field) *If ref* $\in$ *ObjRef is such that actualTypeOf*$(ref) = C$ *where* $C \in$ *Class, then*

$$type(memVal(fieldAdr(ref, C'::F), fieldType(C'::F))) \sqsubseteq fieldType(C'::F)$$

for every $C'::F \in instFields(C)$.

(box) *If ref* $\in$ *ObjRef is such that actualTypeOf*$(ref) = T$ *where* $T \in$ *ValueType, then*

$$type(memVal(addressOf(ref), T)) \sqsubseteq T$$

(del) *If* $d \in$ *ObjRef is such that actualTypeOf*$(d) = D$ *where* $D \in$ *Delegate, then the following hold for every* $(ref, T::M) \in invocationList(d)$*:*

- $ref \in ObjRef$
- *If there exists* $T' \in$ *ValueType such that* $type(ref) = boxed(T')$*, then* $T' \sqsubseteq T$*. Otherwise,* $type(ref) \sqsubseteq T$*.*
- $T::M$ *and* D *are consistent.*

(excstack) *If* $ref \in excStack$*, then* $type(ref) \sqsubseteq$ `object`.

If $(\_, \beta, \_, values^*)$ *is the parent frame of* $(T::M, \_, \_, locAdr^*)$*, then the following dynamic method chain invariants are satisfied:*

(framestack1) β *is the position of an instance creation or of a method call.*

(framestack2) *If* $retType(T::M) \neq$ `void`*, then* $retType(T::M) \sqsubseteq \mathcal{ST}(\beta)$*.*

(framestack3) *If* $T::M$ *is an instance constructor and* $values^*(\beta) = valadr.T::M(valadrs)$*, then*

- *If* $T \in$ *Struct, then* $locAdr^*($`this`$) = valadr$*.*
- *If* $T \in$ *Class, then* $memVal(locAdr^*($`this`$), T) = valadr$*.*

Proof. For the proof of several invariants, *e.g.*, **(val)**, **(adr)**, **(undef)**, the following (frame) invariants corresponding to the frame fr_i are required:

(adr1) *If* $values^*(\alpha) = adr$ *where* $adr \in Adr$*, then*

$$memVal(adr, \mathcal{ST}(\alpha)) = undef \text{ or } type(memVal(adr, \mathcal{ST}(\alpha))) \sqsubseteq \mathcal{ST}(\alpha)$$

(vexp) *If* $\alpha \in$ *Vexp and* $values^*(\alpha) = adr$ *where* $adr \in Adr$ *satisfies* **(va-loc)** *for* $\mathcal{ST}(\alpha)$ *with respect to the stack* $[fr_1, \ldots, fr_i]$*, then* $\alpha \in DefAssVar$ *and* $adrOfVexp(fr_i, vexp) = adr$*, where vexp is the expression at* α*.*

The invariant **(adr1)** says that if an address appearing in the evaluation tree holds a value, then the value is compatible with the static type associated to the position where the address occurs. The invariant **(vexp)** establishes a correspondence between certain addresses from the evaluation tree and variables tracked by the definite assignment analysis. Assume that an address *adr* appearing at an address position of a variable expression *vexp* has been allocated as the address of a local variable of a frame in the stack (*not* necessarily the current frame). Then *vexp* is a variable tracked by the definite assignment analysis, whose address with respect to the current frame is *adr*.

The proof of all invariants, including **(adr1)** and **(vexp)**, proceeds by induction on the number of steps in the run of the operational semantics model.

Base case In the initial state of the semantics model, the invariants are trivially satisfied.

Induction step There is a proof case for each of the rules of the operational semantics model. We only do here one interesting proof case.

Case *pos* is pointing to *valadr* in *valadr*.T::F.

Of particular interest is the application of the soundness of the definite assignment analysis in the proof. To illustrate that, we only study here the subcase characterized by

$$\begin{array}{ll} & \mathcal{ST}(pos) \not\preceq \mathit{ValueType} \lor \mathit{valadr} \in \mathit{Adr} \\ \text{and} & \neg\, \mathit{isAddressPos}(\mathit{up}(pos)) \\ \text{and} & \mathit{valadr} \neq \texttt{null} \end{array} \tag{3.1}$$

In this case, the operational semantics model fires the updates described by the macro YIELDUPINDIRECT($\mathit{fieldAdr}(\mathit{valadr}, T{::}F)$). More exactly, $\mathit{values}(\mathit{up}(pos))$ is set to the value $\mathit{memVal}(\mathit{fieldAdr}(\mathit{valadr}, T{::}F), \mathit{fieldType}(T{::}F))$ (note that the assumption (3.1) implies that $\neg\, \mathit{isAddressPos}(\mathit{up}(pos))$, and so, a value – as opposed to an address – is required at $\mathit{up}(pos)$) and *pos* becomes $\mathit{up}(pos)$. We distinguish the following cases:

Subcase 1 $\mathit{isAddressPos}(pos)$ is false.

By $\neg\, \mathit{isAddressPos}(pos)$ and the induction hypothesis – that is, the invariant **(label1)** – we get $\mathit{values}(pos) \in \mathit{Val}$. By this and the induction hypothesis **(val)**, we have $\mathit{type}(\mathit{valadr}) \sqsubseteq \mathcal{ST}(pos)$.

As *meth* is well-typed, according to the type constraints in Table 3.7, we have $\mathcal{ST}(pos) \preceq T$. By (3.1) and $\mathit{values}(pos) \notin \mathit{Adr}$, we know that $\mathcal{ST}(pos) \not\preceq \mathit{ValueType}$, that is $\mathcal{ST}(pos) \not\preceq \mathit{RefType}$. This and Lemma 3.6.2 imply that $\mathit{valadr} \in \mathit{ObjRef}$.

That the invariant **(val)** holds in the next step of the semantics model follows from $\mathit{valadr} \in \mathit{ObjRef}$ and the induction hypothesis **(field)**. So, we have

$$\mathit{type}(\mathit{memVal}(\mathit{fieldAdr}(\mathit{valadr}, T{::}F), \mathit{fieldType}(T{::}F))) \sqsubseteq \mathit{fieldType}(T{::}F)$$

By this and Lemma 3.6.2, we get $\mathit{values}(\mathit{up}(pos)) \notin \mathit{Adr}$. This implies that the invariant **(label1)** holds in the next step of the semantics model.

The other invariants are trivially satisfied.

Subcase 2 $\mathit{isAddressPos}(\mathit{pos})$ is true.

By $\mathit{isAddressPos}(\mathit{pos})$ and the induction hypothesis **(label1)**, we get $\mathit{valadr} \in \mathit{Adr}$. By this, $\mathit{values}(\mathit{pos}) = \mathit{valadr}$ and the induction hypothesis **(adr)**, we know that $[\mathit{fr}_1, \ldots, \mathit{fr}_i] \vdash \mathit{validAddress}(\mathit{adr}, \mathcal{ST}(\mathit{pos}))$ where $[\mathit{fr}_1, \ldots, \mathit{fr}_i]$ is the current *frameStack* with the current frame on top. According to Definition 3.6.4, this means that *valadr* satisfies one of the conditions **(va-loc)**, **(va-arg)**, **(va-field)** or **(va-box)** with respect to $\mathcal{ST}(\mathit{pos})$. If *valadr* fulfills **(va-arg)**, **(va-field)** or **(va-box)**, then the fact that **(val)** holds in the next step of the model follows by Lemma 3.6.8, the type constraints in Table 3.7 and the induction hypothesis, that is, the invariants **(arg2)**, **(field)**, or **(box)**, respectively. In turn, the invariant **(val)** for $\mathit{up}(\mathit{pos})$ also implies that **(label1)** is satisfied in the next step of the semantics model.

Let us now assume that *valadr* satisfies **(va-loc)**, that is, there exists $j = 1, i$ and $\mathit{loc} \in \mathit{localVars}(\mathit{meth}(\mathit{fr}_j))$ such that

$$\mathit{isAddressIn}(\mathit{valadr}, \mathcal{ST}(\mathit{pos}), \mathit{locAdr}(\mathit{fr}_j)(\mathit{loc}), \mathit{locType}(\mathit{meth}(\mathit{fr}_j), \mathit{loc})) \tag{3.2}$$

By $\mathit{isAddressPos}(\mathit{pos})$ and Lemma 3.5.2, we get $\mathit{pos} \in \mathit{Vexp}$. By this and the induction hypothesis – that is, by the special invariant **(vexp)** – we determine that $\mathit{pos} \in \mathit{DefAssVar}$ and $\mathit{adrOfVexp}(\mathit{fr}_i, \mathit{vexp}') = \mathit{valadr}$, where vexp' is the expression at *pos*.

By $\mathit{isAddressPos}(\mathit{pos})$ and Definition 3.5.5, we have $\mathcal{ST}(\mathit{pos}) \in \mathit{Struct}$. It then follows $\mathcal{ST}(\mathit{pos}) = T$ by Definition 3.2.1, $\mathcal{ST}(\mathit{pos}) \preceq T$ (by the type constraints), and the fact that the $\mathcal{ST}(\mathit{pos})$'s supertypes that are reference types, *i.e.*, `System.ValueType`, `object` and possible interface types, do not declare instance fields.

By $\mathcal{ST}(\mathit{pos}) \in \mathit{Struct}$ and $\mathit{pos} \in \mathit{DefAssVar}$, we derive that $\mathit{up}(\mathit{pos}) \in \mathit{DefAssVar}$. We also have $\mathit{up}(\mathit{pos}) \in \mathit{Vexp}$ by Lemma 3.6.11. Let *vexp* be the variable expression at $\mathit{up}(\mathit{pos})$.

We can now apply the soundness of the definite assignment analysis. Thus, by **[definite assignment analysis]** applied for $\mathit{up}(\mathit{pos})$ and (3.1), we have

$$\mathit{memVal}(\mathit{adrOfVexp}(\mathit{fr}_i, \mathit{vexp}), \mathcal{ST}(\mathit{up}(\mathit{pos}))) \neq \mathit{undef} \tag{3.3}$$

According to Definition 3.6.6, we have

$\mathit{adrOfVexp}(\mathit{fr}_i, \mathit{vexp}) = \mathit{fieldAdr}(\mathit{adrOfVexp}(\mathit{fr}_i), \mathit{vexp}', \mathcal{ST}(\mathit{pos})::F) = \mathit{fieldAdr}(\mathit{valadr}, T::F)$.

By this, $\mathcal{ST}(\mathit{up}(\mathit{pos})) = \mathit{fieldType}(T::F)$ (see the type constraints in Table 3.7) and (3.3), we derive

$$\mathit{memVal}(\mathit{fieldAdr}(\mathit{valadr}, T::F), \mathit{fieldType}(T::F)) \neq \mathit{undef} \tag{3.4}$$

By (3.2) and Lemma 3.6.4, we get

$$\mathit{isAddressIn}(\mathit{fieldAdr}(\mathit{valadr}, T::F), \mathit{fieldType}(T::F), \mathit{locAdr}(\mathit{fr}_j)(\mathit{loc}), \mathit{locType}(\mathit{meth}(\mathit{fr}_j), \mathit{loc}))$$

By this, (3.4) and the induction hypothesis **(loc)**, it follows that the invariant **(val)** is satisfied in the next step of the operational semantics model. The proof of the other invariants is trivial. □

3.7 Related Work

The origin of type safety proofs goes back to the *subject reduction theorem* for typed λ-calculus (for details, see [101]), but starts in earnest with Milner's slogan "*Well-typed programs never go wrong*" [94] in the context of ML. The first statically typed language proved to have a type-hole in its type system is Eiffel [37]. Since then, designers of object-oriented programming languages aimed to prove type safety of their languages (see, for example, the papers [30, 31]).

To the best of our knowledge, our work is the *first* type safety proof for a fragment of C$\sharp$. The most related to our proof are the Java's type safety proofs, referred to in Section 3.7.1. Section 3.7.2 reports on formal specifications of the definite assignment analysis performed by the Java compiler. A brief evaluation of the ASMs applications in the area of programming languages is included in Section 3.7.3.

3.7.1 Java's Type Safety

Closely related to our work is the considerable research for proving Java's type safety [40, 118, 98, 41, 121, 77, 115]. An interested reader can find in [73] an excellent and comprehensive evaluation of this research topic. The most closely related, Stärk *et al.* [115]'s type safety proof is the one that had the most impact on our work. Börger and Stärk [28] provide a concrete comparison of the Java semantics model [115] and C$\sharp$ semantics model [24] and formulate in a precise technical manner where and in which respect the two languages differ among each other and from other programming languages – methodologically, semantically, and pragmatically.

C$\sharp$ introduces several new features with respect to Java, *e.g.*, structs, call-by-reference, an unified type system, delegates. Due to the rich and unified type system, the C$\sharp$'s evaluation rules use the type information, and concerning this aspect, C$\sharp$ contrasts with most of the Java fragments analyzed so far, where the evaluation does not require any type information. Another crucial modelling difference with respect to Java is the "indirection through memory addresses", needed for the call-by-reference mechanism. This has significantly contributed to the complexity of the C$\sharp_S$'s type safety proof: As one can see in Section 3.6, most of the definitions and lemmas required for Theorem 3.6.1 are concerning the handling of addresses. Moreover, the treatment of the addresses gets even trickier in the context of structs and the C$\sharp$'s unified type system, enabled by boxing and unboxing. On the other hand, as a noticeable difference between Java's and C$\sharp$'s type safety proofs, dealing with the unified type system triggers introducing the special "boxed types".

3.7.2 The Definite Assignment Analysis

A definite assignment analysis is also performed by the Java compiler. This analysis has been formally specified in terms of *data flow equations* and *type systems* by Stärk *et al.* [115] and Schirmer [105], respectively. The [115]'s data flow equations approach has inspired us the most. Stärk *et al.* [115] have related the definite assignment analysis not only to the type safety proof, but also to the problem of generating verifiable JVM bytecode from legal Java source code programs. Schirmer [105] has carried out the formalization of the definite assignment analysis and the proofs in the theorem prover Isabelle/HOL [100].

The C$\sharp$ definite assignment analysis differs from the one of Java in many respects. Due to the C$\sharp$ `goto` statement, the C$\sharp$'s analysis gets tricky as it involves a fixed point iteration, which is not the case in Java. Therefore, Schirmer [105]'s approach using type systems is not a feasible way to prove the analysis' soundness (Schirmer [105] himself recognizes this). Note that Java has a `break` *L*; statement, which has no corresponding statement in C$\sharp$. Its treatment is similar to the C$\sharp$ `continue` statement, and so, introduces no complexity. Moreover, the handling of the C$\sharp$ struct type variables represents another crucial difference with respect to Java's analysis. As we have seen, determining their definite assignment status requires considering the so-called "expansion" function, which takes into account the status of the fields of the struct type variables. Another non trivial difference between C$\sharp$ and Java with respect to the definite assignment analysis is given by the treatment of the C$\sharp$ "by reference" arguments: While a Java local variable can only get definitely assigned upon a direct assignment, a C$\sharp$ local variable can also get definitely assigned following a method call with the variable as an `out` argument.

3.7.3 The ASM Method

An important advantage of using the ASM method is that the ASMs are appropriate to describe operational features of languages. Thus, the ASM method was successfully applied to define the dynamic semantics of many widely used languages. Among these are C (see [71]), C++ (see [122]), C$\sharp$ (see [24, 112, 55, 84]), Java (see [115]), Occam (see [72, 23]), Oberon (see [90]), Prolog (see [25]), Smalltalk (see [95]).

For work centered around Java, [18] contains a collection of formal-method-approaches to language specification and analysis. Hartel and Moreau make an evaluation of the ASM-based Java investigations [73]. Given that a semantics model for C$\sharp$ has only been defined by [24], we cannot perform here a similar evaluation.

If appropriately designed, the ASM models are flexible enough to be refined in order to cover other language features. For example, the C$\sharp$ model [24] has been firstly extended by Stärk and Börger with a thread and memory model [112], and then by Jula with the main new features of C$\sharp$ (v2.0), in particular generics, anonymous methods, and iterators [83].

Another strength of the ASM models is that they can be validated and verified against the real systems they model. A purely theoretical specification often contains subtle errors, which are sometimes very hard to discover. An implementation of the theoretical specification can reveal such errors, by showing a misbehavior during its execution. Thus, in [84], we have refined the C$\sharp$ model to .NET executable AsmL code [49], somehow similarly to the AsmGofer refinement developed by Schmid [106, 107] for the Java model [115].

3.8 Summary and Future Work

We proved type safety of a large fragment of C$\sharp$ which includes delegates, call-by-reference, structs, boxing, and unboxing. To our knowledge, the treatment of these features seems to be new. To prove type safety, we have specified the type constraints that should be satisfied by a method accepted by the C$\sharp$ compiler. Further, we have formally defined and proved the soundness of the definite assignment analysis, a crucial ingredient for C$\sharp$'s type safety. We have then identified and formalized the invariants expressing the type safety result. Finally, we proved that the run-time execution, according to the operational semantics model [24], of legal

C$\sharp_{\mathcal{S}}$ programs accepted by the definite assignment analysis and obeying the type constraints preserves the invariants.

As future work, the language C$\sharp_{\mathcal{S}}$ can be extended with additional features of C$\sharp$, *e.g.*, statics and class initialization, asynchronous exceptions[13] (*e.g.*, `OutOfMemoryException`, `StackOverflowException`), generics, access modifiers, without major complications, though this would trigger a substantial expansion in the length of the proof. Future work also includes the specification and verification of our type safety result in the theorem prover Isabelle/HOL [100].

[13]So far, we have formally defined the semantics of the class initialization and of the asynchronous exceptions [58].

4 Type Safety of CLR

Better be despised for too anxious apprehensions, than ruined by too confident security.

Edmund Burke

In this chapter, we present a *formal proof* of CLR's type safety. For this, we begin with specifying the *static* and *dynamic semantics* of the CLR bytecode language by providing an *abstract interpreter*, in terms of an *ASM model*, which executes arbitrary bytecode programs. We then formally define the structural and type consistency checks performed by the bytecode verification. On top of the type checks, we provide a definition of *well-typed* methods and of the *bytecode verification algorithm*. We then prove type safety of well-typed methods, *i.e.*, the execution according to the semantics models of legal and well-typed methods does not lead to any *run-time type violations* and leaves the program in a good state, where certain *structural constraints* hold. Finally, to prove CLR's type safety, we show that the verification algorithm is *sound*, *i.e.*, the methods accepted by the verification are well-typed, and *complete*, *i.e.*, the well-typed methods are accepted by the verification.

Bytecode language decomposition To keep the size of the semantics model manageable and thus to simplify the development and exposition of the type safety proof, we structure the CLR into three *layered modules of orthogonal language features*, namely a *lightweight CLR* (with object classes, value classes, pointers, typed references, inheritance, objects, delegates, dynamic dispatched method calls, tail calls, boxing, unboxing), *exception handling*, and *generics*. This decomposition yields a sequence of bytecode sublanguages, which altogether describe almost the entire bytecode language. Each sublanguage extends its predecessor, and for each one's semantics, we define an ASM model which is a *conservative* (purely incremental) *extension* of its predecessor. Thus, for the lightweight CLR, we build the model CLR$_{\mathcal{L}}$, which is then *refined* (in the sense of *ASM refinements* [22]) by CLR$_{\mathcal{E}}$ to cover the exception handling. In turn, CLR$_{\mathcal{E}}$ is *refined* through CLR$_{\mathcal{G}}$ to deal with generic types and generic methods. The verification type systems and type consistency checks performed by the bytecode verification are defined by *stepwise refinement*, similarly to the layering of the semantics models. For each sublanguage, we develop a type safety proof.

Features omitted from CLR The focus of this chapter is not a full model of the CLR that conforms exactly to the CLR specifications [45] down to every technical detail. Such a model has obviously its merits, but as in every formal development, a balance between abstraction and detail should be determined. As the goal of this thesis is CLR's type safety, besides omitting the *unverifiable code*, the analysis presented here also abstracts as far as possible from everything that does not concern type safety. The bytecode language that we consider does not include *static members*, *arrays*, and *threads*. However, we still consider four static methods: `entrypoint`, *i.e.*, the method that is executed when the code is first run, and, for handling delegates, three static methods declared by the class `System.Delegate` [5]. The omitted features only contribute to the complexity of the system through the number of additional cases they introduce, but they do not introduce any challenging problems.

Chapter outline The rest of this chapter is organized as follows. Section 4.1 specifies the semantics and bytecode verification of the lightweight CLR and proves its type safety. Section 4.2 extends the analysis of the lightweight CLR with exception handling. Finally, the generics are added in Section 4.3, and the resulting bytecode language is showed to be type-safe. The related work is discussed in Section 4.4. Section 4.5 concludes and gives directions for future work.

4.1 The Lightweight CLR

This section analyzes the lightweight CLR and proves its type safety. An informal overview of the lightweight CLR and its type system is provided in Section 4.1.1. Sections 4.1.2 and 4.1.3 define the static and dynamic semantics of the lightweight bytecode language. Bytecode verification is described and formally defined in Section 4.1.4. The bytecode verification algorithm and the definition of well-typed methods are given in Section 4.1.5. Section 4.1.6 introduces the typing rules and proves type safety of the lightweight CLR.

4.1.1 An Overview of the Lightweight CLR and its Type System

The CLR is an abstract stack-based machine, with automatic memory management, for CIL bytecode programs. It encompasses a *managed heap*, for storing objects, and a stack for method calls, for capturing information, in the form of *frames*, about the still active methods.

A new frame is put on the frame stack every time a method is called. Such a frame comprises a *program counter*, any *local variables*, a *local memory pool* (for dynamic allocations), the *arguments* passed to the method (if any), and an *evaluation stack* (for intermediate results of the method's computations).

A method body is a list of bytecode instructions. The *instructions* manipulate the heap, the local variables, the arguments, and the evaluation stack. Mainly, there are three kinds of instructions: instructions that *load* values onto the evaluation stack, instructions that *store* values from the evaluation stack into memory locations, and instructions for *method calls*.

In the rest of this section, we give a detailed informal description of the CLR type system.

4.1.1.1 Reference Types

The CLR supports *reference types* such as object classes, interfaces, pointers, and delegates. A value of a reference type is always allocated on the managed heap.

Object classes An *object class* is a reference type of a "self-describing" value, *i.e.*, a value whose representation and applicable operations are unambiguously defined. The object classes, supporting for example the C$^\sharp$, J$^\sharp$, or Smalltalk classes, can declare instance fields and instance methods (virtual and non-virtual). A value of an object class, also known as an *object class instance*, can be referred to by *object references* and can be created with a constructor. The constructors are special methods, named `.ctor`.

Interfaces Unlike the object classes which fully describe their values, *interfaces* are reference types that are only a partial description. For this reason, an interface cannot declare fields and methods for the values of the interface, *i.e.*, it cannot define instance fields and non-virtual instance methods. However, an interface can declare (abstract) virtual methods.

As there are .NET compliant languages (*e.g.*, Scheme [43], Haskell [14], Mercury [11, 38]) where recursion is the only way to express repetition, the CLR needs to support *tail method calls*. When executing a tail call, the CLR discards the caller's frame prior making the call. This means, for example, that a method which calls itself tail can implement a desired repetition.

Regarding type safety, the tail calls require attention because of two main reasons. Firstly, because their execution results in situations when the method invoked tail does not return to the (in the meantime discarded) frame that contained the tail call, but to another frame. Secondly, because although a call is marked tail, it could happen that the `tail` prefix is ignored when the bytecode is jitted, and consequently the call would be interpreted as a non-tail call. The reason for that is because the JIT compiler views this prefix as an *optional directive*. However, it should be ensured that the call is type safe, regardless of whether the call has been interpreted as tail or not.

Pointers There exists two kinds of *pointers*: managed pointers and method pointers. The CLR also supports *unmanaged pointers*. However, since their use is deemed unverifiable, this thesis does not consider them.

A *managed pointer* is a reference to a managed memory block that has been allocated from the CLR's memory heap. Unlike a reference to an object class instance, a managed pointer can point to the interior of the instance (*e.g.*, to an instance field), rather than to point to the "entry" of the instance. Usually, managed pointers are generated for method arguments that are passed "by-reference", *e.g.*, the C$^\sharp$ `ref`/`out` parameters. The CLR provides two type-safe operations on managed pointer types: loading a value from (operation known as "indirect load") and writing a value (operation known as "indirect store") to the address referenced by a pointer.

A key ingredient in the type safety proof, identifying the typing rules for pointers is fairly complex, somehow comparable with the notion of valid address in the C$^\sharp$'s type safety proof. To type pointers, the notion of "is an address in" introduced in Chapter 3 is reconsidered. Also, ensuring type-safe tail calls can be problematic especially in the presence of pointers, as the tail calls' execution can easily generate dangling pointers.

A *method pointer* is a pointer to a method entry point. It reliably identifies the method assigned to the pointer. Thus, if one knows the method entry point, one can determine, for example, the method signature, the return type[1], the local variable types. Therefore, one can see a method pointer as a *method identifier*. This identifier can be regarded as pointing to an address or directly to the corresponding method reference. The only operations on method pointers are the call of a method via the corresponding method pointer and the creation of delegates. As the bytecode verification does not track method pointer types, the first operation, also known as *indirect call*, is regarded as *unverifiable* by [45] and therefore not considered for our analysis.

Delegate classes Similarly as in C$^\sharp$, the delegates were built with the idea of being the verifiable method pointers. Informally, a delegate wraps a list of method pointers called *invocation list*. Exactly as in C$^\sharp$, the methods pointed to in the invocation list can be invoked *sequentially*, in the order in which they appear through a delegate call. More formally, a *delegate* is simply an instance of a delegate class. Figure 4.1 shows the general form of a delegate class. A *delegate class* is an object class whose definition obeys certain conditions:

- It should be declared as a `sealed` subclass of `System.Delegate`.
- It should declare exactly two members: a `.ctor` and a method `Invoke`. As the implementations of these two methods are provided by the run-time environment, the methods should have empty bodies.

[1]In CLR, the signature of a method reference does not include method return type.

Figure 4.1 The general form of delegate classes.

```
.class private auto ansi sealed DC extends System.Delegate
{
        .method public hidebysig specialname rtspecialname
                instance void .ctor(object o, native int m) runtime managed
       {
       } // end of method DC::.ctor
       .method public hidebysig
                 instance T Invoke(T1 x1,...,Tn xn) runtime managed
       {
       } // end of method DC::Invoke
} // end of class DC
```

- The `.ctor` should have exactly two parameters: the first of type `object` and the second of type `native int` (see Section 4.1.1.2 for details on this type). When the constructor is invoked, the first argument – known as *target object* – is an instance of the class or one of its subclasses that declares the target method, and the second argument is a method pointer to the method – known as *target method* – to be invoked. The second parameter of the `.ctor` should be of type `native int`. This is so since *the method pointers are regarded as values of the type* `native int`.
- The `Invoke` method should have the signature *compatible* with the target method.

Similarly as in C$^\sharp$, each call to an `Invoke` method of a delegate class is turned into a synchronous call of the methods pointed to in the invocation list. The fact that `Invoke` and the target method do not necessarily have the same signature might be problematic in ensuring type safety, and therefore Definition 4.1.8 of compatible signatures requires attention. By only allowing two methods, the delegates can, on one hand, be easier controlled. Because the target object and the target method are passed to the delegate constructor, there exists the information to ensure that run-time (type) errors caused by mismatched parameters and return types do not occur. On the other hand, specifying the delegates semantics is not an easy task because of their ”opaque” implementation: Both the creation and invocation of a delegate are accomplished through methods whose bodies, assumed to be empty, are automatically generated by the run-time system. Concerning the creation of a delegate instance, there are also verification rules which bound to a certain bytecode sequence.

4.1.1.2 Value Types

The *value types* supported by the CLR are the primitive types and the value classes. Unlike the values of reference types, the values of a value type are usually allocated “in place”, *i.e.*, on the evaluation stack. A value type can also be allocated on the heap, but only within (*e.g.*, as a field of) an object class instance.

Primitive types We consider only the following *primitive types*: `int32`, `int64`, `float64`, and `native int`. The type `native int` is mapped at run-time to the natural size of the specific architecture: `int32` on a 32-bits architecture and `int64` on a 64-bits architecture. It becomes efficient when the target machine architecture is not known until run-time.

Figure 4.2 Example: typed references and methods with variable number of arguments.

```
// the __arglist keyword represents
// a method with a variable number of arguments
public static void WriteAtConsole( __arglist )
{
  // an ArgIterator object is used to loop through the arguments
  ArgIterator iterator = new ArgIterator( __arglist );

  // each item in the ArgIterator object is a typed reference,
  // which can be converted to an object using the
  // static method TypedReference.ToObject()
  while ( iterator.GetRemainingCount() > 0 )
  {
    TypedReference tr = iterator.GetNextArg();
    Console.WriteLine ( TypedReference.ToObject(tr) );
  }
}

// the Main method calls the method with
// a variable number of arguments
public static void Main()
{
  // define arguments of different types
  double z = 2.5;
  int x = 31;
  string y = "hello";

  // call the variable argument method, passing
  // the arguments in, using the __arglist keyword
  WriteAtConsole( __arglist( x, y, z ) );
}
```

Value classes The value classes are the CLR support not only for the C♯ structs, but also for C++ structs, Pascal and Modula 2 record types. A *value class* is a value type whose values are "self-contained" values. More exactly, these values – also known as *instances* of the value class – are represented as *mappings* assigning values to the instance fields of the value class. Every value class should have the library object class `System.ValueType` [10] as the direct base class. In particular, it is not possible for a value class to extend another value class. Additionally, a value class can implement one or more interfaces. Similarly to an object class, a value class can declare instance fields. It can only declare non-virtual instance methods.

Just in the same manner as C♯, the CLR supports *boxing* to fill the gap between value types and reference types.

A non-virtual method receives a `this` pointer that is a managed pointer (see below the pointer types) to the (unboxed) value class. One fundamental difference between value classes and object classes is that the formers do not inherit behavior, and therefore cannot declare virtual methods. However, virtual methods defined by `System.ValueType` and by the interfaces implemented by a value class (if any) can be called on that value class instances, but only if the instances are boxed. This makes sense since the value types have identity only when boxed.

`System.TypedReference` [9] is a special value class in the .NET Base Class Library.

Universe of types			Typical use
Type	$=$	$\mathit{RefType} \cup \mathit{ValueType}$	T, T', T'', T'''
$\mathit{RefType}$	$=$	$\mathit{ObjClass} \cup \mathit{Interface} \cup \mathit{PointerType}$	–
$\mathit{ValueType}$	$=$	$\mathit{ValueClass} \cup \mathit{PrimitiveType}$	–
$\mathit{ObjClass}$			OC, OC', OC''
$\mathit{DelegateClass}$	$\subseteq$	$\mathit{ObjClass}$	DC
$\mathit{Interface}$			I
$\mathit{PointerType}$	$=$	$\{T\& \mid T \in \mathit{ReferentType}\}$	$T\&$
$\mathit{ReferentType}$	$=$	$\mathit{ObjClass} \cup \mathit{Interface} \cup \mathit{ValueType}$	–
$\mathit{ValueClass}$			VC
$\mathit{PrimitiveType}$	$=$	{`int32`, `int64`, `native int`, `float64`}	–

Table 4.1: CIL's types.

This class instances are known as *typed references*. They enable the CLR to provide C++-style support for methods that have a *variable* number of arguments. Thus, they can be used to represent method arguments in variable argument lists. In general, the typed references support languages that require "by-reference" passing of unboxed values to methods that are not statically restricted as to the type of values they accept. Therefore, a typed reference is regarded as an *opaque descriptor* of a pointer and a type.

Example 4.1.1 *Figure 4.2 lists a* C$^\sharp$ *example to illustrate the use of typed references. Each of the arguments x, y, and z of the method WriteAtConsole is "packed" into a typed reference. As the declaration of WriteAtConsole does not have a list of parameter types (so, it can have a variable number of arguments), the types of its arguments are determined from the corresponding typed references. For example, the value class* `System.TypedReference` *defines a method* `GetTargetType` *that "extracts" the type information out of a typed reference.*

4.1.2 The Static Semantics

The *execution environment* in which every method runs consists of the type hierarchy and the field and method declarations. Table 4.1 defines the classification of the CIL's *types* described in Section 4.1.1. We also denote by *Class* the set of object classes and value classes:

$$\mathit{Class} = \mathit{ObjClass} \cup \mathit{ValueClass}$$

We assume that the classes and interfaces are organized by the execution environment into an *inheritance hierarchy*, on which the *subtype relation* $\preceq$ defined in Definition 4.1.1 depends.

Definition 4.1.1 (Subtype relation) *The* subtype relation $\preceq$ *is the least reflexive and transitive relation such that*

- *if* $T \in \mathit{Class} \cup \mathit{Interface}$ *and* T' *is* `object`*, or*
- *if* $T \in \mathit{ValueType}$ *and* T' *is* `System.ValueType`*, or*

Universe of values			Typical use
Val	=	$SimpleVal \cup ObjRef$	
	$\cup$	$Map(FRef, Val)$	
	$\cup$	$Adr \cup TypedRef \cup MethPtr$	val, val'
SimpleVal			–
ObjRef			ref, ref', ref''
Adr			adr, adr', adr''
TypedRef			tr
MethPtr			mp

Table 4.2: CIL's values.

Universe			Typical use	Element name
Meth			M	method names
MRef			$mref$, $T{::}M$, $C{::}M$	method references
Instr			$instr$	bytecode instructions
Local	=	$\mathbb{N}$	n	local variables
Arg	=	$\mathbb{N}$	n	arguments
FRef			$C{::}F$, $OC{::}F$, $OC{::}F'$, $VC{::}F$	field references

Table 4.3: Environment specific universes.

- *if $T \in ObjClass$ and T' is a base class of T or an interface implemented by T, or*
- *if $T \in Interface$ and T' is an interface implemented by T, or*
- *if $T \in ValueType$ and T' is an interface implemented by T,*

then $T \preceq T'$.

The possible kinds of values corresponding to the CIL's types are gathered in Table 4.2. Thus, a value can be

- a simple value, element of the universe *SimpleVal*, *i.e.*, a 32-bit integer[2], a 64-bit integer, a native size integer, or a 64-bit floating point number, or
- an object reference, element of the universe *ObjRef*, or
- a value class instance, element of $Map(FRef, Val)$, defined as a mapping assigning values to the instance fields of the value class, or
- a pointer (referencing an address), element of the universe *Adr*, or
- a typed reference, element of *TypedRef*, or

[2]The type `bool` is omitted in our approach as booleans are treated as 32-bit integers: 0 is used for *False*, and 1 for *True*.

- a method pointer, element of *MethPtr*[3].

Each class declares types for a set of instance fields, and each class and interface specifies signatures and return types for a set of instance methods. Every method consists of a list of bytecode instructions. Formally, we consider the universes in Table 4.3 and the environment components in Table 4.4 defined as basic static functions.

The list of instructions of a method body is stored in *code*. Thus, $code(mref)(pos)$ gives the instruction with the code index *pos* of the method *mref*. The abstract bytecode instructions of the bytecode language that we consider are given in Table 4.8. They are constructors for real CIL instructions: One abstract instruction models one or more real instructions.

If omitted in a static or dynamic function, the method reference is considered to be the method of the currently executed frame. For example, $code(pos)$ denotes the instruction with code index *pos* in the current method.

The local variable types are *declared* and *fixed*. They are maintained in the *local signature locTypes*. In the verifiable code, the local variables are assumed to be initialized to default values according to their type. The goal is to simplify the bytecode verification. Due to this decision, the bytecode verification does not need to perform a definite assignment analysis as the C$^\sharp$ compiler does – see Chapter 3, Section 3.4).

For every method, the function *paramTypes* yields the list of parameter types given in the method signature. The list returned by *paramTypes* does not include the type of the `this` pointer used for instance methods. Therefore, we consider the derived function $argTypes : Map(MRef, List(Type))$ which also includes the type of the `this` pointer[4]. For a method declared by an object class (or interface), the `this` pointer is of the class (or interface) type, whereas if the class of the method is a value class, the `this` pointer is a pointer to an instance of the value class.

Definition 4.1.2 *For a method T::M, we define*

$$argTypes(T::M) = \begin{cases} [T] \cdot paramTypes(T::M), & \text{if } T \in ObjClass \cup Interface; \\ [T\&] \cdot paramTypes(T::M), & \text{if } T \in ValueClass. \end{cases}$$

We denote by $locNo : Map(MRef, \mathbb{N})$ and $argNo : Map(MRef, \mathbb{N})$ the length of *locTypes* and *argTypes*, respectively. The return type, the class name, and the name of a method reference are selected by *retType*, *classNm*, and *methNm*, respectively. The set of the fields of a class (including the inherited fields) is maintained in *instFields*. The declared type of any instance field is recorded by *fieldType*.

Besides the execution environment, the static semantics also comprises the universes in Table 4.5 and the basic static functions in Table 4.6.

[3] As the use of a method pointer is very limited, we see the method pointers as elements of the universe *MethPtr* that point to the corresponding method references.

[4] Every time our model exceptionally considers a static method, we make an exception and consider *argTypes* to coincide with *paramTypes*.

Function definition			Function name
code	:	$Map(MRef, List(Instr))$	method code
locTypes	:	$Map(MRef, List(Type))$	local variable types
paramTypes	:	$Map(MRef, List(Type))$	parameter types
retType	:	$Map(MRef, Type \cup \{\texttt{void}\})$	method return type
classNm	:	$Map(MRef, Class)$	method class name
methNm	:	$Map(MRef, Meth)$	method name
instFields	:	$Map(Class, \mathcal{P}(FRef))$	instance fields of a class
fieldType	:	$Map(FRef, Type)$	field declared type

Table 4.4: The components of the execution environment.

Universe	Typical use	Element name
TagType	–	tag type
Op	*op*	operator
Frame	$fr, fr', fr_1, \ldots fr_k$	call frame
Pc	pos, pos', pos''	program counter
InitState	–	object initialization status
Switch	–	execution mode

Table 4.5: Semantics specific universes.

The values carry their *associated CLR types*. Any value can be viewed as a pair consisting of the value and a *tag* representing its CLR type (and *not* the type tracked by the bytecode verification). The set of *tags* is denoted by *TagType* and is defined as follows:

$$TagType = PrimitiveType \cup ValueClass \cup \{\texttt{ref}, \&, \texttt{typedref}\}$$

Given a value, the function *tagType* yields the tag type associated to the value.

The *stack frames*, also known as *activation records*, are defined as elements of the universe *Frame*. Every frame is given as a 6-tuple comprising a *program counter*, *local variable addresses*, addresses allocated in the *local memory pool*, *argument addresses*, an *evaluation stack*, and *a method reference*.

$$\begin{aligned} &Frame = \\ &\quad Pc \times Map(Local, Adr) \times \mathcal{P}(Adr \times ValueClass) \times Map(Arg, Adr) \times List(Val) \times MRef \end{aligned}$$

Bytecode verification has to ensure that all objects are properly initialized before they are used. To make object initialization accessible to the type safety proof, the semantics model has to be designed in such a way that programs which do not properly initialize objects can be identified. In other words, the run-time representation of partially initialized objects must be distinguishable from that of (fully) initialized objects. For this purpose, we consider the universe *InitState*. The initialization status and the run-time type of object references are recorded

Function definition			Function name
tagType	:	$Map(Val, TagType)$	value's tag
opNo	:	$Map(Op, \mathbb{N})$	operator's arity
opResVal	:	$Map(Op \times List(Val), Val)$	result of an operation
opResType	:	$Map(Op \times List(Type), Type)$	result type of an operation
handleOf	:	$Map(Type, Val)$	handle associated to a type

Table 4.6: The basic static functions.

as elements of *InitState*. An object reference whose run-time type is an object class can be either *partially initialized*, *i.e.*, its initialization is in progress, or *initialized*. An object reference whose run-time is a value type can only be *initialized*.

InitState	=	$InProgress(ObjClass)$	initialization in progress
	|	$Init(ObjClass \cup ValueType)$	initialization done

Given an operator, the function *opNo* returns the number of operands necessary to perform the operation indicated by the operator. The resulting value of an operation and the resulting verification type can be determined through the functions *opResVal* and *opResType*, respectively. While *opResVal* is a "semantics" function, *opResType* is defined by [45, Partition III, §1.5].

The *RefAnyType* instruction applied with a type token argument T requires a *handle*, *i.e.*, an *opaque descriptor*, associated to T. For a type T, we assume that $handleOf(T)$ returns a handle, *i.e.*, an instance of the special value class `System.RuntimeTypeHandle` [7], corresponding to T.

The below defined universe *Switch* depicts the possible execution modes of the operational semantics model $CLR_{\mathcal{L}}$ defined in Section 4.1.3.

Switch	=	*Noswitch*	normal execution mode
	|	*Invoke*(*Bool*, *MRef*, *List*(*Val*))	method invocation mode
	|	*Result*(*List*(*Val*))	method returning mode

The model is either in the *normal execution mode*, *i.e.*, it executes an instruction, or in the *method invocation mode*, *i.e.*, it invokes a method with a given list of arguments, or in the *method returning mode*, *i.e.*, it returns from a method possibly with a return value.

Similarly as for the C$^\sharp$'s static semantics, we define two derived static functions $defVal : Map(RefType \cup ValueType, Val)$ and $lookUp : Map((ObjClass \cup ValueType) \times MRef, MRef)$.

Given a type T, $defVal(T)$ is the *default value*, also known as "zero", associated to the type T. Note that the default value of a value class is the result of initializing each instance field of the class to its default value.

Definition 4.1.3 (Default value) *For a type $T \in RefType \cup ValueType$, we define*

$$\mathit{defVal}(T) = \begin{cases} 0, & \text{if } T \in \mathit{PrimitiveType} \setminus \{\texttt{float64}\}; \\ 0.0d, & \text{if } T = \texttt{float64}; \\ \texttt{null}, & \text{if } T \in \mathit{RefType}; \\ \{T{::}F \mapsto \mathit{defVal}(\mathit{fieldType}(T{::}F)) \mid & \\ \qquad T{::}F \in \mathit{instFields}(T)\}, & \text{if } T \in \mathit{ValueClass}. \end{cases}$$

To resolve dynamic dispatched method calls, we consider the function *lookUp*, defined similarly as for C$^\sharp$. We use the same terminology as in Chapter 3: The method argument of a virtual call is *called*, whereas the method selected for the run-time execution is *invoked*. For non-virtual call cases, we use the two notions interchangeably.

Definition 4.1.4 (Method lookup) *For a type T and a method $T'{::}M$, we define*

```
$\mathit{lookUp}(T, T'{::}M) =$
  **if** $T$ declares a non-`abstract` method $M \wedge$
    $((T = T') \vee (T' \in \mathit{ObjClass} \Rightarrow T{::}M$ overrides $T'{::}M)) \wedge$
    $(T' \in \mathit{Interface} \Rightarrow T{::}M$ implements $T'{::}M)$
  **then** $T{::}M$
  **elseif** $T = \texttt{object}$ **then** $\mathit{undef}$
  **else** $\mathit{lookUp}(T'', T'{::}M)$ where $T''$ is the direct base class of $T$
```

The conditions **[override/implement]**, defined for C$^\sharp$ in Chapter 3, Section 3.5.1, should also be satisfied in the CLR:

[override/implement] If $T{::}M \in \mathit{MRef}$ overrides/implements $T'{::}M \in \mathit{MRef}$, then:

(at) $\mathit{argNo}(T{::}M) = \mathit{argNo}(T'{::}M)$ and for every $i = 1, \mathit{argNo}(T{::}M) - 1$,

$$\mathit{argTypes}(T{::}M)(i) = \mathit{argTypes}(T'{::}M)(i)$$

(rt) $\mathit{retType}(T{::}M) = \mathit{retType}(T'{::}M)$

The instructions dealing directly with pointers, *i.e.*, *LoadInd* and *StoreInd*, require the addresses referenced by the pointers to be valid. An address is a *valid address* if it not `null`, it is in the range of *Adr*, and it is "naturally aligned" for the target architecture. To distinguish between invalid addresses and valid addresses, we define the external function $\mathit{validAdr} : \mathit{Map}(\mathit{Adr}, \mathit{Bool})$ which, given an address, indicates the validity of the address. This function is external as its definition depends on the target architecture. However, the addresses of local variables, arguments, fields and value type instances inside boxed objects are regarded as *valid*.

Delegate classes As the two methods declared by a delegate class are not allowed to have a body, the semantics model should treat them specially. We proceed as follows. Every delegate

`.ctor` is implemented via ASM rules. Similarly as for C$^\sharp$, every delegate's `Invoke` method is provided with the body, resulted upon compiling the corresponding C$^\sharp$ `Invoke` method, that expresses the sequentiality of the execution of delegate invocation list elements. To implement that body, we insert two more methods, implemented via ASM rules, in every delegate class. The method `_length` is supposed to return the length of the delegate invocation list, while the method `_invoke` is used to invoke a certain method in the delegate invocation list. Thus, every delegate class is translated as shown in Figure 4.3 (for an `Invoke` method with a non-`void` return type).

So, basically, our model considers the delegates implemented, in particular created, through ASM rules because their CLR implementation is not transparent. This modelling decision has an impact on the members declared by `System.Delegate`, whose implementation relies on the delegates implementation. Therefore, the implementation of these members should be accomplished via ASM rules, too. A full specification of the CLR would consist of ASM rules for each member declared by `System.Delegate`. As such an approach would significantly blow up the size of the exposition, we consider that `System.Delegate` declares only three members, *i.e.*, the *static methods* `Combine`, `Remove`, and `op_Equality`.

4.1.3 The Dynamic Semantics

The *dynamic state* of the machine CLR$_\mathcal{L}$ is given by the *basic dynamic functions* of Table 4.7.

The function *mem* is applied to read and to store values *other* than value class instances from and into the memory, respectively. This function can return an undefined value. The stack of call frames *frameStack* is defined as a list of frames. A call frame comprises a *program counter pc*, *local variable addresses locAdr*, *local memory pool locPool*, *argument addresses argAdr*, *an evaluation stack evalStack*, and *a method reference meth*.

The local memory pool *locPool* is a (possibly empty) set that consists of pairs of the form (adr, T), where *adr* is the address of a memory block allocated for an instance of a value class T. In the lightweight CLR, an address *adr* is considered in *locPool* in a single case[5]: if the address is allocated for a value class instance created upon a constructor invocation (see the semantics of the *NewObj* instruction in Section 4.1.3.1). The memory allocated in the local memory pool is reclaimed upon method context termination. Modelling the local memory pool is an important contribution of our formal specification. The reason behind modelling this pool concerns the type safety proof, more precisely, the typing of pointers (see Remark 4.1.10 in Section 4.1.6).

The derived nullary function *frame* : *Frame* denotes *the currently executed frame*. It is determined as a 6-tuple consisting of *pc*, *locAdr*, *locPool*, *argAdr*, *evalStack*, and *meth*.

$$frame = (pc, locAdr, locPool, argAdr, evalStack, meth)$$

Accordingly, *pc* gives the program counter of the current *frame*, *locAdr* the local variable addresses of the current *frame*, etc. To simplify the technical presentation, we separate the current *frame* from the stack of call frames, *i.e.*, *frame* is *not* contained in *frameStack*.

The function *fieldAdr* is defined analogously as for C$^\sharp$. It assigns to every instance field of a reference or of a value class instance stored at a given address its allocated address. More

[5]Upon adding generics in Section 4.3, there will be another case.

Figure 4.3 The translation scheme of delegate classes.

```
.class private auto ansi sealed DC extends System.Delegate
{
    .method public hidebysig specialname rtspecialname
            instance void .ctor(object o, native int m) runtime managed
    {
    } // end of method DC::.ctor
    .method public hidebysig instance T Invoke(T_1 x_1,...,T_n x_n) runtime managed
    {
    } // end of method DC::Invoke
} // end of class DC
```

$\Downarrow$

```
.class private auto ansi sealed DC extends System.Delegate
{
    .method public hidebysig specialname rtspecialname
            instance void .ctor(object o, native int m) runtime managed
    {
    } // end of method DC::.ctor
    .method public hidebysig instance T Invoke(T_1 x_1, ..., T_n x_n) cil managed
    {
        .maxstack n+2
        .locals init (T V_0, int32 V_1)
        0:      br      n+9
        1:      ldarg   0           // load the delegate instance
        2:      ldloc   1           // load the current index in the invocation list
        3:      ldarg   1           // load the first argument of the delegate call
        ⋮
        n+2 :   ldarg   n           // load the n-th argument of the delegate call
        n+3 :   call    instance T DC::_invoke(int32, T_1, ..., T_n)
        n+4 :   stloc   0
        n+5 :   ldloc   1
        n+6 :   ldc.i4  1
        n+7 :   add
        n+8 :   stloc   1
        n+9 :   ldloc   1
        n+10 :  ldarg   0
        n+11 :  call    instance int32 DC::_length()
        n+12 :  clt
        n+13 :  brtrue  1
        n+14 :  ldloc   0
        n+15 :  ret
    } // end of method DC::Invoke

    .method private instance int32 _length() runtime managed
    {
    } // end of method DC::_length

    .method private instance T _invoke(int32 i,T_1 x_1,...,T_n x_n) runtime managed
    {
    } // end of method DC::_invoke
}
```

Function definition			Function name
mem	:	$Map(Adr, Val \cup \{undef\})$	memory function
frameStack	:	$List(Frame)$	stack of call frames
pc	:	Pc	current program counter
locAdr	:	$Map(Local, Adr)$	local variable addresses
locPool	:	$\mathcal{P}(Adr \times ValueClass)$	local memory pool
argAdr	:	$Map(Arg, Adr)$	argument addresses
evalStack	:	$List(Val)$	evaluation stack
meth	:	$MRef$	current method
fieldAdr	:	$Map((ObjRef \cup Adr) \times FRef, Adr)$	instance field addresses
initState	:	$Map(ObjRef, InitState)$	initialization status
addressOf	:	$Map(ObjRef, Adr)$	boxed value address
typedRefAdr	:	$Map(TypedRef, Adr)$	typed reference address
typedRefType	:	$Map(TypedRef, Type)$	typed reference type
methodOf	:	$Map(MethPtr, MRef)$	method pointer method
invocationList	:	$List(ObjRef \times MRef)$	invocation list of a delegate
switch	:	$Switch$	current execution mode

Table 4.7: The basic dynamic functions.

precisely, the address of an instance field $C::F$ of an object *ref* is given by $fieldAdr(ref, C::F)$. Also, the address of an instance field $C::F$ of a value class instance stored at an address *adr* is $fieldAdr(adr, C::F)$.

The function *initState* records *the run-time type* and *the initialization status* of the object references. Given an object reference *ref*, if $initState(ref)$ is $InProgress(T)$, where T is an object class, the meaning is that the run-time type of *ref* is T, and the initialization of *ref* is *in progress*. If $initState(ref)$ is $Init(T)$, the run-time type of *ref* is T, and *ref* is *init*ialized. The values of value types can be "boxed" in the heap and then addressed by object references. Such a reference is always considered initialized. Formally, if *ref* is a boxed value of a value type T, then it always holds $initState(ref) = Init(T)$.

$InitState$	=	$InProgress(ObjClass)$	initialization in progress
	|	$Init(ObjClass \cup ValueType)$	initialization done

To simplify the exposition of the semantics rules, we define the derived (selector) function $actualTypeOf : Map(ObjRef, Type)$, which assigns to every object reference its run-time type. Given a reference *ref*, $actualTypeOf(ref) = T$ if there exists $T \in ObjClass \cup ValueType$ such that $initState(ref) = InProgress(T)$ or $initState(ref) = Init(T)$.

A boxed value on the heap embeds the value type and a list of instance field addresses. When "unboxing" a "boxed" value, one needs its address on the heap. This address is maintained in *addressOf*, and it is set when the value is boxed (see the semantics of the *Box* instruction in Section 4.1.3.1).

The function *typedRefAdr* returns the address embedded in a typed reference, while the type transmitted with a typed reference is recorded by the function *typedRefType*. For a method pointer, *methodOf* returns the method reference pointed to by the pointer. Exactly as in the C$^\sharp$ model, the invocation list each delegate instance is equipped with upon its creation is maintained by *invocationList*. As in C$^\sharp$, each invocation list is a per delegate instance immutable. The function *methodOf* is only accessed for initializing the *invocationList* upon calling a delegate `.ctor`.

The *current execution mode* of the semantics model defined in Section 4.1.3.1 is maintained by *switch*.

Initial Constraints For the *initial state* of CLR$_\mathcal{L}$, the following conditions are satisfied by the basic dynamic functions:

$mem(adr)$	=	*undef*, for every $adr \in Adr$	$fieldAdr$	=	$\emptyset$
$frameStack$	=	$[\,]$	$initState$	=	$\emptyset$
pc	=	0	$addressOf$	=	$\emptyset$
$locAdr$	=	$\emptyset$	$typedRefAdr$	=	$\emptyset$
$locPool$	=	$\emptyset$	$typedRefType$	=	$\emptyset$
$argAdr$	=	$\emptyset$	$methodOf$	=	$\emptyset$
$evalStack$	=	$[\,]$	$invocationList$	=	$\emptyset$
$meth$	=	`.entrypoint`	$switch$	=	$Noswitch$

Similarly as in the C$^\sharp$ model, we consider $memVal : Map(Adr \times Type, Val \cup \{undef\})$ and WRITEMEM to ”read from” and to ”write in” memory, respectively. Apart from substituting *Struct* by *ValueClass*, the definitions are the same.

Definition 4.1.5 *For an address adr and a type T, we define*

```
$memVal(adr, T) =$
  if $T \in ValueClass$ then
    $\{T{::}F \mapsto memVal(fieldAdr(adr, T{::}F), fieldType(T{::}F)) \mid T{::}F \in instFields(T)\}$
  else $mem(adr)$
```

Definition 4.1.6 *For an address adr, a type T and a type val, we define*

```
$\text{WRITEMEM}(adr, T, val) \equiv$
  if $T \in ValueClass$ then
    forall $T{::}F \in instFields(T)$ do
      $\text{WRITEMEM}(fieldAdr(adr, T{::}F), fieldType(T{::}F), val(T{::}F))$
  else $mem(adr) := val$
```

Obviously, after storing a value *val* at an address of a memory block, sufficient to hold values of a type T, the value read from the memory block is *val*.

Lemma 4.1.1 *After* WRITEMEM(adr, T, val) *is executed,* $memVal(adr, T) = val$.

Proof. By induction on the structure of the possibly value class type T. □

To determine the local variable values and argument values of the current frame, we define the derived functions $locVal : Map(Local, Value)$ and $argVal : Map(Arg, Value)$, respectively. The value of a local variable is determined as the value stored in the memory block at the local variable address, sufficient to hold values of the local variable declared type. Similarly, the value of an argument is determined in terms of the argument address and argument declared type.

$$locVal(n) = memVal(locAdr(n), locTypes(n)), \text{ for every } n = 0, locNo(meth) - 1$$

$$argVal(n) = memVal(argAdr(n), argTypes(n)), \text{for every } n = 0, argNo(meth) - 1$$

4.1.3.1 The Semantics of the Bytecode Instructions

This section defines the effect of the instructions in Table 4.8 on the dynamic state of the virtual machine. The operational semantics model – given in terms of the ASM intrepeter named $\text{CLR}_{\mathcal{L}}$ – executes the macro *execScheme* which is parameterized by the submachines EXECCLR and SWITCHCLR. This macro gives control to these submachines depending on the current execution mode *switch* of the machine $\text{CLR}_{\mathcal{L}}$.

The macro *execScheme* is defined as follows. If *switch* is *Noswitch*, *i.e.*, the machine is in the normal execution mode, then EXECCLR defined in Figures 4.4, 4.5, and 4.6 takes control and executes the current instruction $code(pc)$. If *switch* is not *Noswitch*, *i.e.*, the machine either invokes a method or returns from a method, *execScheme* gives control to the machine SWITCHCLR defined in Figure 4.7.

$$\text{CLR}_{\mathcal{L}} \equiv execScheme(\text{EXECCLR}, \text{SWITCHCLR})$$

```
$execScheme(\text{EXECCLR}, \text{SWITCHCLR}) \equiv$
  **if** $switch = Noswitch$ **then** $\text{EXECCLR}(code(pc))$
  **else** $\text{SWITCHCLR}$
```

The *Dup* instruction duplicates the top element of the stack, while the *Pop* instruction removes the top element from the stack. The instruction *Const*(T,*lit*) pushes the constant (literal) *lit*, of type T, onto the *evalStack*.

The instruction $LoadLoc(n)$ loads on the *evalStack* the value $locVal(n)$ of the n-th local variable of the current frame. The address $locAdr(n)$ of the n-th local variable is pushed onto the stack by the instruction $LoadLocA(n)$. The instruction $StoreLoc(n)$ stores with WRITEMEM the topmost value of *evalStack* at the variable address $locAdr(n)$.

The *Execute* instruction is a “constructor” (abstraction) of several real CLR instructions. The instruction models, for example, the *generic* (read: with no specified data type) instructions `add`, `div`, `mul`, and `sub`, but also instructions performing conversions such as

Instruction	Informal description
Dup	Duplicates the top value of the stack.
Pop	Removes the top element of the stack.
Const(*T*,*lit*)	Loads the literal *lit* of type *T* onto the stack.
LoadLoc(n)	Loads the n-th local variable onto the stack.
StoreLoc(n)	Pops the top stack element, and stores it in the n-th local variable.
Execute(*op*)	Executes the operation indicated by the operator *op*.
Cond(*op*,*target*)	Executes *op* and transfers control to *target* if *op* returns true.
LoadArg(n)	Loads the n-th argument onto the stack.
StoreArg(n)	Pops the top stack element and stores it in the n-th argument.
Call(*tail*,*T*,*mref*)	Invokes the method *mref* of return type *T*; the boolean *tail* indicates whether the call is tail.
CallVirt(*tail*,*T*,*mref*)	Calls the late bound method *mref* of return type *T* associated, at run-time, with an object; *tail* indicates if the call is tail.
Return	Returns from the current method.
NewObj(*mref*)	Creates a new object, and invokes the instance constructor *mref* with the object.
LoadField(T,C::F)	Loads the field C::F, of declared type T, onto the stack.
StoreField(T,C::F)	Stores the top stack element into the field C::F, of declared type T.
LoadLocA(n)	Loads the address of the n-th local variable onto the stack.
LoadArgA(n)	Loads the address of the n-th argument onto the stack.
LoadFieldA(T,C::F)	Loads the address of the field C::F, of declared type T, onto the stack.
LoadInd(T)	Loads value of type T indirect onto the stack.
StoreInd(T)	Stores value of type T indirect from the stack.
Box(T)	Converts value of type T to object reference (boxed value).
Unbox(T)	Converts boxed value of value type T to a pointer to its unboxed form.
Unbox.Any(T)	Converts boxed value to its unboxed form.
MkRefAny(T)	Loads a typed reference pointing to type T onto the stack.
RefAnyType	Loads the handle to the type embedded in a typed reference.
RefAnyVal(T)	Loads the address out of a typed reference onto the stack.
CastClass(T)	Attempts to cast the top stack element to object class or interface T.
IsInstance(T)	Tests if the top stack element is an instance of the object class or interface T.
LoadFtn(*mref*)	Loads the (method) pointer associated to the method *mref*.
LoadVirtFtn(*mref*)	Loads the (method) pointer associated to the virtual method *mref*.

Table 4.8: The considered CIL instructions.

`conv.i4`, `conv.i8`, and `conv.r8`. The execution of the operation underlying a generic instruction requires the operands' types. This is one reason why the values are carrying their types. Given an operator *op*, the instruction *Execute*(*op*) pops a number *opNo*(*op*) of operands from the *evalStack*. If the operation does not throw an exception, the result of the operation is pushed onto the stack . The resulting value is obtained with *opResVal*. The below defined predicate *exceptionCase* captures the cases when an exception is thrown. These are the following:

- division by zero for operators of integral types;
- operations that perform an overflow check and whose results cannot be represented in the result type;
- values that are not "normal" numbers are checked for finiteness or `div`ision/`rem`ainder operations are executed for a minimal value of an integral type and -1.

$$\begin{aligned}
\mathit{exceptionCase}(op, vals) \quad &:\Leftrightarrow \quad \mathit{divByZeroCase}(op, vals) \vee \mathit{overflowCase}(op, vals) \\
&\qquad \vee\ \mathit{invNrCase}(op, vals)
\end{aligned}$$

$$\begin{aligned}
\mathit{divByZeroCase}(op, vals) \quad &:\Leftrightarrow \quad op \in \{\texttt{div}, \texttt{rem}\} \wedge vals(1) = 0 \\
&\qquad \wedge\ \mathit{tagType}(vals(0)) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\} \\
&\qquad \wedge\ \mathit{tagType}(vals(1)) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{overflowCase}(op, vals) \quad :\Leftrightarrow \quad op \in \{\texttt{add.ovf}, \texttt{conv.ovf}, \texttt{mul.ovf}, \texttt{sub.ovf}\} \\
&\quad \wedge\ \mathit{opResVal}(op, vals) < \mathit{MIN}(\mathit{opResType}(op, [\mathit{tagType}(vals(0)), \mathit{tagType}(vals(1))])) \\
&\quad \wedge\ \mathit{opResVal}(op, vals) > \mathit{MAX}(\mathit{opResType}(op, [\mathit{tagType}(vals(0)), \mathit{tagType}(vals(1))]))
\end{aligned}$$

$$\begin{aligned}
\mathit{invNrCase}(op, vals) \quad &:\Leftrightarrow \quad (vals(0) \in \{\texttt{NaN}, \texttt{+infinity}, \texttt{-infinity}\} \\
&\qquad \wedge\ op = \texttt{ckfinite}) \vee (op \in \{\texttt{div}, \texttt{rem}\} \\
&\qquad \wedge\ vals(0) = \mathit{MIN}(\mathit{tagType}(vals(0))) \wedge\ vals(1) = -1 \\
&\qquad \wedge\ \mathit{tagType}(vals(0)) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\} \\
&\qquad \wedge\ \mathit{tagType}(vals(1)) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\})
\end{aligned}$$

Example 4.1.2 *In the following bytecode fragment, the instruction Execute(div) (the abstract instruction corresponding to the real CIL instruction* `div`*) computes the result of dividing the first local variable by the second local variable.*

```
0:  ldloc  0
1:  ldloc  1
2:  div
```

The "type" of the division depends on the operands' types: For example, the 32*-bit integer division by zero throws an exception, whereas the* 64*-bit floating division by zero returns the special value* `NaN` *(read: "not a number"). Therefore, every value has assigned a tag type. In the above example, the types carried by the values of the first two local variables determine the exact "type" of division.*

Similarly as *Execute*, the *Cond* instruction is a constructor of several real CIL instructions, *e.g.*, conditional branch instructions such as `beq`, `bge`, and `blt`. Given an operator *op* and a code index *target*, $\mathit{Cond}(op, target)$ pops a number $\mathit{opNo}(op)$ of operands from the *evalStack*. The *pc* is set to *target* if *opResVal* yields true for the operator *op* and the popped operands, and otherwise, the *pc* is incremented by 1.

The value $\mathit{argVal}(n)$ of the n-th argument of the current *frame* is pushed onto the stack by $\mathit{LoadArg}(n)$. The instruction $\mathit{LoadArgA}(n)$ loads the address $\mathit{argAdr}(n)$. The instruction $\mathit{StoreArg}(n)$ writes at $\mathit{argAdr}(n)$ with WRITEMEM the topmost value of the *evalStack*.

The *Call* instruction can be used to call one of the three methods declared by the class `System.Delegate`, or one of the two methods added to every delegate class definition, or a regular method.

Figure 4.4 The operational semantics rules.

```
$\textsc{execCLR}(instr) \equiv$ match $instr$
  $Dup$                    $\rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
                               $evalStack := evalStack' \cdot [val, val]$
                               $pc := pc + 1$
  $Pop$                    $\rightarrow$ let $evalStack' = pop(evalStack)$ in
                               $evalStack := evalStack'$
                               $pc := pc + 1$
  $Const(\_, lit)$         $\rightarrow$ $evalStack := evalStack \cdot [lit]$
                             $pc := pc + 1$
  $LoadLoc(n)$             $\rightarrow$ $evalStack := evalStack \cdot [locVal(n)]$
                             $pc := pc + 1$
  $StoreLoc(n)$            $\rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
                               $\textsc{WriteMem}(locAdr(n), locTypes(n), val)$
                               $evalStack := evalStack'$
                               $pc := pc + 1$
  $Execute(op)$            $\rightarrow$ let $(evalStack', vals) = split(evalStack, opNo(op))$ in
                               if $\neg exceptionCase(op, vals)$ then
                                 let $val = opResVal(op, vals)$ in
                                   $evalStack := evalStack' \cdot [val]$
                                   $pc := pc + 1$
  $Cond(op, target)$       $\rightarrow$ let $(evalStack', vals) = split(evalStack, opNo(op))$ in
                               $evalStack := evalStack'$
                               $pc :=$ if $opResVal(op, vals)$ then $target$ else $pc + 1$
  $LoadArg(n)$             $\rightarrow$ $evalStack := evalStack \cdot [argVal(n)]$
                             $pc := pc + 1$
  $StoreArg(n)$            $\rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
                               $\textsc{WriteMem}(argAdr(n), argTypes(n), val)$
                               $evalStack := evalStack'$
                               $pc := pc + 1$
  $Call(tail, \_, T{::}M)$ $\rightarrow$ if $T =$ System.Delegate then
                               let $[ref, ref'] = take(evalStack, argNo(T{::}M))$ in
                                 if $M =$ Combine then $\textsc{DelegateCombine}(T, ref, ref')$
                                 if $M =$ Remove then $\textsc{DelegateRemove}(T, ref, ref')$
                                 if $M =$ op_Equality then $\textsc{DelegateEqual}(ref, ref')$
                             else let $(evalStack', [ref] \cdot vals) = split(evalStack, argNo(T{::}M))$ in
                               if $ref \neq$ null then
                                 if $T \in DelegateClass$ then
                                   if $M =$ _length then
                                     $evalStack := evalStack' \cdot [length(invocationList(ref))]$
                                   if $M =$ _invoke then
                                     $\textsc{DelegateCall}(ref, vals)$
                                 else
                                   $evalStack := evalStack'$
                                   $switch\ \ \ := Invoke(tail, T{::}M, [ref] \cdot vals)$
  $CallVirt(tail, \_, T{::}M)$ $\rightarrow$ let $(evalStack', vals) = split(evalStack, argNo(T{::}M))$ in
                               $evalStack := evalStack'$
                               $\textsc{VirtCall}(tail, T{::}M, vals)$
```

Figure 4.5 The operational semantics rules (continued).

```
EXECCLR($instr$) $\equiv$ match $instr$
  ⋮
  $Return$              $\rightarrow$ if $retType(meth) = \texttt{void}$ then $switch := Result([\,])$
                          else $switch := Result(top(evalStack))$
  $NewObj(C::\texttt{.ctor})$  $\rightarrow$ let $vals = take(evalStack, argNo(C::\texttt{.ctor}) - 1)$ in
                            if $C \in DelegateClass$ then DELEGATECREATE($C, vals$)
                            elseif $C \in ObjClass$ then OBJECTCREATE($C::\texttt{.ctor}, vals$)
                            else VALUECLASSCREATE($C::\texttt{.ctor}, vals$)
  $LoadField(T, C::F)$   $\rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
                            if $val \neq \texttt{null}$ then
                              $evalStack := evalStack' \cdot [memVal(fieldAdr(val, C::F), T)]$
                              $pc := pc + 1$
  $StoreField(T, C::F)$  $\rightarrow$ let $(evalStack', [val, val']) = split(evalStack, 2)$ in
                            if $val \neq \texttt{null}$ then
                              WRITEMEM($fieldAdr(val, C::F), T, val'$)
                              $evalStack := evalStack'$
                              $pc := pc + 1$
  $LoadLocA(n)$          $\rightarrow$ $evalStack := evalStack \cdot [locAdr(n)]$
                          $pc := pc + 1$
  $LoadArgA(n)$          $\rightarrow$ $evalStack := evalStack \cdot [argAdr(n)]$
                          $pc := pc + 1$
  $LoadFieldA(\_, C::F)$ $\rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
                            if $val \neq \texttt{null}$ then
                              $evalStack := evalStack' \cdot [fieldAdr(val, C::F)]$
                              $pc := pc + 1$
  $LoadInd(T)$           $\rightarrow$ let $(evalStack', [adr]) = split(evalStack, 1)$ in
                            if $validAdr(adr)$ then
                              $evalStack := evalStack' \cdot [memVal(adr, T)]$
                              $pc := pc + 1$
  $StoreInd(T)$          $\rightarrow$ let $(evalStack', [adr, val]) = split(evalStack, 2)$ in
                            if $validAdr(adr)$ then
                              WRITEMEM($adr, T, val$)
                              $evalStack := evalStack'$
                              $pc := pc + 1$
  $Box(T)$               $\rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
                            let $ref = NewBox(val, T)$ in
                              $evalStack := evalStack' \cdot [ref]$
                              $pc := pc + 1$
  $Unbox(T)$             $\rightarrow$ let $(evalStack', [ref]) = split(evalStack, 1)$ in
                            if $ref \neq \texttt{null} \wedge T = actualTypeOf(ref)$ then
                              $evalStack := evalStack' \cdot [addressOf(ref)]$
                              $pc := pc + 1$
  $Unbox.Any(T)$         $\rightarrow$ let $(evalStack', [ref]) = split(evalStack, 1)$ in
                            if $T \in ValueType$ then
                              if $ref \neq \texttt{null} \wedge actualTypeOf(ref) = T$ then
                                $evalStack := evalStack' \cdot [memVal(addressOf(ref), T)]$
                                $pc := pc + 1$
                            elseif $ref = \texttt{null} \vee actualTypeOf(ref) \preceq T$ then
                              $pc := pc + 1$
```

System.Delegate's method calls If the *Call* instruction's method token argument is System.Delegate::Combine, two delegate references are popped from the *evalStack*, and their invocation lists are concatenated through the macro DELEGATECOMBINE (defined similarly as in the C$^\sharp$ model):

```
DELEGATECOMBINE($DC, ref, ref'$) $\equiv$
  if $ref \neq$ null then
    $evalStack := drop(evalStack, 2) \cdot [ref']$
  elseif $ref' \neq$ null then
    $evalStack := drop(evalStack, 2) \cdot [ref]$
  else let $ref'' = new(ObjRef, DC)$ in
    $initState(ref'')$          $:= Init(DC)$
    $invocationList(ref'')$ $:= invocationList(ref) \cdot invocationList(ref')$
    $evalStack$                 $:= drop(evalStack, 2) \cdot [ref'']$
```

If the called method is System.Delegate::Remove, two delegates are popped from the *evalStack*, and the macro DELEGATEREMOVE (defined similarly as in the C$^\sharp$ model) is applied to remove the last occurrence of the invocation list of the second delegate from the invocation list of the first delegate.

```
DELEGATEREMOVE($DC, ref, ref'$) $\equiv$
  if $ref =$ null then
    $evalStack := drop(evalStack, 2) \cdot$ [null]
  elseif $ref' =$ null then
    $evalStack := drop(evalStack, 2) \cdot [ref]$
  else let $L = invocationList(ref)$ and $L' = invocationList(ref')$ in
    if $L = L'$ then
      $evalStack := drop(evalStack, 2) \cdot$ [null]
    elseif $sublist(L', L)$ then
      let $ref'' = new(ObjRef, DC)$ in
        $initState(ref'')$          $:= Init(DC)$
        $invocationList(ref'')$ $:= prefix(L', L) \cdot suffix(L', L)$
        $evalStack := drop(evalStack, 2) \cdot [ref'']$
    else $evalStack := drop(evalStack, 2) \cdot [ref]$
```

Finally, if System.Delegate::op_Equality is called, the macro DELEGATEEQUAL (defined similarly as in the C$^\sharp$ model) determines whether the two delegates on top of the *evalStack* have identical invocation lists.

```
DELEGATEEQUAL($ref, ref'$) $\equiv$
  if $ref =$ null $\vee$ $ref' =$ null then
    $evalStack := drop(evalStack, 2) \cdot [ref = ref']$
  else let $L = invocationList(ref)$ and $L' = invocationList(ref')$ in
    let $val = (length(L) = length(L')) \wedge \forall i = 0, length(L) - 1 \,:\, L(i) = L'(i)$ in
      $evalStack := drop(evalStack, 2) \cdot [val]$
```

Delegate class method calls If the method token argument of a *Call* instruction is the method `_length` of a delegate class, a delegate instance is popped from the *evalStack*, and the length of its invocation list is loaded. If the called method is the `_invoke` method of a delegate class, then a delegate reference, an index in the invocation list of the delegate and a list of arguments are popped from the *evalStack*. The macro DELEGATECALL (defined similarly as in the C$^\sharp$ model) calls the method in the invocation list pointed to by the index with the given arguments.

```
DELEGATECALL$(ref, [i] \cdot vals) \equiv$
  let $(ref', T{::}M) = invocationList(ref)(i)$ in
    let this $=$ if $T \in ValueClass$ then $addressOf(ref')$ else $ref'$ in
      $switch := Invoke(False, T{::}M, [$this$] \cdot vals)$
```

As a result of the delegate classes translation in Figure 4.3, the call of a delegate class `Invoke` method is regarded as a regular call.

Regular method calls The instruction $Call(tail, T', T{::}M)$ takes a number $argNo(T{::}M)$ of arguments from the *evalStack*, and if the target reference (denoting the instance whose method is invoked) is not `null`, the method $T{::}M$, of return type T', is called with the arguments popped from the *evalStack* by setting *switch* to the appropriate method invocation mode. Note that although [3] performs a `null` check for the target reference, [45] omits to mention it.

In contrast with the *Call* instruction, the *CallVirt* instruction's method token argument is late bound, *i.e.*, the method to be invoked is looked up dynamically by means of the *lookUp* function. Furthermore, if the *CallVirt* instruction finds a boxed instance of a value type on the stack and is applied to a virtual method inherited from `System.ValueType` or from an interface implemented by the value type, and the value type overrides or implements that method, then the boxed instance is "unboxed" and passed as the `this` pointer to the implementing method, *i.e.*, the method determined through the *lookUp* function. More exactly, the address of the boxed instance is passed as the `this` pointer. Another difference between the *Call* and *CallVirt* instructions is that the latter cannot have a `.ctor` as method token argument. However, this detail is omitted by [45].

```
VIRTCALL$(tail, T{::}M, [ref] \cdot vals) \equiv$
  if $ref \neq$ null then
    let $T'{::}M = lookUp(actualTypeOf\,(ref), T{::}M)$ in
      if $actualTypeOf\,(ref) = T' \in ValueType$ then
        $switch := Invoke(tail, T'{::}M, [addressOf\,(ref)] \cdot vals)$
      else $switch := Invoke(tail, T'{::}M, [ref] \cdot vals)$
```

The *Return* instruction takes from the *evalStack* no or one value, depending on the return type of the current method. It then returns this value (if any) by setting *switch* to the appropriate method returning mode.

The *NewObj* instruction can be used for three purposes: to construct an object (other than a delegate), to construct a delegate, or a to create a value class instance.

Object creation If OC is an object class (other than a delegate class), the instruction $NewObj(OC::$`.ctor`$)$ first allocates[6] an object on the heap for a fresh object reference ref. It sets $initState(ref)$ to $InProgress(OC)$ and pushes the reference ref onto the $evalStack$. The reference gets fully initialized, *i.e.*, $initState(ref)$ becomes $Init(OC)$, upon invoking the instance constructor `object::.ctor` (see the object initialization rules in Section 4.1.4.3). It then invokes the constructor $OC::$`.ctor` with ref and the arguments on the $evalStack$ by setting $switch$ to the appropriate method invocation mode. Finally, the reference ref is pushed onto the stack.

```
$\text{OBJECTCREATE}(OC::\texttt{.ctor}, vals) \equiv$
  $\textbf{let } ref = new(ObjRef, OC) \textbf{ in}$
    $evalStack := drop(evalStack, argNo(OC::\texttt{.ctor}) - 1) \cdot [ref]$
    $initState(ref) := InProgress(OC)$
    $\textbf{forall } OC'::F \in instFields(OC) \textbf{ do}$
      $\text{WRITEMEM}(fieldAdr(ref, OC'::F), fieldType(OC'::F), defVal(fieldType(OC'::F)))$
    $switch := Invoke(False, OC::\texttt{.ctor}, [ref] \cdot vals)$
```

Delegate creation If the `.ctor` reference token argument of the $NewObj$ instruction is declared by a delegate class, an object and a method pointer are popped from the $evalStack$. A new delegate is then created and loaded onto the $evalStack$. Its invocation list is initialized with the object on the stack and the method reference referred to by the method pointer.

```
$\text{DELEGATECREATE}(DC, [ref, m]) \equiv$
  $\textbf{let } ref' = new(ObjRef, DC) \textbf{ in}$
    $initState(ref')$       $:= Init(DC)$
    $invocationList(ref')$ $:= [(ref, methodOf(m))]$
    $evalStack$            $:= drop(evalStack, 2) \cdot [ref']$
```

Value class instance creation If VC is a value class, the instruction $NewObj(VC::$`.ctor`$)$ first allocates a block of memory where the new value class instance is stored. The address of the memory block is considered in the local memory pool $locPool$ of the current $frame$. The constructor is then invoked with the address of the allocated memory block and the necessary arguments (assumed to be on the $evalStack$). Finally, the value class instance is pushed onto the stack.

```
$\text{VALUECLASSCREATE}(VC::\texttt{.ctor}, vals) \equiv$
  $\textbf{let } adr = new(Adr, VC) \textbf{ in}$
    $locPool := locPool \cup \{(adr, VC)\}$
    $evalStack := drop(evalStack, argNo(VC::\texttt{.ctor}) - 1) \cdot [memVal(adr, VC)]$
    $\textbf{forall } VC::F \in instFields(VC) \textbf{ do}$
      $\text{WRITEMEM}(fieldAdr(adr, VC::F), fieldType(VC::F), defVal(fieldType(VC::F)))$
    $switch := Invoke(False, VC::\texttt{.ctor}, [adr] \cdot vals)$
```

[6]The macros $new(ObjRef, C)$, $new(ObjRef)$ and $new(Adr, T)$, defined in Chapter 3 Table 3.19 to allocate object references and memory blocks, are also applied throughout the current chapter with a trivial change in the definition: *ValueClass* instead of *Struct*.

The instructions $\mathit{LoadField}(T, C{::}F)$ and $\mathit{StoreField}(T, C{::}F)$ loads and updates the value of the field $C{::}F$ of declared type T, respectively. The address of the field $C{::}F$, of declared type T is addressable through the instruction $\mathit{LoadFieldA}(T, C{::}F)$. Both instructions $\mathit{LoadField}(T, C{::}F)$ and $\mathit{LoadFieldA}(T, C{::}F)$ pop the topmost value of the *evalStack*, which is an object reference *ref* (if C is an object class) or a pointer *adr* to a value class instance (if C is a value class). The *LoadField* instruction computes the field value with *memVal*: It is the value of type $\mathit{fieldType}(C{::}F)$ stored at $\mathit{fieldAdr}(\mathit{ref}, C{::}F)$ (if C is an object class) or $\mathit{fieldAdr}(\mathit{adr}, C{::}F)$ (if C is a value class). *LoadFieldA* loads on the *evalStack* the address $\mathit{fieldAdr}(\mathit{ref}, C{::}F)$ (if C is an object class) or $\mathit{fieldAdr}(\mathit{adr}, C{::}F)$ (if C is a value class). The instruction $\mathit{StoreField}(T, C{::}F)$ takes from the *evalStack* the two topmost values: The first is a reference *ref* (if C is an object class) or a pointer *adr* to a value class instance (if C is a value class), and the second is a value *val*. The instruction stores *val* at the address given by $\mathit{fieldAdr}(\mathit{ref}, C{::}F)$ (if C is an object class) or $\mathit{fieldAdr}(\mathit{adr}, C{::}F)$ (if C is a value class).

The instruction $\mathit{LoadInd}(T)$ pops the topmost value of the *evalStack*. This is supposed to be a pointer pointing to an address, say *adr*. If *adr* is a valid address, then the value of type T stored at *adr* is computed. Finally, the value is loaded on the *evalStack*. The instruction $\mathit{StoreInd}(T)$ takes the two topmost values of the *evalStack* which are supposed to be a pointer pointing to an address, say *adr*, and a value, say *val*. If kadr is a valid address, then *val* is stored at the address *adr*.

According to [45], to load an object reference, the *LoadInd* instruction should be used in the form *LoadInd*(`object`). Similarly, to store an object reference, the *StoreInd* instruction should be used in the form *StoreInd*(`object`).

The *Box* instruction turns a value type instance into a heap-allocated object "by copying", while *Unbox* performs the inverse coercion. Applied to a value type, *Box* copies the data from the value type instance into a newly allocated object, operation accomplished by means of the macro *NewBox*, defined exactly as for C$^\sharp$. The *Box* instruction does nothing if it is applied to a reference type[7].

The macro $\mathit{NewBox}(\mathit{val}, T)$ picks up an unallocated address *adr* from the heap and writes the top stack value *val* (assumed to be of type T) at the memory block at *adr* sufficient to hold values of type T. The run-time type, used, for example, by the instructions *CastClass* and *IsInstance*, is set to T.

```
let $ref = NewBox(val, T)$ in $P \equiv$
  let $ref = new(ObjRef)$ and $adr = new(Adr, T)$ in
    $\textsc{WriteMem}(adr, T, val)$
    $addressOf(ref) := adr$
    $initState(ref) \quad := Init(T)$
  seq $P$
```

The *Unbox* instruction takes from the *evalStack* an object reference to a boxed value and extracts the value type instance out it, assuming that the value type coincides with the instruction's type token argument. However, the value pushed onto the stack is a pointer representing the address (given by *addressOf*) of the value type instance embedded into the boxed value.

[7]In .NET Framework (v1.1) and (v1.0), the *Box* instruction was only applicable to value types. Upon introducing the generics in (v2.0), the applicability of *Box* has been extended to reference types.

Figure 4.6 The operational semantics rules (continued).

```
EXECCLR($instr$) $\equiv$ **match** $instr$
  ⋮
  $MkRefAny(T)$        $\rightarrow$ **let** $(evalStack', [adr]) = split(evalStack, 1)$ **in**
                           **let** $tr = new(TypedRef)$ **in**
                             $typedRefAdr(tr) := adr$
                             $typedRefType(tr) := T$
                             $evalStack := evalStack' \cdot [tr]$
                             $pc := pc + 1$
  $RefAnyType$         $\rightarrow$ **let** $(evalStack', [tr]) = split(evalStack, 1)$ **in**
                           **let** $T = typedRefType(tr)$ **in**
                             $evalStack := evalStack' \cdot [handleOf(T)]$
                             $pc := pc + 1$
  $RefAnyVal(T)$       $\rightarrow$ **let** $(evalStack', [tr]) = split(evalStack, 1)$ **in**
                           **if** $T = typedRefType(tr)$ **then**
                             $evalStack := evalStack' \cdot [typedRefAdr(tr)]$
                             $pc := pc + 1$
  $CastClass(T)$       $\rightarrow$ **let** $ref = top(evalStack)$ **in**
                           **if** $ref = \texttt{null} \lor actualTypeOf(ref) \preceq T$ **then**
                             $pc := pc + 1$
  $IsInstance(T)$      $\rightarrow$ **let** $(evalStack', [ref]) = split(evalStack, 1)$ **in**
                           $pc := pc + 1$
                           **if** $ref \neq \texttt{null} \land actualTypeOf(ref) \not\preceq T$ **then**
                             $evalStack := evalStack' \cdot [\texttt{null}]$
  $LoadFtn(T::M)$      $\rightarrow$ **let** $mp = new(MethPtr, T::M)$ **in**
                           $evalStack := evalStack \cdot [mp]$
                           $pc := pc + 1$
  $LoadVirtFtn(T::M)$  $\rightarrow$ **let** $(evalStack', [ref]) = split(evalStack, 1)$ **in**
                           **if** $ref \neq \texttt{null}$ **then**
                             **let** $T' = actualTypeOf(ref)$ **in**
                               **let** $mp = new(MethPtr, lookUp(T', T::M))$ **in**
                                 $evalStack := evalStack' \cdot [mp]$
                                 $pc := pc + 1$
```

For value types, the *Unbox.Any* instruction, unlike *Unbox*, leaves the value, not the address of the value, on the *evalStack*. Moreover, while the *Unbox*'s type token argument can only represent value types, the *Unbox.Any* instruction can also be applied to reference types, in which case has the same effect as the *CastClass* instruction below, namely it casts the object reference on top of the stack to the reference type.

A typed reference is created and loaded onto the stack using the *MkRefAny* instruction. The new typed reference, imported from the universe *TypedRef*, embeds the pointer which is on top of the *evalStack* and the instruction's type token argument.

```
**let** $tr = new(TypedRef)$ **in** $P \equiv$
  **import** $tr$ **do**
    $TypedRef(tr) := True$
  **seq** $P$
```

The *RefAnyType* instruction takes a typed reference from the *evalStack* and loads not the ”type token” embedded in the typed reference (as *wrongly* specified by [45, Partition III, §4.21]), but a handle, *i.e.*, an instance of the value class `System.RuntimeTypeHandle` corresponding to the type in the typed reference. The handle is obtained by means of the function *handleOf*. The *RefAnyVal* instruction retrieves the address embedded in the typed reference on top of the *evalStack*, assuming that the instruction's type token argument is the same as the type stored in the typed reference.

The instruction $CastClass(T)$ tests whether the topmost value on the *evalStack* is of type T. If not, an exception is thrown. The instruction $IsInstance(T)$ pops a reference to an (possibly boxed) object from the *evalStack*. If the actual type of the object is a subtype of T, then it is cast to T, and the result is pushed on the stack, exactly as $CastClass(T)$ had been called. If the object is not an instance of T, `null` is pushed on the *evalStack*.

The *LoadFtn* instruction pushes onto the *evalStack* a pointer to the instruction's method as a `native int` value. The method pointer, imported from the universe *MethPtr*, is mapped by *methodOf* to the instruction's method token. A method pointer for a virtual method can be obtained with the *LoadVirtFtn* instruction. This instruction pops an object reference from the *evalStack*. It then loads on the *evalStack* a pointer to the virtual method determined based on the run-time type of the object reference and the given method. It also sets the function *methodOf* to map the method pointer into the determined method reference.

```
**let** $mp = new(MethPtr, T::M)$ **in** $P \equiv$
  **import** $mp$ **do**
    $MethPtr(mp) \quad := True$
    $methodOf(mp) := T::M$
  **seq** $P$
```

We now define the submachine SWITCHCLR which is responsible for method transfers, *i.e.*, for invoking methods and returning from methods. This submachine takes control whenever *switch* is not *Noswitch*.

Method invocations If $switch = Invoke(tail, mref, vals)$, the current *frame* is pushed onto the *frameStack*, unless the boolean *tail* indicates a tail call, in which case the current *frame* is discarded. The new current *frame* is set up trough the below defined macro SETFRAME.

The macro $\text{SETFRAME}(T::M, vals)$ sets the frame for invoking the method $T::M$ with the arguments *vals*: *pc* is set to 0, *locPool* to $\emptyset$, *evalStack* to $[]$, *meth* to $T::M$. Moreover, by means of the below defined macro ALLOCARGLOC, the arguments *vals* are stored at the newly allocated argument addresses *argAdr*, and ”zeros” of the appropriate types are stored at the newly allocated local variable addresses *locAdr*. If the invoked method is a `.ctor` declared by `object`, the target reference becomes fully initialized. This is according to the object initialization rules: An object gets fully initialized when a `.ctor` of the inheritance tree's root is invoked.

```
$\text{SETFRAME}(T{::}M, vals) \equiv$
    $pc \quad := 0$
    $evalStack := [\,]$
    $locPool \quad := \emptyset$
    $meth \quad := T{::}M$
    let $zeros = [\,defVal(locTypes(T{::}M)(n)) \mid n = 0\,,\, locNo(T{::}M) - 1\,]$ in
        $\text{ALLOCARGLOC}(argTypes(T{::}M), locTypes(T{::}M), vals \cdot zeros)$
    if $T =$ object $\wedge\, M =$ .ctor then
        let $ref = vals(0)$ in
            let $InProgress(T') = initState(ref)$ in
                $initState(ref) := Init(T')$
```

The macro ALLOCARGLOC is defined as follows. For the current method, two lists of types L, L' and a list of values *vals*, ALLOCARGLOC$(L, L', vals)$ allocates addresses[8] for arguments of the types in L and for local variables of the types in L' and then writes the values *vals* at the allocated addresses. The definition of ALLOCARGLOC is *well-founded*, since the sum of the lengths of L and L' is equal with *vals*'s length.

```
$\text{ALLOCARGLOC}(L, L', vals) \equiv$
    forall $i \in 0, length(L) - 1$ do
        let $adr_i = new(Adr, L(i))$ in
            $argAdr(i) := adr_i$
            $\text{WRITEMEM}(adr_i, L(i), vals(i))$
    forall $j \in 0, length(L') - 1$ do
        let $adr_j = new(Adr, L'(j))$ in
            $locAdr(i) := adr_j$
            $\text{WRITEMEM}(adr_j, L'(j), vals(j + length(L)))$
```

Remark 4.1.1 [45] *contradicts itself. On one hand, it states that the local variables whose declared types are reference types (so, in particular, also those whose declared types are pointer types) are initialized to* `null` *upon the method's entry* [45, Partition II, §15.4.1.3]. *On the other hand, it claims that pointers cannot be* `null` [45, Partition III, §1.1.4.2]. *The truth is that the pointers can be assigned the value* `null`, *but only in a single case: when the local variables of pointer types are automatically initialized upon the method's entry.*

Remark 4.1.2 *It is worth noticing an important difference between the* C$^\sharp$ *and CLR models. The* C$^\sharp$ *model does not allocate addresses for the* `ref`/`out` *arguments, whereas the CLR model allocates addresses for pointer type arguments (the equivalent of the* C$^\sharp$ *by-reference arguments). The modelling for CLR is justified as long as the CLR, unlike* C$^\sharp$, *can directly handle pointers, in particular, pointers to pointers.*

Method return If $switch = Return(vals)$, then, by means of POPFRAME, the current *frame* is discarded, and the new current *frame* is set to the topmost frame on the *frameStack* with the possibly empty list of return values *vals* pushed onto the *evalStack*.

[8] All simultaneous allocations, *i.e.*, calls of the macro *new*, are supposed to provide pairwise different fresh elements from *Adr*; see [62] and for a justification of this assumption.

Figure 4.7 The SWITCHCLR submachine.

```
SWITCHCLR ≡ match $switch$
  $Invoke(tail, T{::}M, vals)$ → if $\neg tail$ then $frameStack := push(frameStack, frame)$
                                $\textsc{SetFrame}(T{::}M, vals)$
                                $switch := Noswitch$

  $Result(vals)$              → $\textsc{PopFrame}(1, vals)$
                                $switch := Noswitch$
```

The definition of the macro POPFRAME is as follows. POPFRAME$(k, vals)$ discards the current *frame* by returning to its caller frame the possibly empty list of values *vals* and by incrementing the *pc* by k.

```
$\textsc{PopFrame}(k, vals) \equiv$
  let $(frameStack', [(pc', locAdr', locPool', argAdr', evalStack', meth')]) =$
    $split(frameStack, 1)$ in
      $pc$         $:= pc' + k$
      $locAdr$     $:= locAdr'$
      $locPool$    $:= locPool'$
      $argAdr$     $:= argAdr'$
      $evalStack$  $:= evalStack' \cdot vals$
      $meth$       $:= meth'$
      $frameStack$ $:= frameStack'$
```

4.1.4 The Bytecode Verification

A global view of the CLR bytecode verification is given in Section 4.1.4.1. Section 4.1.4.2 concentrates on the verification type system that we consider. It also states what conditions should be satisfied by the bytecode structure to be verifiable. The type-consistency checks performed by the bytecode verification are specified in Section 4.1.4.3. The function used by the bytecode verification to simulate the execution paths through the bytecode is defined in Section 4.1.4.4.

4.1.4.1 An Overview of the Bytecode Verification

The *bytecode verification* is performed on a per-method basis. The verification attempts to associate a valid *stack state* with every instruction. The stack state specifies the number of values on the *evalStack* at that point in the code and for each slot of the *evalStack*, a required type that should be present in that slot. Also, in order to decide whether an object reference is fully initialized, the verification tracks in a `.ctor` a special type for the `this` pointer (*zeroth argument*). Given a method, we refer to the type of the zeroth argument[9] and to the stack state as a *type state* of the method.

[9]The type of the zeroth argument is relevant only when verifying a `.ctor`.

Universe of verification types		
VerificationType	$=$	$\mathit{Type} \cup \mathit{BoxedType} \cup \{\mathit{UnInit}, \texttt{typedref}, \mathit{Null}\}$
BoxedType	$=$	$\{\mathit{boxed}(T) \mid T \in \mathit{ValueType}\}$

Table 4.9: CIL's verification types.

Bytecode verification should simulate all possible control flow paths through the bytecode and ensure that a valid type state exists for every reachable instruction. By a *valid* type state, we mean a type state that satisfies certain type-consistency checks (see Section 4.1.4.3). The verification uses only type assignments and does not take advantage of any values during the simulation. The verification terminates successfully if all the control paths have been simulated. It finishes unsuccessfully when the verification cannot compute a valid type state for a particular instruction.

At the beginning of the simulation, the stack state is empty and the zeroth argument type is set appropriately (see Definition 4.1.9 in Section 4.1.5). For each instruction, the verification checks that the stack before the instruction execution contains enough slots and these slots are types compatible with the expected types for the instruction. It then simulates the effect of the instruction on the *evalStack* and zeroth argument (in case of verifying a `.ctor`). Any type mismatch on instruction arguments, evaluation stack overflow or underflow, or any violation of other verification conditions causes the verification to fail. It then propagates the inferred type state to all possible successors of the instruction. If an instruction is the target of several branches, the verification has to "merge" the type states along these branches. The "merge" for two zeroth argument types succeeds if the types are the same. The "merge" for two stack states is executed slot-by-slot: The *merged stack state* contains for each entry the supremum, *i.e.*, "least upper bound", of the corresponding types in the stack states. The simulation algorithm fails if it cannot compute a merged type state. This happens if the two stack states have different lengths[10] or one slot of a stack state has a type non-compatible with the type in the corresponding slot of the other stack state.

While the verification is described above as both computing type states and checking them, we will assume that the information stored in the type states has already been computed prior to the type-consistency checks.

4.1.4.2 The Verification Type System and Bytecode Structure

The verification types are depicted by *VerificationType* in Table 4.9. A *verification type* is a CIL type as described in Table 4.1, a boxed type described by *BoxedType*, the type *UnInit*, the type `typedref`, the special type *Null* (used as the type of the `null` reference).

For every value type T, there exists a *reference type boxed*(T), called *boxed type*. The value of a type *boxed*(T) is a location where a value of type T can be stored. Only the bytecode verification has knowledge of the boxed types. In the bytecode, they can only be referred to as `object`, `System.ValueType`, or as interfaces implemented by the underlying value

[10]The situation when two stack states have different lengths corresponds to a program point where the run-time stack can have different heights depending on the path by which the point is reached; such bytecode cannot be proved correct in the framework described in this section, and must be rejected according to [45, Partition III, §1.8.1.1].

type. The type *UnInit* is a special type used by the verification (only) to track the type of the zeroth argument of a `.ctor`.

We define the *compatibility relation* $\sqsubseteq$ for verification types.

Definition 4.1.7 (Compatible verification types) *The relation $\sqsubseteq$, for verification types, is the least reflexive and transitive relation such that*

- *if T is Null and $T' \in ObjClass \cup Interface \cup BoxedType$, or*
- *if $T \in ObjClass \cup Interface \cup BoxedType$ and T' is* `object`*, or*
- *if $T \in BoxedType$ and T' is* `System.ValueType`*, or*
- *if $T \in ObjClass$ and T' is a base class of T or an interface implemented by T, or*
- *if $T \in Interface$ and T' is an interface implemented by T, or*
- *if T is boxed(T''), $T'' \in ValueType$, and T' is an interface implemented by T'',*

then $T \sqsubseteq T'$.

The following lemma, required for the type safety proof, claims that a pointer type is only compatible with itself.

Lemma 4.1.2 *If $T\& \sqsubseteq T'$ or $T' \sqsubseteq T\&$, then $T' = T\&$.*

Proof. Follows immediately from Definition 4.1.7. □

There are a number of restrictions imposed by [45] in using pointer types in verifiable code. Since the typed references embed pointers, there are also restrictions in using the type `typedref`. The restrictions are *sufficient* conditions for the type safety result.

Address Of	The declared type of a local variable and argument used with the instruction *LoadLocA* and *LoadArgA*, respectively, should not be a pointer type or `typedref`.
Field Type	The declared type of a field should not be a pointer type or `typedref`.
Return Type	The return type of a method should not be a pointer type or `typedref`.
Tail Call Argument Types	The argument types of a method called tail should not be pointer types or `typedref`.
Box Type	The *Box* instruction should not be applied to `typedref`.
MkRefAny Type	The *MkRefAny* instruction should not be applied to `typedref`.

Basically, all these constraints aim to prevent the existence of *dangling pointers*, *i.e.*, pointers which outlive their targets. Consider, for example, ***Tail Call Argument Types***. As the `tail` prefix shortens the lifetime of the caller frame, passing a pointer to the caller frame as an argument produces a dangling pointer. This would happen since the caller frame is discarded just before the tail call.

Remark 4.1.3 *Upon entering a method, the macro* ALLOCARGLOC *allocates memory blocks for all local variables and arguments independent of their declared type, in particular, also for those whose declared type is a pointer type. We took this modelling decision to simplify the exposition. The restriction* ***Address Of*** *helps us, in particular, to guarantee that the decision is "safe" in the sense that the address of a pointer type local variable or argument cannot be addressed.*

Remark 4.1.4 *As pointed out in* [66], ***Field Type*** *can be relaxed to allow fields of value classes to hold pointers, and yet the type safety result holds. However, this has a downside: the possibility that every value class may be boxed is lost. If a value class has at least one pointer type field, then one should not be allowed to box this value class. Otherwise, dangling pointers could be created, e.g., a pointer in a frame could be stored on the heap in the pointer type field of the boxed value class.*

Also ***Return Type*** *could be relaxed to allow methods to return a special kind of pointers. It is about the pointers that point to a permanent "home" that still exists after the method returns, e.g., locations on the heap. When being tracked by the verification, these pointers should, however, be marked distinctly.*

There are also constraints on the bytecode structure. [45] states that the following conditions should hold or the bytecode verification will fail. To provide the reader with a complete view of the bytecode verification, we list here all the conditions required by the verification, regardless of whether they are relevant in the type safety proof or aimed to simplify the JIT compilation.

"This" of `.ctor`	There should be no instruction *StoreArg*(0) in the body of a `.ctor`.
Tail Call Return Type	If the method *mref*$'$ calls tail the method *mref*, then either both *mref* and *mref*$'$ have the return type `void` or *retType*$(mref) \sqsubseteq$ *retType*$(mref')$.
Tail Call Pattern	The instruction following a tail call should be *Return*.

The first condition guarantees that the `this` pointer of a `.ctor` is always pointing to the same location in the heap. This matters, for example, when checking whether the `.ctor` is called *on the same* `this` pointer (and not on another object reference) of a `.ctor` of the same class or the base class (see the object initialization rules in Section 4.1.4.3).

The restriction ***Tail Call Return Type*** insures that the frame of the method called tail returns a value of the type expected by the caller frame of the discarded frame (the frame which contained the tail call).

The reader might ask what is the reason for requiring ***Tail Call Pattern***. Recall that the `tail` prefix is an optional directive, which might be ignored when the bytecode is jitted. If the prefix is ignored, the caller frame would continue to execute after the callee's frame returns. However, because of ***Tail Call Pattern***, the caller frame would immediately (read: in the next step of the operational semantics model) terminate if the callee's frame returned normally. Thus, the program will execute properly even if the caller's frame was not discarded before performing the tail call.

While ***"This" of*** `.ctor` and ***Tail Call Return Type*** are relevant in the type safety proof, ***Tail Call Pattern*** serves to simplify the JIT compilation and is included here for completeness only.

In order to guarantee the type-safe execution of delegates, the verification imposes for a delegate creation the below specified constraint ***Delegate Pattern***. For this, we need first to define the compatibility relation for delegate classes and methods. A delegate class is compatible with a target method if the arguments that can be passed to the delegate's `Invoke` method can also be passed to the target method and every value that can be returned by the target method can also be returned by (read: is of the return type of) the delegate's `Invoke` method.

Definition 4.1.8 *A delegate class DC is* compatible *with a method mref if the following conditions are satisfied:*

- $argTypes(DC::\texttt{Invoke})(i) \sqsubseteq argTypes(mref)(i)$, *for every* $i = 1, argNo(mref) - 1$
- $retType(mref) \sqsubseteq retType(DC::\texttt{Invoke})$ *or* $retType(DC::\texttt{Invoke}) = retType(mref) = \texttt{void}$.

Delegate Pattern	A delegate creation, *i.e.*, an instruction $NewObj(DC::\texttt{.ctor})$ with $DC \in DelegateClass$, should occur immediately after an instruction $LoadFtn(mref)$ or after an instruction $LoadVirtFtn(mref)$ immediately preceded by the *Dup* instruction. Moreover, *DC* should be *compatible* with *mref*. Furthermore, no branch target should be within these instruction sequences (other than at the beginning of the sequences).

As the .NET Framework (v2.0) [3]'s bytecode verification does not track method pointer types, requiring ***Delegate Pattern*** makes the verification's job possible. It assures that the *NewObj* instruction used to construct a delegate always finds a method pointer on the stack. The rationale for the *Dup* instruction is provided in [45, Partition III, §1.8.1.5.1]: It guarantees that the same object is used as the target object (embedded in the delegate) and to determine the target virtual method (embedded in the delegate).

Example 4.1.3 *To understand why the target object should be the same as the object used to determine the target method, we consider the following bytecode fragment which does not obey* ***Delegate Pattern****. We assume that the object class* OC' *is a subclass of the object class OC,* OC' *contains an implementation of the method OC::M, and DC is a delegate class.*

```
1:  newobj     instance void OC::.ctor()
2:  newobj     instance void OC'::.ctor()
3:  ldvirtftn  instance void OC::M(int32)
4:  newobj     instance void DC::.ctor(object, native int)
```

Provided that OC' implements $OC::M$, the LoadVirtFtn instruction loads a method pointer to $OC'::M$ on the stack. Consequently, the last NewObj instruction creates a delegate with a target object of run-time type OC and the target method $OC'::M$. When calling this delegate, $OC'::M$ gets invoked on an object of type OC. Therefore, in the body of $OC'::M$, the `this` *pointer is of run-time type OC, but it is used as being of type OC'. However, this could lead to memory violations if, for example, $OC'::M$ accesses an instance field declared by OC'.*

4.1.4.3 Verifying the Bytecode Instructions

Before simulating the execution of an instruction, the bytecode verification checks whether certain conditions are satisfied. In this section, we formally specify these conditions by means of the predicate *check* defined in Figure 4.8. The checks for a single instruction operate on the stack state $evalStackT : List(VerificationType)$ and, in case of verifying a `.ctor`[11], on the type of the `this` pointer $argZeroT : VerificationType$. Some of the specified checks do not matter for the type safety proof. They are required to simplify the JIT compilation. However, for the sake of completeness, we specify all the conditions imposed by the bytecode verification.

Several instructions, *e.g.*, *Dup*, *Const*, *LoadLoc*, push a value onto the stack, but do not pop off any values. In their case, one has to avoid an overflow of the evaluation stack. More exactly, the number of values on the evaluation stack should not exceed an upper bound computed by every .NET compliant language's compiler. For this purpose, we consider the function *maxEvalStack* (see Table 4.10) which assigns to every method an upper bound for the number of elements on the stack.

A few instructions, *e.g.*, *Dup*, *LoadField*, *StoreField*, require a minimum number of values on the stack. For them, there is an underflow check.

$$\boxed{overflow(evalStackT, n) :\Leftrightarrow length(evalStackT) + n > maxEvalStack(meth)}$$

$$\boxed{underflow(evalStackT, n) :\Leftrightarrow length(evalStackT) < n}$$

The instructions *Dup* and *Pop* require the stack to have at least one value. Additionally, in case of *Dup*, the duplicated value should not overflow the stack.

Since the local variables in the verifiable CLR bytecode are zeroed upon the method's entry, the bytecode verification does not need to check for a *LoadLoc* instruction that the corresponding variable has been previously assigned. The *StoreLoc* instruction requires that the topmost value on the stack is of the corresponding variable's declared type.

The *Execute* instruction expects on the stack values of the types for which the operation indicated by the given operator is verifiable. These types are maintained in the function *verifOpTypes* in Table 4.10, which given an operator, returns the set of lists of types for

[11]For technical reasons, we consider the *argZeroT* component for every method.

Function definition	Function name
$maxEvalStack \; : \; Map(MRef, \mathbb{N})$	the maximum number of elements on the stack
$verifOpTypes \; : \; Map(Op, \mathcal{P}(List(Type)))$	the operands' types for which an operation is verifiable

Table 4.10: The verification specific functions.

which the corresponding operation is verifiable. The definition of *verifOpTypes* can be found in [45, Partition III, §1.5].

Example 4.1.4 *Consider the addition operator* `add`. *According to* [45, Partition III, §1.5], *this operator is verifiable for several pairs of types. For example, add is verifiable for operands of type* `int32`, *but also for operands of type* `int64`. *The complete set of type pairs for which add is verifiable is defined through the function verifOpTypes:*

$$verifOpTypes(add) = \{ [\texttt{int32}, \texttt{int32}], [\texttt{int64}, \texttt{int64}], [\texttt{int32}, \texttt{native int}], [\texttt{native int}, \texttt{int32}], [\texttt{native int}, \texttt{native int}], [\texttt{float64}, \texttt{float64}] \}$$

The result type for each possible combination of operand types is defined through the function opResType defined in Section 4.1.2. For example, $opResType(add, [\texttt{int32}, \texttt{int32}]) = \texttt{int32}$ *and* $opResType(add, [\texttt{int32}, \texttt{native int}]) = \texttt{native int}$.

We will call the elements of the set *UncondBranchInstr* = {*Return*} *unconditional branch instructions*[12]. The evaluation stack before executing an instruction *instr* immediately following an unconditional branch instruction should be empty, unless *instr* is the target of a forward branch instruction[13]. Therefore, the verification fails if a later branch instruction $Cond(op, target)$ – whose *target* points to *instr* – were to have on the evaluation stack more values than needed for executing *op*.

When executing *op*, the instruction $Cond(op, target)$ expects on the stack operands of types given by $verifOpTypes(op)$.

```
$backBranch(pos, target, evalStackT, \mathcal{L}) :\Leftrightarrow$
  **if** $target < pos \wedge code(target - 1) \in UncondBranchInstr \wedge$
    $\neg \exists pos \in Pc \; : \; (0 \leq pos \wedge pos < target \wedge code(pos) = Cond(\_, target))$ **then**
    $evalStackT \in_{len} \mathcal{L}$
  **else** $evalStackT \in_{suf} \mathcal{L}$
```

Example 4.1.5 *Consider the following bytecode fragment:*

[12]The set *UncondBranchInstr* consists now of a single element, but it will be extended with exceptions specific instructions in Section 4.2.4.2.

[13]This condition is required to ensure that *it is possible* to infer the state of the evaluation stack at the beginning of each instruction even through a *single forward-pass* analysis. The constraint guarantees, in particular, that the bytecode can be processed by a simple CIL-to-native-code compiler; see [48] for details.

Figure 4.8 Verifying the bytecode instructions.

```
$check(meth, pos, argZeroT, evalStackT) :\Leftrightarrow$
  $\textbf{match } code(pos)$
    $Dup$                        $\rightarrow \neg underflow(evalStackT, 1) \wedge \neg overflow(evalStackT, 1)$
    $Pop$                        $\rightarrow \neg underflow(evalStackT, 1)$
    $Const(\_, \_)$              $\rightarrow \neg overflow(evalStackT, 1)$
    $LoadLoc(\_)$                $\rightarrow \neg overflow(evalStackT, 1)$
    $StoreLoc(n)$                $\rightarrow evalStackT \sqsubseteq_{suf} [locTypes(n)]$
    $Execute(op)$                $\rightarrow evalStackT \in_{suf} verifOpTypes(op)$
    $Cond(op, target)$           $\rightarrow backBranch(pos, target, evalStackT, verifOpTypes(op))$
    $LoadArg(\_)$                $\rightarrow \neg overflow(evalStackT, 1)$
    $StoreArg(n)$                $\rightarrow evalStackT \sqsubseteq_{suf} [argTypes(n)]$
    $Call(tail, \_, T{::}M)$     $\rightarrow (\textbf{if } tail \textbf{ then } length(evalStackT) = argNo(T{::}M)$
                                   $\textbf{else } \neg underflow(evalStackT, argNo(T{::}M))) \wedge$
                                   $\textbf{let } [T'] \cdot types = take(evalStackT, argNo(T{::}M)) \textbf{ in}$
                                     $types \sqsubseteq_{suf} paramTypes(T{::}M) \wedge$
                                     $\textbf{if } T \in ObjClass \wedge M = \texttt{.ctor} \textbf{ then } initCtorCompat(meth, T', T)$
                                     $\textbf{else } argZeroCompat(T', T)$
    $CallVirt(tail, \_, T{::}M)$ $\rightarrow \textbf{if } tail \textbf{ then } evalStackT \sqsubseteq_{len} argTypes(T{::}M)$
                                   $\textbf{else } evalStackT \sqsubseteq_{suf} argTypes(T{::}M)$
    $Return$                     $\rightarrow evalStackT \sqsubseteq_{len} void(retType(meth)) \wedge$
                                   $\textbf{if } methNm(meth) = \texttt{.ctor} \wedge classNm(meth) \neq \texttt{object} \textbf{ then}$
                                     $argZeroT \neq UnInit$
    $NewObj(C{::}\texttt{.ctor})$ $\rightarrow evalStackT \sqsubseteq_{suf} paramTypes(C{::}\texttt{.ctor}) \wedge$
                                   $\neg overflow(evalStackT, 2 - argNo(C{::}\texttt{.ctor})) \wedge$
                                   $\textbf{if } C \in DelegateClass \textbf{ then}$
                                     $\textbf{let } [T, \_] = take(evalStackT, 2) \textbf{ in } delegateTargetObj(pos, T)$
    $LoadField(\_, C{::}F)$      $\rightarrow \neg underflow(evalStackT, 1) \wedge$
                                   $\textbf{let } T = top(evalStackT) \textbf{ in } initFldCompat(T, classNm(meth), C{::}F)$
    $StoreField(T, C{::}F)$      $\rightarrow \neg underflow(evalStackT, 2) \wedge$
                                   $\textbf{let } [T'', T'] = take(evalStackT, 2) \textbf{ in}$
                                     $initFldCompat(T'', classNm(meth), C{::}F) \wedge T' \sqsubseteq T$
    $LoadLocA(\_)$               $\rightarrow \neg overflow(evalStackT, 1)$
    $LoadArgA(\_)$               $\rightarrow \neg overflow(evalStackT, 1)$
    $LoadFieldA(\_, C{::}F)$     $\rightarrow \neg underflow(evalStackT, 1) \wedge$
                                   $\textbf{let } T = top(evalStackT) \textbf{ in } initFldCompat(T, classNm(meth), C{::}F)$
    $LoadInd(T)$                 $\rightarrow \neg underflow(evalStackT, 1) \wedge$
                                   $\textbf{let } T' = top(evalStackT) \textbf{ in}$
                                     $\exists T'' \in ReferentType \; : \; (T' = T''\& \wedge T'' \sqsubseteq T)$
    $StoreInd(T)$                $\rightarrow \neg underflow(evalStackT, 2) \wedge$
                                   $\textbf{let } [T'', T'] = take(evalStackT, 2) \textbf{ in}$
                                     $\exists T''' \in ReferentType \; : \; (T'' = T'''\& \wedge T' \sqsubseteq T''' \sqsubseteq T)$
    $Box(T)$                     $\rightarrow evalStackT \sqsubseteq_{suf} [T]$
    $Unbox(\_)$                  $\rightarrow evalStackT \sqsubseteq_{suf} [\texttt{object}]$
    $Unbox.Any(\_)$              $\rightarrow evalStackT \sqsubseteq_{suf} [\texttt{object}]$
    $MkRefAny(T)$                $\rightarrow evalStackT \sqsubseteq_{suf} [T\&]$
    $RefAnyType$                 $\rightarrow evalStackT \sqsubseteq_{suf} [\texttt{typedref}]$
    $RefAnyVal(\_)$              $\rightarrow evalStackT \sqsubseteq_{suf} [\texttt{typedref}]$
    $CastClass(\_)$              $\rightarrow evalStackT \sqsubseteq_{suf} [\texttt{object}]$
    $IsInstance(\_)$             $\rightarrow evalStackT \sqsubseteq_{suf} [\texttt{object}]$
    $LoadFtn(\_)$                $\rightarrow True$
    $LoadVirtFtn(T{::}M)$        $\rightarrow evalStackT \sqsubseteq_{suf} [T]$
```

```
0: ret
1: ldloc  1
2: ldloc  2
3: beq    1
```

The instruction LoadLoc(1) *is immediately following the unconditional branch instruction Return, and it is not the target of a forward branch instruction. Hence, the stack state before the instruction LoadLoc*(1) *cannot be determined from the already derived information. Therefore, the bytecode verification considers an empty stack state before LoadLoc*(1)*. To ensure that a single forward-pass suffices for the bytecode simulation, one has to guarantee that the conditional instruction Cond*$(eq, 1)$ *(corresponding to the real instruction* `beq` *1) has a stack state with only the two values, required by the equality operator eq.*

Bytecode verification ensures for the *LoadArg* instruction that the loaded argument's value does not overflow the stack. The *StoreArg* instruction needs the top stack value to be of the corresponding argument's type.

A *Call* instruction requires that the argument types, except the type of the `this` pointer, are compatible with types given in the signature of the invoked method. A tail call requires the stack to be empty except for the arguments necessary to perform the call. Concerning the check for the type of the `this` pointer, we do a case distinction. If the invoked method $T::M$ is not a `.ctor`, the type of the `this` pointer should be compatible with T if T is an object class or an interface, or with $T\&$ if T is a value class.

$$\begin{aligned}&\mathit{argZeroCompat}(T', T) :\Leftrightarrow\\&\quad (T \in \mathit{ObjClass} \cup \mathit{Interface} \wedge T' \sqsubseteq T) \vee\\&\quad (T \in \mathit{ValueClass} \wedge T' \sqsubseteq T\&)\end{aligned}$$

For the `this` pointer of an object class `.ctor`, there are two special verification rules called *object initialization rules*:

- A `.ctor` should not return unless the `this` pointer is initialized.
- The `this` pointer is initialized after calling a `.ctor` of the same class or a base `.ctor`.

Remark 4.1.5 *The object initialization rules are not necessary for type safety of the bytecode language. The fields initialization with default values is sufficient for that purpose. The rules are important because significantly large parts of CLR's security assume consistent object states. Such a state relies, in particular, on constructors being invoked before objects are used and on base class constructors being invoked before initialization begins.*

We now describe the specification of the object initialization rules. For a `.ctor`, the bytecode verification also tracks the type of the zeroth argument. This is initially set to the special verification type *UnInit*. An important contribution of our framework consists in adding this type to the verification type system though this is *not* mentioned in [45]. The only allowed operations on an uninitialized `this` pointer are loading/storing of and into the corresponding instance fields.

If the *Call* instruction invokes a `.ctor` of an object class, the type of the `this` pointer needs to be either *UnInit* or a subtype of the `.ctor`'s class, and the `.ctor` is either of the same class as the caller's method (which is also assumed to be a `.ctor`) or of its base class.

$$\begin{aligned}&\mathit{initCtorCompat}(T{::}M, T', T'') :\Leftrightarrow\\ &\quad (T' \sqsubseteq T'' \lor T' = \mathit{UnInit}) \land\\ &\quad (T'' = T \lor T'' \text{ is the base class of } T)\end{aligned}$$

We illustrate the object initialization rules in Example 4.1.6 and Example 4.1.7.

Example 4.1.6 *We consider the following body of a method declared by an object class OC. We assume that OC declares the fields OC::F and OC::F′, both of type* `int32`*, and the instance method OC::M.*

```
0: ldarg   0
1: dup
2: ldfld   int32 OC::F
3: stfld   int32 OC::F′
4: ldarg   0
5: dup
5: ldc.i4  2
6: call    instance void OC::.ctor(int32)
7: call    instance void OC::M()
8: ret
```

If this method is a not a `.ctor`*, then the instruction Call(*`void`*, OC::*`.ctor`*(*`int32`*)) on line 6 would not be allowed since this instruction expects the type of the* `this` *pointer to be UnInit. However, this type can only occur in a* `.ctor`*. Consequently, a* `.ctor` *can be invoked explicitly, i.e., with a Call instruction,* only *from another* `.ctor`*.*

Let us now assume that this method is a `.ctor`*. Initially, the type of the zeroth argument argZeroT is UnInit. This type is loaded on the evalStackT by the instruction LoadArg(0) at code index 0. Although the* `this` *pointer is uninitialized, the loading of the field OC::F and then the storing into the field OC::F′ are verifiable. Also, the constructor call on an uninitialized* `this` *pointer at the code index 6 is verifiable. After this call, the* `this` *pointer is considered initialized (see Section 4.1.4.4 for the setting of argZeroT), and therefore, the method return in line 8 is verifiable.*

See Example 4.1.7 for a justification why the call at code index 7 is verifiable.

Note that the object initialization rules are not crucial for type safety. This would be anyway ensured since all fields of the `this` pointer are zeroed before the `.ctor` is run. The initialization rules are important to guarantee certain invariants (established by the execution of the `.ctor`) between the instance fields of the same class objects.

For the *CallVirt* instruction, one has to guarantee that the types of the arguments on the stack are compatible with the types expected by the method call. In case of a tail call, the evaluation stack should contain no other values than the arguments of the call.

The *Return* instruction expects the stack to be empty except for the value being returned (if any). To shorten the specification of the underlying check, we use the following notation:

$$void(T) = \begin{cases} [\,], & \text{if } T \text{ is } \texttt{void}; \\ [T], & \text{otherwise.} \end{cases}$$

The *Return* of a `.ctor` ensures that the type of the `this` pointer, tracked by means of *argZeroT*, is not *UnInit*.

Within a `.ctor`, the instructions *LoadField*, *StoreField* and *LoadFieldA* can also be used with an uninitialized `this` pointer.

$$\begin{aligned} & initFldCompat(T, T', C{::}F) :\Leftrightarrow \\ & \quad (C \in ObjClass \ \ \Rightarrow (T \sqsubseteq C \vee T = UnInit \wedge T' \sqsubseteq C)) \wedge \\ & \quad (C \in ValueClass \Rightarrow T \sqsubseteq C\&) \end{aligned}$$

Remark 4.1.6 *In earlier versions of .NET Framework, the object initialization rules for value classes ensured that each value class instance needed to be initialized by calling the* `.ctor` *which was supposed to initialize every instance field. Bytecode verification of .NET Framework (v2.0)* [3] *does not check this anymore.*

Since the *NewObj* instruction invokes a `.ctor` after creating a fresh reference, one has to ensure that the types of the arguments on the stack are compatible with the `.ctor`'s parameter types. The pushed object reference should not overflow the stack from which the `.ctor`'s parameters have been dropped.

We now consider the case when the *NewObj* instruction creates a delegate, that is *NewObj* is considered with a delegate class's `.ctor`. In this case, besides a `native int` value, the instruction also expects an object reference of a type compatible with the class of the method whose pointer has been loaded by the preceding instruction. Remember that the constraint ***Delegate Pattern*** given in Section 4.1.4.2 guarantees that the preceding instruction is either a *LoadFtn* or *LoadVirtFtn*. As the value class methods require a "home" for the `this` pointer, the value class instances should be boxed before being encapsulated into the delegate.

```
$delegateTargetObj(pos, T) :\Leftrightarrow$
  if $classNm(mref) \in ValueClass$ then
    $T \sqsubseteq boxed(classNm(mref))$
  else $T \sqsubseteq classNm(mref)$
  where $mref$ is such that $code(pos-1) = LoadFtn(mref)$ or
                            $code(pos-1) = LoadVirtFtn(mref)$
```

The delegate's `.ctor` invoked through a *NewObj* instruction also expects on the stack a value of type `native int`. This is used to represent the method pointer which is going to be embedded in the delegate. Bytecode verification of .NET Framework (v2.0) [3] does not track method pointer types. Knowing this, the reader might ask how it is ensured that the `native int` value represents indeed a method pointer and not an arbitrary `native int` value. The answer is to be found in the constraint ***Delegate Pattern*** (defined in Section 4.1.4.2) which guarantees that a method pointer is on top of the stack.

The *LoadInd* instruction ensures that the top stack value is of a pointer type whose referent type is compatible with the instruction's type token argument. The *StoreInd* instruction expects on the stack a pointer and a value to be stored at the address referenced by the pointer. The value should be of a type compatible with the referent type of the pointer type. Moreover, the referent type has to be compatible with the instruction's type token argument.

The *Box* instruction expects on the stack an instance of the instruction's value type token. An object reference is required on the stack by the *Unbox* and *Unbox.Any* instructions.

Remark 4.1.7 [45, Partition III, §4.30] *says that for the Unbox instruction, the bytecode verification checks whether the object reference on the stack represents a boxed instance of the instruction's value type token argument. However, in .NET Framework (v2.0)* [3], *this is not checked by the bytecode verification, but it is the subject of a run-time check.*

The *MkRefAny* instruction expects a pointer to the instruction's type token argument, while the instructions *RefAnyType* and *RefAnyVal* require a typed reference on the stack. The instructions *CastClass* and *IsInstance* require an object reference on the stack. No check is required for *LoadFtn*. The *LoadVirtFtn* instruction expects on the stack an object of a type compatible with the class of the method given in the instruction.

4.1.4.4 Computing Successor Type States

The type state for an instruction at index *pos* is constrained by referring to the type states of all instructions that are control-flow successors of *pos*. In Figure 4.9, we define the function *succ* which, given an instruction and a type state, computes the type states and the code indices of the instruction's successors.

For the instructions *LoadLoc* and *LoadLocA*, the function *succ* uses the variable's declared type maintained in *locTypes*. For the *Execute* instruction, the result type of the corresponding operation is determined by means of *opResType*, based on the given operator and on the operands' types.

The *Cond* instruction has two successors. One successor is given by the target instruction to which control is transferred if the operator given in the instruction returns true. The other successor is given by the instruction following the *Cond* instruction. The stack state of both successors is given by the stack state of the *Cond* instruction from which the operands of *Cond*'s operator have been dropped.

If the instructions *LoadArg* and *LoadArgA* are applied to the zeroth argument of a `.ctor`, the *succ* function uses the type *argZeroT* tracked for the `this` pointer. Otherwise, *succ* uses the argument's declared type recorded in *argTypes*.

If the *Call* instruction is applied to a `.ctor`, the type tracked for the zeroth argument (assumed to be *UnInit*) becomes the type denoted by the class of the current method. Also, all occurrences of *UnInit* in *evalStackT* are replaced by the fully initialized type, *i.e.*, the class of the current method.

Example 4.1.7 *Consider again the method body given in Example 4.1.6. Before the constructor call at code index* 6*, argZeroT is UnInit, and the stack state evalStackT is* $[UnInit, UnInit]$*. After the constructor call,*

Figure 4.9 The type state successors.

```
$succ(meth, pos, argZeroT, evalStackT) =$
  match $code(pos)$
    $Dup$                  $\rightarrow \{(pos+1, argZeroT, evalStackT \cdot top(evalStackT))\}$
    $Pop$                  $\rightarrow \{(pos+1, argZeroT, pop(evalStackT))\}$
    $Const(T, \_)$         $\rightarrow \{(pos+1, argZeroT, evalStackT \cdot [T])\}$
    $LoadLoc(n)$           $\rightarrow \{(pos+1, argZeroT, evalStackT \cdot [locTypes(n)])\}$
    $StoreLoc(\_)$         $\rightarrow \{(pos+1, argZeroT, pop(evalStackT))\}$
    $Execute(op)$          $\rightarrow$ let $(evalStackT', types) = split(evalStackT, opNo(op))$ in
                             $\{(pos+1, argZeroT, evalStackT' \cdot [opResType(op, types)])\}$
    $Cond(op, target)$     $\rightarrow$ let $evalStackT' = drop(evalStackT, opNo(op))$ in
                             $\{(target, argZeroT, evalStackT'), (pos+1, argZeroT, evalStackT')\}$
    $LoadArg(n)$           $\rightarrow$ if $n = 0 \wedge methNm(meth) = \texttt{.ctor}$ then
                             $\{(pos+1, argZeroT, evalStackT \cdot [argZeroT])\}$
                           else $\{(pos+1, argZeroT, evalStackT \cdot [argTypes(n)])\}$
    $StoreArg(\_)$         $\rightarrow \{(pos+1, argZeroT, pop(evalStackT))\}$
    $Call(tail, T', T::M)$ $\rightarrow$ if $tail$ then $\emptyset$
                           else let $evalStackT' = drop(evalStackT, argNo(T::M)) \cdot void(T')$ in
                             if $M = \texttt{.ctor}$ then
                               let $evalStackT'' = evalStackT'\,[classNm(meth)/UnInit]$ in
                                 $\{(pos+1, classNm(meth), evalStackT'')\}$
                             else $\{(pos+1, argZeroT, evalStackT')\}$
    $CallVirt(tail, T', T::M)$ $\rightarrow$ if $tail$ then $\emptyset$
                           else let $evalStackT' = drop(evalStackT, argNo(T::M)) \cdot void(T')$ in
                             $\{(pos+1, argZeroT, evalStackT')\}$
    $Return$               $\rightarrow \emptyset$
    $NewObj(C::\texttt{.ctor})$  $\rightarrow \{(pos+1, argZeroT, drop(evalStackT, argNo(C::\texttt{.ctor})-1) \cdot [C])\}$
    $LoadField(T, \_)$     $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T])\}$
    $StoreField(\_, \_)$   $\rightarrow \{(pos+1, argZeroT, drop(evalStackT, 2))\}$
    $LoadLocA(n)$          $\rightarrow \{(pos+1, argZeroT, evalStackT \cdot [locTypes(n)\&])\}$
    $LoadArgA(n)$          $\rightarrow$ if $n = 0 \wedge methNm(meth) = \texttt{.ctor}$ then
                             $\{(pos+1, argZeroT, evalStackT \cdot [argZeroT\&])\}$
                           else $\{(pos+1, argZeroT, evalStackT \cdot [argTypes(n)\&])\}$
    $LoadFieldA(T, \_)$    $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T\&])\}$
    $LoadInd(T)$           $\rightarrow$ if $T = \texttt{object}$ then
                             let $T'\& = top(evalStackT)$ in
                               $\{(pos+1, argZeroT, pop(evalStackT) \cdot [T'])\}$
                           else $\{(pos+1, argZeroT, pop(evalStackT) \cdot [T])\}$
    $StoreInd(\_)$         $\rightarrow \{(pos+1, argZeroT, drop(evalStackT, 2))\}$
    $Box(T)$               $\rightarrow$ if $T \in RefType$ then $\{(pos+1, argZeroT, pop(evalStackT) \cdot [T])\}$
                           else $\{(pos+1, argZeroT, pop(evalStackT) \cdot [boxed(T))]\}$
    $Unbox(T)$             $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T\&])\}$
    $Unbox.Any(T)$         $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T])\}$
    $MkRefAny(\_)$         $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [\texttt{typedref}])\}$
    $RefAnyType$           $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [\texttt{System.RuntimeTypeHandle}])\}$
    $RefAnyVal(T)$         $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T\&])\}$
    $CastClass(T)$         $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T])\}$
    $IsInstance(T)$        $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [T]),$
                            $(pos+1, argZeroT, pop(evalStackT) \cdot [Null])\}$
    $LoadFtn(\_)$          $\rightarrow \{(pos+1, argZeroT, evalStackT \cdot [\texttt{native int}])\}$
    $LoadVirtFtn(\_)$      $\rightarrow \{(pos+1, argZeroT, pop(evalStackT) \cdot [\texttt{native int}])\}$
```

- *evalStackT is set to* $[OC]$ *since UnInit is replaced by OC (therefore, the call at code index* 7 *is verifiable) and*
- *argZeroT is set to OC (this guarantees that the return at code index* 8 *is verifiable)*

The successor of a *CallVirt* instruction is defined similarly to the successor of a *Call* instruction for the case of a non-constructor method (we mentioned in Section 4.1.3.1 that the `.ctor`s cannot be called with *CallVirt*).

If the *Call* instruction is used in a tail call, the function *succ* contains no successors since the current frame gets immediately discarded. Similarly, the *Return* instruction has no successors since its effect is to leave the current method.

The type on top of the stack state in the successor of the *NewObj* instruction is the class of the `.ctor` given in the instruction. The successors of the instructions *LoadField* and *LoadFieldA* use the declared type of the field given in the instructions. In our abstract instructions, the declared type is stored as the first parameter in the above instructions.

For the *LoadInd* instruction, we make a case distinction. If the instruction is used in the form *LoadInd*(`object`), *i.e.*, for loading an object reference, the type added to the stack state of the successor is the referent type of the topmost type[14] of the current stack state. If the instruction is applied with a primitive type, this type is added to the stack state of the successor.

The successor of the instruction *Box* has on top of the stack state the boxed type corresponding to the instruction's type token argument. The successors of the *Unbox.Any* and *Unbox* instructions have on top of the stack state the type referred to by the instruction's type token and a pointer type whose referent type is the instruction's type token, respectively.

A `typedref` is pushed onto the stack by the *MkRefAny* instruction. The instruction *RefAnyType* pushes onto the stack a `System.RuntimeTypeHandle`. The successor of the *RefAnyVal* instruction has on top of the stack state the pointer type whose referent type is the instruction's type token. If the *RefAnyVal* instruction did not have a type token argument, the type placed on the stack by this instruction would have been unknown. In this case, the instruction would not have been verifiable. In contrast, the *RefAnyType* instruction does not require a type token argument as it always loads a type handle on the stack. On the other hand, the *RefAnyVal* instruction has to perform a run-time check.

The instruction $\mathit{CastClass}(T)$ leaves on the stack an object of type T. The instruction $\mathit{IsInstance}(T)$ has two possible successors: Depending if the class of the object on the stack is a subtype of T, T or the type *Null* are added to the stack state.

The instructions *LoadFtn* and *LoadVirtFtn* load onto the stack a `native int` representing a method pointer.

4.1.5 The Bytecode Verification Algorithm and Well-typed Methods

The bytecode verification algorithm, vaguely specified in [45], is described in [65]. Its formal specification, inspired by [115], is given in Figure 4.10. The verifier attempts to determine a valid type state for every instruction of a method. For that, the algorithm uses the dynamic functions described in Table 4.11. The set $\mathcal{V}$ of code indices *visited* by the algorithm, *i.e.*, the

[14]For *LoadInd*, the topmost type of the stack state is assumed to be a pointer type.

Function definition			Function name
isVisited	:	*Map*(*Pc*,*Bool*)	visited positions
isPending	:	*Map*(*Pc*,*Bool*)	not-yet-verified positions
argZeroV	:	*Map*(*Pc*, *VerificationType*)	`this`' inferred type
evalStackV	:	*Map*(*Pc*, *List*(*VerificationType*))	stack slots' inferred types

Table 4.11: The dynamic functions of the bytecode verification algorithm.

indices that have already a type state assigned, is defined in terms of the function *isVisited*. The set $\mathcal{P}$ of code indices that are *still to be explored* is defined through the function *isPending*. The indices in $\mathcal{P}$ may be revisited with refined type states.

$$\begin{aligned}
\mathcal{V} &= \{i \in Pc \mid isVisited(i) = True\} \\
\mathcal{P} &= \{i \in Pc \mid isPending(i) = True\}
\end{aligned}$$

Initial Constraints The following conditions should be satisfied in the initial state of the bytecode verification algorithm run for a method *mref*:

$$\mathcal{V} = \{0\}$$

$$\mathcal{P} = \{0\}$$

$$argZeroV_0 = \begin{cases} UnInit, & \text{if } classNm(mref) \in ObjClass \setminus \{\texttt{object}\} \text{ and} \\ & methNm(mref) = \texttt{.ctor}; \\ argTypes(mref)(0), & \text{otherwise.} \end{cases}$$

$$evalStackV_0 = [\,]$$

Initially, the type tracked for the zeroth argument of an object class `.ctor` (other than `object`::`.ctor`) is *UnInit*. The stack state is empty upon the method entry.

When propagated, the type states are merged as described in Section 4.1.4.1: The types tracked for the zeroth arguments are merged to *the least common supertype*, while the stack states are similarly merged, but component-wise (see also Remark 4.1.8). We assume that as soon as *halt* is set, the verification algorithm stops and consequently, the verified method is rejected by the algorithm.

Remark 4.1.8 *The least common supertype is* not *always unique, since an object class or an interface may implement multiple interfaces. Surprisingly, this aspect is not clarified in* [45]. *An elegant solution, described in detail by Stärk et al. [115, §16.1.2], is to allow finite* sets of object types *in the bytecode verification process. This approach requires extending the compatibility relation $\sqsubseteq$ to such sets. Two sets are related through the new relation if for every type in the first set, there exists in the second set a supertype of the type in the first set. This relation*

Figure 4.10 The bytecode verification algorithm.

```
ONESTEPVERIFICATION ≡
  choose $pos \in \mathcal{P}$ do
    if $check(mref, pos, argZeroV_{pos}, evalStackV_{pos})$ then
      PROPAGATE($pos$)
      $isPending(pos) := undef$
    else $halt :=$ "Check failed"

PROPAGATE($pos$) ≡
  forall $(i, argZeroT', evalStackT') \in succ(mref, pos, argZeroV_{pos}, evalStackV_{pos})$ do
    PROPAGATESUCC($i, argZeroT', evalStackT'$)

PROPAGATESUCC($i, argZeroT', evalStackT'$) ≡
  if $\neg\, isVisited(i)$ then
    if $0 \leq i \wedge i < length(code(mref))$ then
      $argZeroV_i \;\; := argZeroT'$
      $evalStackV_i := evalStackT'$
      $isVisited(i) \;\; := True$
      $isPending(i) := True$
    else $halt :=$ "Invalid code index"
  elseif $argZeroT' \sqsubseteq argZeroV_i \wedge evalStackT' \sqsubseteq_{len} evalStackV_i$ then
    skip
  elseif $argZeroT'$ and $argZeroV_i$ have a common supertype $\wedge$
        $length(evalStackT') = length(evalStackV_i) \wedge$
        $evalStackT'(k)$ and $evalStackV_i(k)$ have a common supertype,
        for every $k = 0, length(evalStackT') - 1$
  then
    $argZeroV_i \;\; := sup(argZeroT', argZeroV_i)$
    $evalStackV_i := [sup(evalStackT'(k), evalStackV_i(k)) \mid k = 0, length(evalStackT') - 1]$
    $isPending(i) := True$
  else $halt :=$ "Propagation not possible"
```

could be introduced without harm into our approach. To keep the presentation simple, we do not consider it here explicitly though the bytecode verification "merge" operations mutually assume *it. It must, however, be emphasized that the verifier's soundness and completeness proofs would follow absolutely the same lines if sets of object types were introduced in the verification type system.*

Remark 4.1.9 *The bytecode verification algorithm terminates. This can be easily shown by taking into account that the number of elements in the state of the algorithm cannot exceed the total number of bytecode instructions, and with each algorithm iteration the type state of at least one instruction is "increased" according to the ordering $\sqsubseteq$ (upon "merging"), and this ordering is finite.*

Example 4.1.8 *To illustrate the merging of stack states, we consider the example in Figure 4.11: a bytecode fragment on the left and stack states on the right. Let us assume that*

Figure 4.11 Example: merging stack states.

```
.method public hidebysig specialname rtspecialname
        instance void .ctor() cil managed
{
  .maxstack 4                                                          evalStackT
  .locals init (class OC′ V_0, class OC V_1, int32 V_2)
  0:   ldarg  0                                                         [ ]
  1:   ldloc  1                                                         [UnInit]
  2:   ldloc  2                                                         [OC, UnInit]
  3:   ldc.i4 0                                                         [int32, OC, UnInit]
  4:   bgt    9                                                         [int32, int32, OC, UnInit]
  5:   pop                                                              [OC,UnInit]
  6:   dup                                                              [UnInit]
  7:   call   instance void System.Object::.ctor()                      [UnInit,UnInit]
  8:   ldloc  0                                                         [OC″]
  9:   stfld  OC  OC″::F                                                ?
  10:  ret                                                              ?
} // end of method OC″::.ctor
```

this fragment is the body of a constructor OC''::`.ctor` *of an object class OC'', derived from* `object`*, which declares a field OC''::F of type the object class OC. Moreover, we assume that OC' is an object class that extends OC.*

The instruction at 9 is a merge point. More precisely, the instruction `stfld` *OC OC''::F is the successor of two instructions, namely* `bgt` 9 *and* `ldloc` 0*. The stack state after the instruction* `bgt` 9 *is $[OC, UnInit]$, whereas the stack state following the instruction* `ldloc` 0 *is $[OC', OC'']$. The two stack states cannot be merged since the verification types OC'' and UnInit have no common supertype*[15]*. So, this constructor violates the object initialization rules, and therefore, it is not considered well-typed. Basically, as the* `this` *pointer is not initialized on a flow path to 9, the bytecode verification cannot determine the initialization status at the merge point 9.*

If the `this` *pointer were initialized before the branch instruction, the stack states would have been $[OC, OC'']$ and $[OC', OC'']$. As $OC' \sqsubseteq OC$, the two stack states can be merged into $[OC, OC'']$, and thus, the constructor would be regarded as well-typed.*

We need a characterization of the type properties satisfied by the methods accepted by the bytecode verification. For this purpose, we introduce the notion of well-typed methods in Definition 4.1.9. We say that a method is *well-typed* if it is possible to assign a valid type state to every instruction of the method. More precisely, a method is well-typed if there exists a type state family that satisfies certain initial conditions, the type-consistency checks defined in Section 4.1.4.3 for all instructions as well as the relations dictated by execution simulation of the bytecode (see the function *succ* defined in Section 4.1.4.4) and by the rules for merging type states specified in [45, Partition III, §1.8.1.3].

Definition 4.1.9 (Well-typed) *A method mref is* well-typed *if there exists a family of type states $(argZeroT_i, evalStackT_i)_{i \in \mathcal{D}}$ over the domain $\mathcal{D}$ which satisfies the following conditions:*

[15]The "supertype" is defined in terms of the compatibility relation $\sqsubseteq$ for verification types.

(wt1) *The elements of $\mathcal{D}$ are valid code indices of mref.*

(wt2) $0 \in \mathcal{D}$.

(wt3) *If $classNm(mref) \in ObjClass \setminus \{$`object`$\}$ and $methNm(mref) =$ `.ctor`, then $argZeroT_0 = UnInit$. Otherwise, $argZeroT_0 = argTypes(mref)(0)$.*

(wt4) $evalStackT_0 = [\,]$.

(wt5) *If $i \in \mathcal{D}$, then $check(mref, i, argZeroT_i, evalStackT_i)$ is true.*

(wt6) *If $i \in \mathcal{D}$ and $(j, argZeroT', evalStackT') \in succ(mref, i, argZeroT_i, evalStackT_i)$, then $j \in \mathcal{D}$, $argZeroT' \sqsubseteq argZeroT_j$ and $evalStackT' \sqsubseteq_{len} evalStackT_j$.*

The domain $\mathcal{D}$ of the family denotes the code indices which are reachable from the code index 0. **(wt1)** states that $\mathcal{D}$ consists of valid code indices only and **(wt2)** says that the code index 0 is in the domain. **(wt3)** and **(wt4)** mention conditions for the type state of the code index 0. Thus, the type tracked for the zeroth argument of an object class `.ctor` (other than `object`::`.ctor`) is initially considered *UnInit*. The evaluation stack has to be empty upon the method entry. **(wt5)** ensures that the type states satisfy all type-consistency checks. **(wt6)** says that a successor type state has to be more specific than the type state corresponding to the successor index. In particular, this means that the stack state asserted in the successor should me more specific, but of the same length as the stack state associated in the type state. The reasons for these conditions are to be found in the definition of the stack state "merging" described in Section 4.1.4.1.

Given a method, the bytecode verification algorithm should decide whether it is possible to assign a type state to every instruction of the method. In the positive case, the algorithm computes a *most specific* family of type states the method is well-typable with. Definition 4.1.10, inspired by [115], specifies what we mean by "more specific":

Definition 4.1.10 (More specific type states) *A type state family $(argZeroV_i, evalStackV_i)_{i \in \mathcal{V}}$ is more specific than a family of type states $(argZeroT_i, evalStackT_i)_{i \in \mathcal{D}}$ if the following conditions are met:*

- $\mathcal{V} \subseteq \mathcal{D}$,
- $argZeroV_i \sqsubseteq argZeroT_i$, *for every* $i \in \mathcal{V}$,
- $evalStackV_i \sqsubseteq_{len} evalStackT_i$, *for every* $i \in \mathcal{V}$.

4.1.6 Type Safety

If a method is legal, well-typed and, in addition, the bytecode structure satisfies the restrictions stipulated in Section 4.1.4.2, then several properties are guaranteed to hold at its run-time execution:

Type safety: Every instruction executes with arguments of expected types. The evaluation stack has values of the types assigned in the stack state and the same length as the stack state. The values stored in the local variables and arguments are of the types given in the “local signature” and method signature, respectively. The objects on the heap, including the boxed objects, are of the expected static types. The fields of the objects on the heap have values of the corresponding declared types.

Object initialization: Every object on the heap is fully initialized before is used properly.

Bounded evaluation stack: The evaluation stack does not exceed the upper bound specified by the method. No instruction attempts to pop a value from an empty evaluation stack.

Code containment: The program counter never leaves the code array of the method. In particular, it must not fall off the end of the method’s code.

To reason if a value is *typable* with a type, we introduce a *typing judgment*. As pointed out in Section 4.1.2, the values are tagged. The type of a value that is neither a pointer nor a value class nor a typed reference, *i.e.*, a value that is not tagged with $\&$ or a value class or `typedref`, is given by the function *type*. For a primitive type, *type* simply returns the tag type *tagType*. For an object reference, *type* is given by the function *verifType*, which, based on the values of *initState*, assigns to every object reference its verification type.

$$type(val) = tagType(val), \text{ if } tagType(val) \notin \{\mathtt{ref}, \&\} \cup ValueClass$$

$$type(val) = verifType(val), \text{ if } tagType(val) = \mathtt{ref}$$

$$\begin{array}{l} verifType(ref) = \textbf{if } ref = \mathtt{null} \textbf{ then } Null \\ \qquad\qquad\qquad \textbf{else match } initState(ref) \\ \qquad\qquad\qquad\qquad InProgress(T) \rightarrow UnInit \\ \qquad\qquad\qquad\qquad Init(T) \qquad\quad\; \rightarrow T \end{array}$$

The most interesting part of the typing judgment concerns the pointer types and the type `typedref`: When is a pointer of type $T\&$? when is a typed reference of type `typedref`? To answer these questions, we need the predicate *isAddressIn*, defined similarly as in the C$^\sharp$’s type safety proof (see Definition 3.6.3 in Chapter 3). Thus, given two types T, T' and two addresses *adr* and *adr'*, $isAddressIn(adr, T, adr', T')$ holds if a value of type T is stored at *adr*, which “is an address in” the memory block pointed to by *adr'*, where a value of type T' is stored.

Definition 4.1.11 *For the types T, T′ and the addresses adr, adr′, we define*

$$\begin{array}{l} isAddressIn(adr, T, adr', T') :\Leftrightarrow \\ \quad (adr = adr' \wedge T = T') \vee \\ \quad (T' \in ValueClass \wedge \\ \qquad \exists\, T'{::}F \in instFields(T') \;\; : \\ \qquad\quad isAddressIn(adr, T, fieldAdr(adr', T'{::}F), fieldType(T'{::}F))) \end{array}$$

The next definition introduces the typing rules. The typing is with respect to the heap *actualTypeOf*, the current *frame*, and the *frameStack*.

Definition 4.1.12 (Typing judgment) *Let val be a value, T a verification type and $[fr_1, \ldots, fr_k]$ a list of frames. The typing $[fr_1, \ldots, fr_k] \vdash val : T$ holds if at least one of the following conditions is met:*

- $T \in PrimitiveType \cup RefType \cup \{UnInit, Null\}$ *and* $type(val) \sqsubseteq T$, *or*
- $T \in ValueClass$, $val \in Map(instFields(T), Val)$, *and for every* $T::F \in instFields(T)$, $[fr_1, \ldots, fr_k] \vdash val(T::F) : fieldType(T::F)$, *or*
- $val \in ObjRef$ *and there exists* $T' \in ValueType$ *such that* $actualTypeOf(val) = T'$ *and* $boxed(T') \sqsubseteq T$, *or*
- T *is* `typedref` *and* $[fr_1, \ldots, fr_k] \vdash typedRefAdr(val) : typedRefType(val)\&$, *or*
- T *is a pointer type* $T'\&$ *and at least one of the following conditions is satisfied:*

(tj-null) *val is* `null`, *or*

(tj-loc) *there exists* $i = 1, k$ *and* $n = 0, locNo(meth(fr_i)) - 1$ *such that* $isAddressIn(val, T', locAdr(fr_i)(n), locTypes(meth(fr_i))(n))$ *holds, or*

(tj-pool) *there exists* $i = 1, k$ *and* $(val', T'') \in locPool(meth(fr_i))$ *such that* $isAddressIn(val, T', val', T'')$ *holds, or*

(tj-arg) *there exists* $i = 1, k$ *and* $n = 0, argNo(meth(fr_i)) - 1$ *such that* $isAddressIn(val, T', argAdr(fr_i)(n), argTypes(meth(fr_i))(n))$ *holds, or*

(tj-field) *there exists* $OC \in ObjClass$, $OC'::F \in instFields(OC)$ *and* $ref \in ObjRef$ *such that* $isAddressIn(val, T', fieldAdr(ref, OC'::F), fieldType(OC'::F))$ *holds and* $actualTypeOf(ref) = OC$, *or*

(tj-box) *there exists* $T'' \in ValueType$ *and* $ref \in ObjRef$ *such that* $actualTypeOf(ref) = T''$ *and* $isAddressIn(val, T', addressOf(ref), T'')$ *holds.*

A value is typable with a value class if is a mapping of the value class instance fields into values of the field declared types. A typed reference is typable with `typedref` if the pointer embedded in the typed reference is typable with the pointer type whose referent type is embedded in the typed reference. A pointer *adr* is typable with $T = T'\&$ if *adr* is: `null`[16], the address of a local variable or argument of a frame (the current *frame* or a frame on the *frameStack*), or the address of an object class instance field of declared type T', or the address of a value type in a boxed value on the heap, or an address typable with T in a memory block pointed to by one of the addresses of the above locations.

The method pointers are not treated specially by Definition 4.1.12 since they are regarded as values tagged with `native int`. It is then immediate that they are typable with `native int`.

[16]A `null` pointer can only occur when a local variable, whose declared type is a pointer type, is accessed in its default state, resulted upon the method's entry. Given the issue pointed out in Remark 4.1.1, this typing ensures the typing of such a local variable.

Remark 4.1.10 *Through the definition of the typing judgment, it should become clear why do we have to consider the local memory pool in our framework: Without locPool, the obviously correct typing of the managed pointers to the value class instances (constructed with NewObj) would have got "lost". It would then have been impossible to prove the correct typing of the* `this` *pointer in a value class* `.ctor`*, invoked through a NewObj instruction.*

For the type safety proof, we need several lemmas. The next lemma claims that after storing a value of type T at an address *adr* of a memory block, sufficient to hold values of a T''s supertype, the value read at *adr* is of type T.

Lemma 4.1.3 *Let T and T' be types such that $T \sqsubseteq T'$. We assume that $[fr_1, \ldots, fr_k] \vdash val : T$ and $[fr_1, \ldots, fr_k] \vdash adr : T\&$. Then, after the macro* WRITEMEM(adr, T', val) *is executed, we have $[fr_1, \ldots, fr_k] \vdash memVal(adr, T) : T$.*

Proof. By induction on the structure of the possibly value class T, using Lemma 4.1.1. □

Let adr'' be an address of a memory block, sufficient to hold values of a type T''. Assume that, in this block, adr' is an address where a value of type T' is stored. Storing a value of type T' at adr' raises the following concern: Is the value stored at adr'' still of type T''? The following lemma addresses this issue.

Lemma 4.1.4 *If $[fr_1, \ldots, fr_k] \vdash val : T'$, $[fr_1, \ldots, fr_k] \vdash memVal(adr'', T'') : T''$, $T' \sqsubseteq T$ and $isAddressIn(adr', T', adr'', T'')$, then the execution of* WRITEMEM(adr', T, val) *preserves $[fr_1, \ldots, fr_k] \vdash memVal(adr'', T'') : T''$.*

Proof. By induction on the definition of the *isAddressIn* predicate. The base case of the induction is proved by applying Lemma 4.1.3. □

Definition 4.1.12 does not contain a *subsumption rule*. However, this can be easily derived: If *val* is typable with T and T is compatible with T', then *val* can also be typable with T'.

Lemma 4.1.5 *If $[fr_1, \ldots, fr_k] \vdash val : T$ and $T \sqsubseteq T'$, then $[fr_1, \ldots, fr_k] \vdash val : T'$.*

Proof. By induction on the definition of $\vdash$. □

The following lemma establishes that if the address of a value class instance "is an address in" a memory block, then also the address of each instance field of the value class instance "is an address in" that memory block.

Lemma 4.1.6 *If $VC \in ValueClass$ and $isAddressIn(adr, VC, adr', T')$, then*

$$isAddressIn(fieldAdr(adr, VC{::}F), fieldType(VC{::}F), adr', T')$$

for every $VC{::}F \in instFields(VC)$.

Proof. By case distinction, applying Definition 4.1.11. □

The next lemma says that the typing is preserved upon pushing new frames. The typing with a non-pointer type other than `typedref` is also preserved upon popping frames.

Lemma 4.1.7 *Assume that* $[fr_1, \ldots, fr_k] \vdash val : T$*. If* fr *is a frame, then* $[fr_1, \ldots, fr_k, fr] \vdash val : T$*. If* T *is neither a pointer type nor* `typedref`*, then* $[\,] \vdash val : T$ *and* $[fr_1, \ldots, fr_i] \vdash val : T$ *for every* $i = 1, k$*.*

Proof. In Definition 4.1.12, the list of frames is relevant only for **(tj-loc)**, **(tj-arg)**, and the typing of a typed reference. However, in these cases, T is a pointer type or `typedref`. □

The following two lemmas are required to ensure type safety of dynamically dispatched method calls. Lemma 4.1.8 relates the argument types (including the type of the `this` pointer) of the called method and the invoked one. Similarly, Lemma 4.1.9 establishes the relation between return types.

Lemma 4.1.8 *If* $T''::M = lookUp(T, T'::M)$ *holds for a type* T*, then*

- *If* $T \in ValueType$*, then*
  - *if* $T'' \neq T$*, then* $boxed(T) \sqsubseteq argTypes(T''::M)(0) \sqsubseteq argTypes(T'::M)(0)$
  - *if* $T'' = T$*, then* $argTypes(T''::M)(0) = T\&$ *and* $boxed(T) \sqsubseteq argTypes(T'::M)(0)$

  If $T \in RefType$*, then* $T \sqsubseteq argTypes(T''::M)(0) \sqsubseteq argTypes(T'::M)(0)$
- $argNo(T'::M) = argNo(T''::M)$ *and for every* $i = 1, argNo(T'::M) - 1$

$$argTypes(T'::M)(i) = argTypes(T''::M)(i)$$

Proof. By Definition 4.1.4 and **(at)** in the group of constraints **[override/implement]**. □

Lemma 4.1.9 *If* $T''::M = lookUp(T, T'::M)$ *holds for a type* T*, then* $retType(T''::M) = retType(T'::M)$*.*

Proof. By Definition 4.1.4 and **(at)** in the group of constraints **[override/implement]**. □

We say that a *frame contains an object reference ref* with the assigned type *UnInit* if *ref* is the `this` pointer for which the bytecode verification expects the type *UnInit* or *ref* is in an evaluation stack slot, where the verification expects *UnInit*.

Definition 4.1.13 *Let* $fr = (pc^*, locAdr^*, locPool^*, argAdr^*, evalStack^*, meth^*)$ *be a frame and* $(argZeroT_j, evalStackT_j)_{j \in \mathcal{D}}$ *be a family of type states* $meth^*$ *is well-typable with. Let* $argVal^*$ *be the function* $argVal$ *determined based on* $argAdr^*$*. The frame* fr contains a reference *ref* with the assigned type *UnInit if one of the following conditions is fulfilled:*

- $argVal^*(0) = ref$ *and* $argZeroT_{pc^*} = UnInit$*, or*

- *there exists* $i = 0, length(evalStack^*) - 1$ *with* $evalStack^*(i) = ref$ *and* $evalStackT_{pc^*}(i) =$ *UnInit.*

In the theorem asserting type safety of well-typed methods, we prove invariants for *frame* as well as for the frames on the *frameStack*. However, an invariant for a frame *fr* in the *frameStack* has to be considered as *fr* would be the current frame. But, when *fr* becomes the current frame, the initialization status of certain object references as well as the *evalStack* of *fr* might have been changed. This remark triggers the next definitions, inspired by [115].

Definition 4.1.14 (Succession of a frame) *Let the list* $[fr_1, \ldots, fr_k]$ *be the frameStack and* $fr_i = (pc^*, locAdr^*, locPool^*, argAdr^*, evalStack^*, meth^*)$ *be an arbitrary frame in the frameStack. If mref is method reference of* fr_i*'s callee frame (for* fr_k*, the callee frame is frame), the succession of* fr_i *is defined as follows:*

- *If mref returns a value, say val, with* $[fr_1, \ldots, fr_i] \vdash val : retType(mref)$*, then* $(pc^*{+}1, locAdr^*, locPool^*, argAdr^*, evalStack^* \cdot [val], meth^*)$ *is called* the succession *of* fr_i *in frameStack.*
- *If retType(mref) is* `void`*, then* the succession *of* fr_i *in frameStack is given by* $(pc^* + 1, locAdr^*, locPool^*, argAdr^*, evalStack^*, meth^*)$.

Definition 4.1.15 (Succession of the initialization status) *A function* $initState^*$ *is the* succession *of initState for a frame fr in the frameStack if* $initState^*$ *is defined as follows.*

- *If fr is immediately followed in the frameStack by an object class* `.ctor` *with the* `this` *pointer ref, then* $initState^*(ref') = initState(ref')$ *for every reference* $ref' \neq ref$*, and* $initState^*(ref) = Init(OC)$*, where OC is the object class given by* $initState(ref) = InProgress(OC)$.
- *If fr is not followed by an object class* `.ctor`*, then* $initState^*$ *coincides with initState.*

Theorem 4.1.1 proves type safety of legal and well-typed methods. Thus, the following invariants are guaranteed to hold at run-time. The invariant **(pc)** implies that the program counter is always a valid code index. By **(stack1)**, we know that the *evalStack* has the same length as the assigned stack state and will never overflow. That the values on the *evalStack* are of the types assigned in the stack state is ensured by **(stack2)**. By **(loc)** and **(arg)**, we have that the local variables and arguments contain values of the declared types. In case of a `.ctor`, the value of the `this` pointer is of the type assigned in the type state. The invariant **(init)** makes precise when a value has the type *UnInit* assigned in the type state. The invariant **(field)** ensures that the fields of an object class instance are of the declared types. That the value type instance embedded in a boxed value is of the expected value type is guaranteed by the invariant **(box)**. The invariant **(del)** is similar with corresponding invariant for C$^\sharp$ in Theorem 3.6.1. It claims that every target object in the invocation list of a delegate is an object of an appropriate type, *i.e.*, a subtype of the type that declares the corresponding target method. It also guarantees that the delegate class is compatible, according to Definition 4.1.8, with the target method.

Theorem 4.1.1 (Type safety of well-typed methods) *Let $[fr'_1, \dots, fr'_{k-1}]$ be the frameStack and fr_k be the current frame. For every $i = 1, k-1$, we denote by fr_i the succession of fr'_i.*

We assume that the type system and the bytecode structure of the methods in $(fr_i)_i$ satisfy the conditions stated in Section 4.1.4.2. Moreover, we assume that all methods of $(fr_i)_{i=1}^{k}$ are well-typed.

Let $fr_i = (pc^*, locAdr^*, \_, argAdr^*, evalStack^*, meth^*)$ be one of the frames $(fr_i)_{i=1}^{k}$. We denote by $locVal^*$ and $argVal^*$ the functions locVal and argVal derived based on $locAdr^*$ and $argAdr^*$, respectively. If fr_i is the current frame, we consider $initState^*$ to be the current initialization status initState. If fr_i is not the current frame, we denote by $initState^*$ the succession of initState for fr_i. Let $(argZeroT_j, evalStackT_j)_{j \in \mathcal{D}}$ be a family of type states $meth^*$ is well-typable with.

The following invariants are satisfied at run-time for every frame fr_i (for the current frame, fr_k, the invariants **(stack1)** *and* **(stack2)** *should* only *hold if switch = Noswitch):*

(pc) $pc^* \in \mathcal{D}$.

(stack1) $length(evalStack^*) = length(evalStackT_{pc^*}) \leq maxEvalStack(meth^*)$.

(stack2) $[fr_1, \dots, fr_i] \vdash evalStack^*(j) : evalStackT_{pc^*}(j)$, *for every* $j = 0, length(evalStack^*) - 1$.

(loc) $[fr_1, \dots, fr_i] \vdash locVal^*(n) : locTypes(meth^*)(n)$, *for every* $n = 0, locNo(meth^*) - 1$.

(arg) *If* $classNm(meth^*) \in ObjClass$ *and* $methNm(meth^*) =$ `.ctor`, *then it holds* $[fr_1, \dots, fr_i] \vdash argVal^*(0) : argZeroT_{pc^*}$; *otherwise,* $[fr_1, \dots, fr_i] \vdash argVal^*(0) : argTypes(meth^*)(0)$.

If $argNo(meth^*) \geq 2$, *then* $[fr_1, \dots, fr_i] \vdash argVal^*(n) : argTypes(meth^*)(n)$, *for every* $n = 1, argNo(meth^*) - 1$.

(init) *If fr_i contains $ref \in ObjRef$ with the assigned type UnInit, then $meth^*$ is an object class* `.ctor` *(not of class* `object`*), $ref = argVal^*(0)$, and $initState^*(ref) = InProgress(OC)$, where OC is an object class such that $OC \preceq classNm(meth^*)$.*

(field) *If $ref \in ObjRef$ is such that $actualTypeOf(ref) = OC$, where $OC \in ObjClass$, and $OC'::F \in instFields(OC)$ has the declared type T, then*

$$[\,] \vdash memVal(fieldAdr(ref, OC'::F), T)) : T$$

(box) *If $ref \in ObjRef$ is such that $actualTypeOf(ref) = T$, where $T \in ValueType$, then $[\,] \vdash memVal(addressOf(ref), T)) : T$.*

(del) *If $ref \in ObjRef$ is such that $actualTypeOf(ref) = DC$, where $DC \in DelegateClass$, then the following hold for every $(ref', T::M) \in invocationList(ref)$:*

- *$ref' \in ObjRef$*
- *$actualTypeOf(ref') \preceq T$.*
- *DC is compatible with T::M.*

Proof. The invariants are proved by induction on the run of the operational semantics model defined in Section 4.1.3.1.

Base case In the initial state of the machine $\text{CLR}_{\mathcal{L}}$, the invariants for the single existing frame (of the `.entrypoint` method) are satisfied as follows. **(pc)** holds since $pc = 0$ and $0 \in \mathcal{D}$ (from Definition 4.1.9 **(wt2)**). The invariants **(stack1)** and **(stack2)** follow from $evalStack = [\,]$ and Definition 4.1.9 **(wt4)**. Upon the method's entry, the virtual machine stores, by means of WRITEMEM, "zeros" of the appropriate types at the local variable addresses *locAdr*. By this and Lemma 4.1.3, we derive the invariant **(loc)**. The other invariants obviously hold.

Induction step We assume that all invariants are satisfied in the current state of $\text{CLR}_{\mathcal{L}}$, and we prove that invariants are preserved after $\text{CLR}_{\mathcal{L}}$ executes a step. Different cases have to be considered depending on $code(pc)$. For brevity, we only consider here a few cases.

Case 1 $code(pc) = StoreInd(T)$

By **(pc)** and Definition 4.1.9 **(wt5)**, there exists $evalStackT'$, T' and T'' such that

$$evalStackT' \cdot [T'\&, T''] = evalStackT_{pc} \tag{4.1}$$

and

$$T'' \sqsubseteq T' \sqsubseteq T \tag{4.2}$$

By (4.1), (4.2), **(stack1)** and **(stack2)**, we get that the *evalStack* is of the form $evalStack' \cdot [adr, val]$, where $[fr_1, \ldots, fr_k] \vdash adr : T'\&$, $[fr_1, \ldots, fr_k] \vdash val : T''$ and

$$[fr_1, \ldots, fr_k] \vdash evalStack'(j) : evalStackT'(j), \text{ for every } j = 0, length(evalStack') - 1 \tag{4.3}$$

Definition 4.1.9 **(wt6)** implies

$$evalStackT' \sqsubseteq_{len} evalStackT_{pc+1} \tag{4.4}$$

When the instruction $StoreInd(T)$ is executed, the new evaluation stack is $evalStack'$ and $\text{WRITEMEM}(adr, T, val)$ is fired. The invariants **(stack1)** and **(stack2)** are implied by (4.3) and (4.4).

None of the real CIL `stind` instructions defined in [45, Partition III, §3.62] has a pointer type or `typedref` as a type token argument. Therefore, T is neither a pointer type nor `typedref`. By this and Lemma 4.1.2, we get that T'' is neither a pointer type nor `typedref`. Concerning the typing of *adr*, by Definition 4.1.12, we have that *adr* is `null` **(tj-null)**, or "is an address in" the memory block pointed to by the address of a local variable **(tj-loc)**, or of an argument **(tj-arg)**, or of an instance field of an object reference **(tj-field)** or of a value embedded in a boxed value **(tj-box)**, or "is an address in" the memory block pointed to by an address allocated in the local memory pool **(tj-pool)**. Since all cases are similar, we only treat here one. Let this be **(tj-loc)**. It means that there exists a frame fr_i and a local variable n in the frame fr_i such that $isAddressIn(adr, T', locAdr(fr_i)(n), locTypes(meth(fr_i))(n))$ holds. Except for **(loc)** corresponding to the frame fr_i, no invariant is affected by WRITEMEM. Therefore, by the induction hypothesis, all invariants except **(loc)** hold in the next execution step of CLR$_{\mathcal{L}}$. For **(loc)**, we prove $[fr_1, \ldots, fr_i] \vdash locVal(fr_i)(n) : locTypes(meth(fr_i))(n)$. By Lemma 4.1.7, we have $[fr_1, \ldots, fr_i] \vdash val : T''$ since T'' is neither a pointer type nor `typedref`. By (4.2) and Lemma 4.1.5, we get $[fr_1, \ldots, fr_i] \vdash val : T'$. Lemma 4.1.4 says that the execution of WRITEMEM(adr, T, val) preserves the typing $[fr_1, \ldots, fr_i] \vdash locVal(fr_i)(n) : locTypes(meth(fr_i))(n)$, and this concludes the proof of this case.

Case 2 $code(pc) = Call(tail, T', T::M)$

Subcase 2.1 Let us first assume that the call is tail, *i.e.*, $tail = True$, and $T::M$ is not a `.ctor`. From the invariant **(pc)** and Definition 4.1.9 **(wt5)**, we obtain that $check(meth, pc, argZeroT_{pc}, evalStackT_{pc})$ holds. Therefore, we have

$$length(evalStackT_{pc}) = argNo(meth) \tag{4.5}$$

and

$$evalStackT_{pc} \sqsubseteq_{suf} argTypes(T::M) \tag{4.6}$$

The relation (4.6) follows from the definition of *argZeroCompat* and the definition of *argTypes* in terms of *paramTypes*.

Let $[T_0, \ldots, T_{n-1}]$ be the list of argument types $argTypes(T::M)$. Since the call is tail, we know that none of $(T_j)_{j=0}^{n-1}$ is a pointer type or `typedref` (see the constraints in Section 4.1.4.2).

The relation (4.5) implies that $evalStackT_{pc}$ should have the length n. We then derive that $evalStackT_{pc}(j) \sqsubseteq T_j$, for $j = 0, n-1$. The invariants **(stack1)** and **(stack2)** ensure that the *evalStack* contains the values $[val_0, \ldots, val_{n-1}]$ such that $[fr_1, \ldots, fr_k] \vdash val_j : evalStackT_{pc}(j)$, for $j = 0, n-1$. By Lemma 4.1.5, we get $[fr_1, \ldots, fr_k] \vdash val_j : T_j$, for $j = 0, n-1$. By this and Lemma 4.1.7, we have

$$[fr_1, \ldots, fr_{k-1}] \vdash val_j : T_j \text{ for every } j = 0, n-1 \tag{4.7}$$

since no $(T_j)_{j=0}^{n-1}$ is a pointer type or `typedref`.

When the *Call* instruction is executed, the current frame fr_k is discarded and not pushed onto the *frameStack*. The semantics model CLR$_{\mathcal{L}}$ does one *intermediate step* between the time the *Call* instruction is encountered and the set up of the new current frame. This step consists in updating *evalStack* to $[\,]$ and *switch* to $Invoke(True, T::M, (val_j)_{j=0}^{n-1})$. After these updates are fired, **(stack1)** and **(stack2)** do *not* hold anymore since $evalStackT_{pc}$ remained unchanged, whereas *evalStack* became $[\,]$. However, as *switch* is not *Noswitch*, one does not have to prove these invariants. Actually, the theorem does not claim that these invariants hold when $switch \neq Noswitch$ since the above intermediate step is a *model specific step*, which *cannot* be observed in the runs of the real CLR (note that *switch* is a model specific global variable).

After the intermediate step, the new current frame, maintained in *frame*, is defined as follows: the *pc* is 0, the *evalStack* is $[\,]$, the local variables hold "zeros" of the appropriate types, and the arguments are $(val_j)_{j=0}^{n-1}$.

The invariants **(pc)**, **(stack1)** and **(stack2)** trivially hold for the new current frame. Upon the method's entry, a WRITEMEM is executed (by means of the macro ALLOCARGLOC) to write the values $(val_j)_{j=0}^{n-1}$ to *argAdr*. By (4.7) and Lemma 4.1.7, we have $[fr_1, \ldots, fr_{k-1}, frame] \vdash val_j : T_j$, for $j = 0, n-1$. It then follows from Lemma 4.1.3 that **(arg)** holds for *frame*.

After setting the new current frame, the callee's frame of fr'_{k-1} is the frame for executing $T{::}M$ and not the previously discarded frame. Therefore, we also have to show the invariant **(stack2)** for the new succession of fr'_{k-1}. If the return type T' of $T{::}M$ is `void`, **(stack2)** follows from the induction hypothesis. If T' is not `void`, let *val* be the return value assumed to satisfy $[fr_1, \ldots, fr_{k-1}] \vdash val : retType(T{::}M)$. From Section 4.1.4.2, we know that $retType(T{::}M) \sqsubseteq retType(meth)$. By this and Lemma 4.1.5, we get $[fr_1, \ldots, fr_{k-1}] \vdash val : retType(meth)$. The invariant **(stack2)** follows then from the induction hypothesis, Definition 4.1.9 **(wt6)**, and Lemma 4.1.5.

Subcase 2.2 Let us suppose that the call is not tail, T is an object class, and M is a `.ctor`. In this case, the return type T' is `void`. By the induction hypothesis **(pc)** and Definition 4.1.9 **(wt5)**, there exists $evalStackT'$, T'', and L such that $evalStackT' \cdot [T''] \cdot L = evalStackT_{pc}$, $L \sqsubseteq_{suf} paramTypes(T{::}\texttt{.ctor})$, and $initCtorCompat(meth, T'', T)$. By the definition of *initCtorCompat*, we obtain that $T'' = UnInit$ or $T'' \sqsubseteq T$ and either $classNm(meth) = T$ or T is the base class of $classNm(meth)$. We will only treat here the case $T'' = UnInit$. The invariant **(arg)** ensures that there exists on the *evalStack* a reference *ref* typable with *UnInit*. By **(init)**, we get that *meth* is a `.ctor` (not of class `object`), $ref = argVal(0)$, and $initState(ref) = InProgress(OC)$, where OC is an object class such that $OC \preceq classNm(meth)$. For the current frame, it only remains to prove **(init)** since the other invariants can be proved with the same arguments as in case of a tail call (**Subcase 2.1**). The invariant **(init)** is preserved since $initState(ref) = InProgress(OC)$ and $OC \preceq T$.

We also need to show the invariants for every succession of the topmost frame fr_k, *i.e.*, the current frame before doing the call, and every succession of the initialization status *initState*. In the succession of fr_k, the *pc* is incremented by 1. In the succession of the initialization status for fr_k, $initState(ref)$ is set to $Init(OC)$, where the reference *ref* is given by $ref = argVal(fr_k)(0)$. By **(pc)** and the definition of *succ* in Figure 4.9, we get that

$$(pc + 1, classNm(meth), evalStackT'' \circ [classNm(meth)/UnInit])$$

is in the set $succ(meth, pc, argZeroT_{pc}, evalStackT_{pc})$, where the list $evalStackT''$ is given by $drop(evalStackT_{pc}, argNo(T{::}\texttt{.ctor}))$. By Definition 4.1.9 **(wt6)**, we can deduce that $classNm(meth) \sqsubseteq argZeroT_{pc+1}$ and

$$evalStackT'' \circ [classNm(meth)/UnInit] \sqsubseteq_{len} evalStackT_{pc+1} \tag{4.8}$$

We have to show the invariant **(stack2)** for the succession of fr_k. By the induction hypothesis, from **(init)** we know that if there is on the evaluation stack a value *val* typable with *UnInit*, then $val = argVal(fr_k)(0) = ref$. Furthermore, in the succession, $initState(val) = Init(OC)$, where $OC \preceq classNm(meth)$. The invariant **(stack2)** follows then from the induction hypothesis, (4.8), Definition 4.1.12, and Lemma 4.1.5.

Case 3 $\quad code(pc) = Return$

From **(pc)** and Definition 4.1.9 **(wt5)**, we obtain $evalStackT_{pc} \sqsubseteq_{len} void(retType(meth))$ and $argZeroT_{pc} \neq UnInit$ if the method *meth* is a `.ctor` other than `object::.ctor`. Upon executing the *Return* instruction, the current frame is discarded, and the topmost frame on the *frameStack* becomes the new current frame. The *pc* of the current frame is incremented by 1. If *meth* returns a value, say *val*, this value is pushed onto the new *evalStack*. By **(stack2)** and Lemma 4.1.5, $[fr_1, \ldots, fr_k] \vdash val : retType(meth)$. By the constraints stated in Section 4.1.4.2, $retType(meth)$ is neither a pointer type nor `typedref`. By Lemma 4.1.7, we get

$$[fr_1, \ldots, fr_{k-1}] \vdash val : retType(meth) \tag{4.9}$$

So, after executing *Return*, the new current frame is exactly fr_{k-1}, *i.e.*, the succession of fr'_{k-1} according to Definition 4.1.14. The current *initState* does not change after executing *Return*. If *Return* does not correspond to a `.ctor`, then *initState* is obviously the succession of itself for the frame fr_{k-1} according to Definition 4.1.15. If *Return* corresponds to a `.ctor`, then, by $argZeroT_{pc} \neq UnInit$ and **(arg)**, we get that $initState(argVal(0))$ cannot be an $InProgress(\_)$ value. Therefore, *initState* is the succession of itself for fr'_{k-1} according to Definition 4.1.15. Therefore, the induction hypothesis and (4.9) imply that the invariants also hold after executing *Return*.

Case 4 $\quad code(pc) = MkRefAny(T)$

We get $evalStackT \sqsubseteq_{suf} [T\&]$ from the invariant **(pc)** and Definition 4.1.9 **(wt5)**. The instruction pops the top value of the *evalStack*. Since this value is supposed to be a pointer, let us denote it by *adr*. By $evalStackT \sqsubseteq_{suf} [T\&]$, **(stack1)**, and **(stack2)**, we have that *adr* is a pointer typed with $T\&$. Formally,

$$[fr_1, \ldots, fr_k] \vdash adr : T\& \tag{4.10}$$

The instruction then pushes a typed reference, say *tr*, onto the *evalStack*. It also sets the functions *typedRefAdr* and *typedRefType* as follows: $typedRefAdr(tr) = adr$ and $typedRefType(tr) = T$. We need to show that the invariant (**stack2**) is preserved.

From the definition of *succ* in Figure 4.9, we have that

$$(pc + 1, argZeroT_{pc}, pop(evalStackT_{pc}) \cdot [\texttt{typedref}])$$

is in the set $succ(pc, argZeroT_{pc}, evalStackT_{pc})$. From Definition 4.1.9 **(wt6)**, we obtain $pc + 1 \in \mathcal{D}$ (where $\mathcal{D}$ is the domain of the family of type states for the current frame fr_k) and $pop(evalStackT_{pc}) \cdot [\texttt{typedref}] \sqsubseteq_{len} evalStackT_{pc+1}$. The first condition ensures that the invariant **(pc)** is preserved. Based on the second condition and on Definition 4.1.7, we know that in order to prove **(stack2)**, it suffices to show $[fr_1, \ldots, fr_k] \vdash tr : \texttt{typedref}$. This typing follows from (4.10) and Definition 4.1.12. □

Remark 4.1.11 CLR$_{\mathcal{L}}$ *follows strictly the* ECMA Standard [45] *and consequently considers that the* `tail` *prefix cannot be ignored. The semantics model, bytecode verification and the* *proof could be, however, easily adapted to take into account the aspect that the prefix can be ignored (when jitting). Thus, in the definition of succ, the tail calls would be treated as regular calls. More precisely, the tail call would be considered as having a successor (namely a Return instruction – see* ***Tail Call Pattern****), as opposed to the actual definition of succ which does not define any successors. That the invariants are preserved upon performing a tail call with the* `tail` *prefix ignored would be proved as in the case of a regular call.*

It is worth mentioning that the restriction ***Tail Call Return Type*** *would become redundant upon redefining the function succ for tail calls. Given that we only deal with well-typed methods, the above restriction would follow from* ***Tail Call Pattern****.*

Bytecode verification's soundness and completeness Well-typed methods' type safety alone is, however, *not* sufficient. We also need to show that (1) the methods accepted by the verification algorithm are well-typed, and (2) if a method is well-typed, then it is accepted by the verification algorithm and the algorithm computes a most specific family of type states. In other words, we have to prove the *soundness* and the *completeness* of the verification algorithm. These proofs, similar to the corresponding proofs developed in [115], require the following lemmas[17]: Lemmas 4.1.10 and 4.1.11 claim that the type checks and successors are monotonic, respectively.

Lemma 4.1.10 (Monotonicity of checks) *Let mref be a method and pos a code index of a bytecode instruction of mref. If* $argZeroT' \sqsubseteq argZeroT$*,* $evalStackT' \sqsubseteq_{len} evalStackT$ *and* $check(mref, pos, argZeroT, evalStackT)$ *is true, then* $check(mref, pos, argZeroT', evalStackT')$ *is true.*

Proof. By a case distinction on the instruction $code(mref)(pos)$. The transitivity of the relation $\sqsubseteq$ and Lemma 4.1.2 are applied. □

Lemma 4.1.11 (Monotonicity of successors) *Let mref be a method and pos a code index of a bytecode instruction of mref. Let us assume that we have* $argZeroT' \sqsubseteq argZeroT$*,* $evalStackT' \sqsubseteq_{len} evalStackT$ *and* $check(mref, pos, argZeroT, evalStackT)$ *is true. Then for every* $(j, argZeroT''', evalStackT''') \in succ(mref, pos, argZeroT', evalStackT')$*, there exists* $(j, argZeroT'', evalStackT'') \in succ(mref, pos, argZeroT, evalStackT)$ *such that*

$$argZeroT''' \sqsubseteq argZeroT'' \text{ and } evalStackT''' \sqsubseteq_{len} evalStackT''$$

Proof. By a case distinction on the instruction $code(mref)(pos)$. The transitivity of the relation $\sqsubseteq$ is used. □

Theorem 4.1.2 (Soundness of the verification algorithm) *During the bytecode verification of a method mref, the following conditions are met:*

(bv1) $\mathcal{P} \subseteq \mathcal{V}$ *and the elements of* $\mathcal{V}$ *are valid code indices of mref.*

[17] The proofs of these lemmas and of the theorems asserting the soundness and completeness of the verification algorithm can also be easily extended for $\text{CLR}_{\mathcal{E}}$ and $\text{CLR}_{\mathcal{G}}$, and therefore we will not reconsider them explicitly.

(bv2) $0 \in \mathcal{V}$.

(bv3) *If* $\mathit{classNm}(\mathit{mref}) \in \mathit{ObjClass} \setminus \{\texttt{object}\}$ *and* $\mathit{methNm}(\mathit{mref}) = \texttt{.ctor}$*, then* $\mathit{argZeroV}_0 = \mathit{UnInit}$*. Otherwise,* $\mathit{argZeroV}_0 = \mathit{argTypes}(\mathit{mref})(0)$.

(bv4) $\mathit{evalStackV}_0 = [\,]$.

(bv5) *If* $i \in \mathcal{V} \setminus \mathcal{P}$*, then* $\mathit{check}(\mathit{mref}, i, \mathit{argZeroV}_i, \mathit{evalStackV}_i)$ *is true.*

(bv6) *If* $i \in \mathcal{V} \setminus \mathcal{P}$ *and* $(j, \mathit{argZeroV}', \mathit{evalStackV}') \in \mathit{succ}(\mathit{mref}, i, \mathit{argZeroV}_i, \mathit{evalStackV}_i)$*, then* $j \in \mathcal{V}$*,* $\mathit{argZeroV}' \sqsubseteq \mathit{argZeroV}_j$ *and* $\mathit{evalStackV}' \sqsubseteq_{\mathit{len}} \mathit{evalStackV}_j$*.*

Proof. By induction on the run of the bytecode verification algorithm in Figure 4.10. □

One can easily notice that, for $\mathcal{P} = \emptyset$, the conditions **(bv1)-(bv6)** correspond to the well-typedness constraints **(wt1)-(wt6)** in Definition 4.1.9. So, if a method is accepted by the verification (in particular, this means that the verification algorithm terminated, *i.e.*, $\mathcal{P} = \emptyset$), then the method is well-typed.

Theorem 4.1.3 (Completeness of the verification algorithm) *If mref is well-typable with a type state family* $(\mathit{argZeroT}_i, \mathit{evalStackT}_i)_{i \in \mathcal{D}}$*, then during the verification of mref's bytecode* $(\mathit{argZeroT}'_i, \mathit{evalStackT}'_i)_{i \in \mathcal{V}}$ *is always more specific than* $(\mathit{argZeroT}_i, \mathit{evalStackT}_i)_{i \in \mathcal{D}}$ *and halt is not set (to a verification error).*

Proof. By induction on the run of the verification algorithm in Figure 4.10, where Lemmas 4.1.10 and 4.1.11 and Definition 4.1.10 are applied. □

4.2 The Exceptions

This section extends the bytecode language introduced in Section 4.1 by exceptions[18]. The ECMA Standard [45] contains only a few yet *incomplete* paragraphs about the exception handling mechanism. A more detailed description of the mechanism can be found in one of the few existing documents on the CLR exception handling mechanism [32]. The mechanism has its origins in the *Windows NT Structured Exception Handling* (SEH), described in [102].

We use three different methods to check the faithfulness (with respect to CLR) of the modelling decisions we had to take where [45] exhibits gaps. First of all, we did a series of experiments with CLR, some of which are made available in [54]. They show border cases which have to be considered to get a full understanding and definition of exception handling in CLR. Secondly, some authoritative evidence for the correctness of the modelling ideas we were led to by our experiments has been provided by Jonathan Keljo [86], the CLR Exception System Manager, who essentially confirmed our ideas about the exception mechanism issues left open in [45]. Last but not least, a way is provided to test the internal correctness of the model presented in this section and its conformance to the experiments with CLR, namely by an executable version [92] of the CLR model.

Challenges The mechanism handling the exceptions is *spectacularly complex*, especially due to the handling of exceptions in *two passes*, but also due to certain exception constructs, known as `filter` handlers, whose treatment, loosely specified by [45], is clarified in this thesis. The meaning of exception `filter`s is not trivial: A `filter` allows disregarding some exceptions depending on the result of executing `filter` code in the context of some caller on the stack. As the handling of an exception is performed in two passes, an important challenge in the study of `filter`s is to deal with exceptions thrown by the `filter` codes. It turns out that the exception mechanism needs to maintain a *stack of passes*. On the other hand, an exception specific instruction (`leave`) may trigger the execution of several exception handlers which in turn may throw exceptions. Consequently, besides the two kinds of exception passes, vaguely specified by [45], we consider *a third kind of pass*, *i.e.*, *Leave* passes.

Once the semantics of the exception handling mechanism has been defined, the type safety proof is pretty straightforward. And yet, there are several non-obvious properties concerning the structural correctness of the exception mechanism that have to be proved. Moreover, the complex control flow graph underlying the execution of the exception mechanism complicates the bytecode verification the type safety proof significantly depends on.

Plan of the section An overall view of the exception mechanism is provided in Section 4.2.1. Sections 4.2.2 and 4.2.3 define the static and dynamic semantics of the mechanism, respectively. The specification of the bytecode verification is extended in Section 4.2.4 to cover the handling of the exceptions. The bytecode language with exceptions is then proved type-safe in Section 4.2.5.

[18]For the sake of clarity, the special `System.ThreadAbortException` [8] is not treated here as its handling goes much beyond the scope of this thesis. The refinements that should be applied to our formal model in order to also treat the handling of this exception are defined in [61].

4.2.1 An Overview of the Exception Handling Mechanism

Exception handling is supported in CIL through *exception objects* and *exception handlers*. The exception handlers are maintained in an *exception handling array* associated with every method. There are four kinds of exception handlers: `catch`, `filter`, `finally`, and `fault`. To aid the reader, Figure 4.12 contains an example for each type of exception handlers. We briefly describe below each kind:

- A `catch` handler consists of a `try` block and a `catch` *handler region*. The handler region handles any exception of the handler specified type or any of its subtypes which occurs in the `try` block.
- A `filter` handler is made of a `try` block, a `filter` *region*, and a `filter` *handler region*. The `filter` region is executed to decide if the handler region handles an exception raised in the `try` block.

  Visual Basic and Managed C++ have special `catch` blocks which can "filter" the exceptions based on the exception type and / or any conditional expression. These are compiled into `filter` handlers in the CIL bytecode. Figure 4.13 presents an example of a Visual Basic `filter`.
- A `finally` handler consists of a `try` block and a `finally` *handler region*. The handler region is executed whenever the `try` block exits, regardless of whether the `try` block exits normally or through an unhandled exception.
- A `fault` handler embodies a `try` block and a `fault` *handler region*. The handler region is executed if the `try` block is exited by an unhandled exception. Unlike a `finally` handler region, it should not be executed if the `try` block is exited normally. Currently, no language (other than CIL) exposes `fault` handlers directly.

Exit from the blocks and regions of the exception handlers is restricted to be accomplished only through certain bytecode instructions. Thus, every normal execution path of a `try` block or a `catch` or `filter` handler region must end with a `leave` instruction. Similarly, every normal execution path of a `finally` or `fault` handler region must end with the `endfinally` instruction. Alternatively, every execution path of a `try` block or a handler region can also terminate with a `throw` instruction. The last instruction of a `filter` region should be the `endfilter` instruction.

We now describe the CLR exception handling mechanism. Every time an exception occurs, control is transferred from "normal" execution to the exception handling mechanism. This mechanism proceeds in two passes. In the first pass, the run-time system runs a "stack walk" searching, in the possibly empty exception handling array associated to the current method, for the first handler that might want to handle the exception:

- a `catch` handler whose type *typeExc* is a supertype of the type of the exception, or
- a `filter` handler – to see whether a `filter` wants to handle the exception, one has first to execute (in the first pass) the code of the filter region in a *separate* frame pushed onto the stack of call frames (detail skipped by [45]): If it returns 1, then it is chosen to handle the exception; if it returns 0, this handler is not good to handle the exception.

Figure 4.12 Example: exception handlers.

```
.try
{
  2: newobj instance void C::.ctor()
  3: call   instance void C::M()
  4: leave  30
} // end try block
⋮
catch NullReferenceException
{
  20: stloc 0
  21: newobj instance void C′::.ctor()
  22: ldloc 0
  23: call instance void C′::M(Exception)
  24: leave 30
} // end handler region
⋮
30: ...
```

```
.try
{
  2: newobj instance void C::.ctor()
  3: call   instance void C::M()
  4: leave  30
} // end try block
⋮
filter // decide if handle exception
{
  10: stloc 0
  11: ldc.i4 1
  12: newobj instance void C″::.ctor()
  13: ldloc 0
  14: call instance int32 C″::M(object)
  15: ldc.i4 2
  16: beq 19
  17: pop
  18: ldc.i4 0
  19: endfilter
} // end filter region
{
  20: stloc 0
  21: newobj instance void C‴::.ctor()
  22: ldloc 0
  23: call instance void C‴::M(object)
  24: leave 30
} // end handler region
⋮
30: ...
```

```
.try
{
  2: newobj instance void C::.ctor()
  3: call   instance void C::M()
  4: leave  30
} // end try block
⋮
finally
{
  7: newobj instance void C′::.ctor()
  8: call   instance void C′::M()
  9: endfinally
} // end handler region
⋮
30: ...
```

```
.try
{
  2: newobj instance void C::.ctor()
  3: call   instance void C::M()
  4: leave  30
} // end try block
⋮
fault
{
  7: newobj instance void C′::.ctor()
  8: call   instance void C′::M()
  9: endfinally
} // end handler region
⋮
30: ...
```

Figure 4.13 Example: Visual Basic `Catch` with `When` clause.

```
Try
    x = M(y)
Catch exc As Exception When M'(y) < 3
    z = M'(4)
End Try
```

If a match is not found in the *faulting frame*, *i.e.*, the frame where the exception has been raised, the calling method is searched, and so on. This search eventually terminates since the exception handling array of the `entrypoint` method has as last entry a so-called *backstop entry* placed there by the operating system.

When a match is found, the first pass terminates, and in the second pass, called “unwinding of the stack”, the exception mechanism walks once more through the stack of call frames to the handler determined in the first pass, but this time executing the `finally` and `fault` handlers and popping their frames. It then starts the corresponding exception handler region.

Wrapping non-`Exception`s There are .NET compliant languages, for instance Managed C++, where one can throw exceptions of any non-pointer type, as opposed to most of the .NET languages that require that every thrown exception is derived from the class `System.Exception` [6], the base class of all exception types. To maintain compatibility between languages, the CLR *wraps* objects that do not derive from `Exception` in a `RuntimeWrappedException` [4] object. One can use a special boolean attribute, called `RuntimeCompatibilityAttribute`, to specify whether exceptions should appear wrapped inside `catch` and `filter` handler regions. The CLR still wraps exceptions even if the attribute `RuntimeCompatibilityAttribute` is used to specify that one does not want them wrapped. In this case, exceptions are unwrapped only inside `catch` and `filter` handlers. Independent of the attribute’s value, the new `RuntimeWrappedException` object, with the originally thrown object as an instance field `WrappedException` of declared type `object`, is propagated. Surprisingly, the ECMA Standard [45] does not mention anything about this special semantics.

4.2.2 The Static Semantics

To model the exception handling, the universe *Switch* of execution modes is extended with the value *ExcMech*.

$$\begin{array}{lll} \mathit{Switch} & = \ldots & \\ & \mid \quad \mathit{ExcMech} & \text{exceptional mode} \end{array}$$

If *switch* is *ExcMech*, the semantics model is in the *exceptional mode*. This means that the exception mechanism defined in Section 4.2.3 has control either to handle an exception or to normally exit an exception block.

The exception specific universes are gathered in Table 4.12, while the basic static (exception specific) function are defined in Table 4.13. Thus, every method has assigned a possibly empty

Universe	Typical use	Element name
Exc	h, h'	exception handlers
ExcRec	*rec*	exception records
LeaveRec	*rec*	leave records
ClauseKind	–	exception handler kinds

Table 4.12: Semantics exception specific universes.

Function definition			Function name
excHA	:	*Map*(*MRef*, *List*(*Exc*))	the exception handling array
wrapNonExc	:	*Bool*	non-`Exception` wrapping

Table 4.13: The basic static exception specific functions.

exception handling array maintained in *excHA*, and every bytecode program has assigned a flag *wrapNonExc*, representing the `RuntimeCompatibilityAttribute`.

The elements of *excHA* are known as *handlers* and are defined as elements of the universe *Exc*.

$$
\begin{array}{llll}
Exc = Exc\,(& clauseKind & : & ClauseKind \\
 & tryStart & : & Pc \\
 & tryLength & : & \mathbb{N} \\
 & handlerStart & : & Pc \\
 & handlerLength & : & \mathbb{N} \\
 & typeExc & : & ObjClass \cup Interface \cup ValueType \\
 & filterStart & : & Pc\ )
\end{array}
$$

$$ClauseKind = \texttt{catch} \mid \texttt{filter} \mid \texttt{finally} \mid \texttt{fault}$$

Any 7-tuple of the above form describes an exception handler of kind *clauseKind* which "protects" the region that starts at *tryStart* and has the length *tryLength*, handles the exception in the handler region that starts at *handlerStart* and has the length *handlerLength*; if the handler is a `catch`, then the type *typeExc* of exceptions it handles is provided, whereas if the handler is a `filter`, then the first instruction of the `filter` region is at *filterStart*. In case of a `filter` handler, the handler region starting at *handlerStart* should immediately follow the `filter` region (this is the reason there is no `filter` region length provided).

Example 4.2.1 *Consider the exception handlers from Figure 4.12. They are represented in the exception handling array through the following entries:*

$$
\begin{array}{llllllll}
(\texttt{catch}, & 2, & 3, & 20, & 5, & \texttt{NullReferenceException}, & undef &) \\
(\texttt{filter}, & 2, & 3, & 20, & 5, & undef, & 10 &) \\
(\texttt{finally}, & 2, & 3, & 7, & 3, & undef, & undef &) \\
(\texttt{fault}, & 2, & 3, & 7, & 3, & undef, & undef &)
\end{array}
$$

The predicates *isInTry*, *isInHandler* and *isInFilter* are used to identify those code indices that belong to a `try` block, handler region, and `filter` region, respectively.

$$isInTry(pos,h) \quad :\Leftrightarrow \quad tryStart(h) \leq pos < tryStart(h) + tryLength(h)$$

$$isInHandler(pos,h) \quad :\Leftrightarrow \quad handlerStart(h) \leq pos < handlerStart(h) + handlerLength(h)$$

$$isInFilter(pos,h) \quad :\Leftrightarrow \quad filterStart(h) \leq pos < handlerStart(h)$$

Lemma 4.2.1 *The predicates isInTry, isInHandler and isInFilter are pairwise disjoint.*

Proof. Based on the lexical nesting constraints of protected blocks specified in [45, Partition I, §12.4.2.7] and the definitions of the predicates. □

We assume all the constraints concerning the lexical nesting of handlers specified in [45, Partition I, §12.4.2.7]. The ordering assumption on handlers given in [45, Partition I, §12.4.2.5] is the following:

Ordering Assumption *If handlers are nested, the most deeply nested handlers should come in the exception handling array before the handlers that enclose them.*

We also assume all the syntactic constraints stated in [45, Partition I, §12.4.2.8] concerning the *Leave* instructions:

- it is not legal to exit with a *Leave* instruction a `filter` region, a `finally`/`fault` handler region;
- it is not legal to branch with a *Leave* instruction into a handler region from outside the region;
- it is legal to exit with a *Leave* a `catch` handler region and branch to any instruction within the associated `try` block, so long as that branch target is not protected by yet another `try` block;
- a *Leave* instruction is executed *only* upon the normal exit from a `try` block or a `catch`/`filter` handler region;
- the target of any branch instruction, in particular of $Leave(target)$, points to an instruction within the same method as the branch instruction;

Moreover, we assume that the `entrypoint` has a *backstop entry*:

Backstop entry *The excHA of the* `entrypoint` *method has as last entry a handler called* backstop entry *which can handle any exception.*

In order to determine the non-`Exception` object wrapped as an instance field in a `RuntimeWrappedException` object, we define the (derived) function $wrappedExc : Map(ObjRef, Val \cup \{undef\})$.

Function definition			Function name
exc	:	$ObjRef \cup \{undef\}$	the current exception
pass	:	$\{StackWalk, Unwind, Leave\}$	the current pass
stackCursor	:	$Frame \times \mathbb{N}$	the current stack cursor
handler	:	$Frame \times \mathbb{N}$	the suitable handler
target	:	Pc	the target of current *Leave* pass
passRecStack	:	$List(ExcRec \cup LeaveRec)$	the pass record stack

Table 4.14: The basic dynamic exception specific functions.

$$\begin{aligned} &wrappedExc(ref) = \\ &\quad mem(fieldAdr(ref, \texttt{RuntimeWrappedException}::\texttt{WrappedException})) \end{aligned}$$

4.2.3 The Dynamic Semantics

The *dynamic state* of the exception mechanism is defined by the basic dynamic functions listed in Table 4.14. Thus, to handle an exception, the exception mechanism needs to record:

- the exception reference *exc*,
- the handling *pass*,
- a *stackCursor*, *i.e.*, the position currently reached in the stack of call frames (a frame) and in the exception handling array (an index in *excHA*),
- the suitable *handler* (also known as the *target handler*) determined at the end of the *StackWalk* pass (if any) is the handler that is going to handle the exception in the pass *Unwind* – until the end of the *StackWalk* pass, *handler* is *undef*.

Every normal execution of a `try` block or `catch`/`filter` handler region must end with an instruction *Leave*(*target*). When doing this, every `finally` code up to the *target* has to be executed. Since `finally` code can throw exceptions, and implicitly determine the exception mechanism to execute exception passes, one needs to record for the *Leave* instruction the currently reached `finally` code as well as the *target*. This triggers one significant contribution of our framework: the introduction of *Leave* passes although they are not mentioned in [45]. A *Leave* pass records the *stackCursor* together with the *target* of the *Leave* instruction that triggered the pass.

$$\begin{array}{lclll} ExcRec & = & ExcRec\,(& exc & : & ObjRef \\ & & & pass & : & \{StackWalk, Unwind\} \\ & & & stackCursor & : & Frame \times \mathbb{N} \\ & & & handler & : & Frame \times \mathbb{N}\,) \\ \\ LeaveRec & = & LeaveRec\,(& pass & : & \{Leave\} \\ & & & stackCursor & : & Frame \times \mathbb{N} \\ & & & target & : & Pc\,) \end{array}$$

The nesting of passes determines the exception handling mechanism to maintain a stack *passRecStack* of *exception* or *leave records* for the passes that are still to be performed. The need for introducing the *passRecStack* is not revealed, and constitutes another important contribution of our framework. Analogously to *frameStack*, the *passRecStack* does not contain the currently handled pass record.

Initial Constraints The stipulations defined in Section 4.1.3 on the initial state of the virtual machine are now extended to also include the basic dynamic exception specific functions:

$$\begin{array}{lcl@{\qquad\qquad}lcl} \mathit{passRecStack} & = & [\,] & \mathit{stackCursor} & = & \mathit{undef} \\ \mathit{exc} & = & \mathit{undef} & \mathit{handler} & = & \mathit{undef} \\ \mathit{pass} & = & \mathit{undef} & \mathit{target} & = & \mathit{undef} \end{array}$$

We can now summarize the overall behavior of the exception mechanism, which is analyzed in detail in the following sections, by saying that if there is a handler in the frame defined by *stackCursor*, then the exception mechanism will try to find (when *StackWalk*ing) or to execute (when *Unwind*ing) or to leave (when *Leave*ing) the corresponding handler; otherwise it will continue its work in the caller's frame, or it ends its *Leave* pass at the *target*.

4.2.3.1 The *StackWalk* Pass

During a *StackWalk* pass, the exception mechanism EXCCLR starts in the current *frame* to search for a suitable handler of the current exception in this frame. Such a handler exists if the search position n in the current *frame* has not yet reached the last element of the handlers array *excHA* of *frame*'s method *mref*. Formally, there are no more handlers to inspect in a frame pointed to by a *stackCursor* of the form $((\_,\_,\_,\_,\_,\mathit{mref}),n)$ if the below defined predicate evaluates to false.

$$\mathit{existsHanWithinFrame}((\_,\_,\_,\_,\_,\mathit{mref}),n) :\Leftrightarrow n < \mathit{length}(\mathit{excHA}(\mathit{mref}))$$

If there are no (more) handlers in the frame pointed to by *stackCursor*, then the search has to be continued at the caller frame. This means to reset the *stackCursor* to point to the caller frame and the index 0 in its exception handling array.

```
SEARCHINVFRAME($fr$) $\equiv$
  let $\_ \cdot [fr', fr] \cdot \_ = \mathit{frameStack} \cdot [\mathit{frame}]$ in
    RESET($\mathit{stackCursor}, fr'$)
```

There are three groups of possible handlers h the exception mechanism is looking for in a given frame during its *StackWalk*:

- a `catch` handler whose `try` block protects the program counter *pc* of the frame pointed at by *stackCursor* and whose type *typeExc* is a supertype of the exception type (if the ”wrapping” is turned on) or of the wrapped exception type (if the ”wrapping” is turned off). The handlers of this type are identified through the predicate *matchCatch*.

$$\boxed{matchCatch(pos, T, h) :\Leftrightarrow isInTry(pos, h) \ \wedge clauseKind(h) = \texttt{catch} \wedge T \preceq typeExc(h)}$$

- a `filter` handler whose `try` block protects the *pc* of the frame pointed at by the *stackCursor*. Such handlers are determined with the predicate *matchFilter*.

$$\boxed{matchFilter(pos, h) :\Leftrightarrow isInTry(pos, h) \wedge clauseKind(h) = \texttt{filter}}$$

- a `filter` handler whose `filter` region contains the *pc* of the frame pointed at by the *stackCursor* (these handlers are selected through the predicate *isInFilter* defined in Section 4.2.2). This corresponds to an outer exception and will be described in more detail below.

The order of the **if** clauses in the **let** statement from the rule *StackWalk* in Figure 4.2.3.1 is not important. This is justified by the fact that no handler can be of all the above types.

Lemma 4.2.2 *Let T be a type. The predicates matchCatch*^{T}*, matchFilter, and isInFilter are pairwise disjoint*[19].

Proof. Using the definitions of the three predicates and Lemma 4.2.1. □

The *handler pointed to by a stackCursor* of the form $((\_, \_, \_, \_, \_, mref), n)$, is defined to be $excHA(mref)(n)$. If this handler is not of any of the above types, then the *stackCursor* is incremented to point to the next handler in the *excHA* of the method *mref*.

The ***Ordering Assumption*** in Section 4.2.2 and the lexical nesting constraints stated in [45, Partition I, §12.4.2.7] ensure that if *stackCursor* points to a handler of one of the above types, then this handler is the first handler in the exception handling array (starting at the position indicated in the *stackCursor*) of any of the above types.

Handler Case 1 If the handler pointed to by the *stackCursor* is a matching `catch`, then this handler becomes the *handler* to handle the exception in the pass *Unwind*. The *stackCursor* is reset to be reused for the *Unwind* pass: It should point to the faulting frame, *i.e.*, the current *frame*. Note that during *StackWalk*, *frame* always points to the faulting frame except in case a `filter` region is executed. However, the frame built to execute a `filter` is never searched for a handler corresponding to the current exception.

```
FOUNDHANDLER ≡
  pass    := Unwind
  handler := stackCursor
```

[19]By *matchCatch*^{T} we understand the predicate defined by the set $\{(pos, h) \mid matchCatch(pos,T,h)\}$.

Figure 4.14 The exception handling mechanism EXCCLR.

```
EXCCLR ≡ match $pass$
  $StackWalk$ → if $existsHanWithinFrame(stackCursor)$ then
                  let $h = hanWithinFrame(stackCursor)$ in
                    let $T =$ if $wrapNonExc$ then $actualTypeOf(exc)$
                              else $actualTypeOf(wrappedExc(exc))$ in
                      if $matchCatch(pos, T, h)$ then
                        FOUNDHANDLER
                        RESET$(stackCursor, frame)$
                      elseif $matchFilter(pos, h)$ then EXECFILTER$(h)$
                      elseif $isInFilter(pos, h)$ then EXITINNEREXC
                      else GOTONXTHAN
                else  SEARCHINVFRAME$(fr)$
                where $stackCursor = (fr, \_)$ and $fr = (pos, \_, \_, \_, \_, \_)$

  $Unwind$    → if $existsHanWithinFrame(stackCursor)$ then
                  let $h = hanWithinFrame(stackCursor)$ in
                    if $matchTargetHan(handler, stackCursor)$ then
                      EXECHAN$(h)$
                    elseif $matchFinFault(pc, h)$ then
                      EXECHAN$(h)$
                      GOTONXTHAN
                    elseif $isInHandler(pc, h)$ then
                      ABORTPREVPASSREC
                      GOTONXTHAN
                    elseif $isInFilter(pc, h)$ then
                      CONTINUEOUTEREXC
                    else GOTONXTHAN
                else
                  POPFRAME$(0, [\,])$
                  SEARCHINVFRAME$(frame)$

  $Leave$     → if $existsHanWithinFrame(stackCursor)$ then
                  let $h = hanWithinFrame(stackCursor)$ in
                    if $isFinFromTo(h, pc, target)$ then
                      EXECHAN$(h)$
                    if $isRealHanFromTo(h, pc, target)$ then
                      ABORTPREVPASSREC
                    GOTONXTHAN
                else
                  $pc := target$
                  $evalStack := [\,]$
                  POPREC
                  $switch := Noswitch$
```

```
RESET$(s, fr)$ ≡
  $s := (fr, 0)$
```

Handler Case 2 If the handler is a `filter`, then its `filter` region is executed. The execution is performed in a separate frame constructed especially for this purpose (detail omitted by [45]). The current *frame* becomes the frame for executing the `filter` region. The faulting frame is pushed onto the *frameStack*. The current frame points now to the method, local variables, local memory pool, and arguments of the frame in which *stackCursor* is, it has the exception reference on the *evalStack* (if the ”wrapping” is turned on) or the wrapped exception $wrappedExc(exc)$ (if the ”wrapping” is turned off), and the program counter *pc* set to the beginning *filterStart* of the `filter` region. Control is switched to normal execution mode by setting *switch* to *Noswitch*.

```
EXECFILTER(h) ≡
    pc         := filterStart(h)
    evalStack  := if wrapNonExc then [exc] else wrappedExc(exc)
    locAdr     := locAdr'
    locPool    := locPool'
    argAdr     := argAdr'
    meth       := meth'
    frameStack := frameStack · [frame]
    switch     := Noswitch
    where stackCursor = ((_, locAdr', locPool', argAdr', _, meth'), _)
```

Handler Case 3 The *stackCursor* points to a `filter` handler whose `filter` region contains the *pc* of the frame pointed at by *stackCursor*.

Exceptions in `filter` region? It is not clearly documented in [45] what happens if an (*inner*) exception is thrown while executing the `filter` region during the *StackWalk* pass of an *outer* exception. The following cases are to be considered:

- if the exception *is taken care of* in the `filter` region, *i.e.*, it is successfully handled by a `catch`/`filter` handler or it is aborted because it occurred in yet another `filter` region of a nested handler (see the second case below), then the given `filter` region continues executing normally (after the exception has been taken care of);
- if the exception *is not taken care of* in the `filter` region, then the exception will be discarded (via the CONTINUEOUTEREXC macro defined in Section 4.2.3.2) after its `finally` and `fault` handlers have been executed. Therefore, in this case EXCCLR exits via the macro EXITINNEREXC the *StackWalk* pass and starts an *Unwind* pass, during which all the `finally`/`fault` handlers for the inner exception are executed until the `filter` region where the inner exception occurred is reached.

```
EXITINNEREXC ≡
    pass := Unwind
    RESET(stackCursor, frame)
```

Example 4.2.2 *Consider that the* `entrypoint` *has the body in Figure 4.15. According to the* ***Ordering Assumption*** *stated in Section 4.2.2, the stackCursor points initially to the* `filter` *handler. To decide whether this handler is appropriate to handle the* `NullReferenceException` *thrown at* 1*, the* `filter` *region is executed in a separate frame. So, when the instruction* 2 *is executed, the frameStack is consisting of*

Figure 4.15 Example: discarding exception and leave records in the `entrypoint` method.

```
.try
{
 .try
 {
  .try
  {
   0: newobj instance void NullReferenceException::.ctor()
   1: throw
  }
  filter
  {
   2: newobj instance void InvalidCastException::.ctor()
   3: throw
   4: ldc.i4  1
   5: endfilter
  }
  {
   6: pop
   7: leave 12
  }
 }
 catch NullReferenceException
 {
  8: pop
  9: leave 12
 }
}
finally
{
 10: newobj instance void DivideByZeroException::.ctor()
 11: throw
}
12: ret
```

the frame associated to the `entrypoint`*. The* `filter` *region throws another exception. The context of the just handled outer exception is saved in the passRecStack, and the* `InvalidCastException` *raised at* 3 *becomes the current (inner) exception. However, the StackWalk pass of the inner exception is immediately exited since the inner exception cannot be propagated out of the* `filter` *region. The Unwind pass for the inner exception is then started.*

4.2.3.2 The *Unwind* Pass

As soon as the pass *StackWalk* terminates, the exception mechanism starts the *Unwind* pass with the *stackCursor* pointing to the faulting frame. Starting there, one has to walk down to the *handler* determined in the *StackWalk* pass, executing on the way every `finally`/`fault` handler region. This also happens in case *handler* is *undef*. The handlers of interest for the exception mechanism in the *Unwind* pass are:

- the matching target handler, *i.e.*, the *handler* determined at the end of the *StackWalk* pass (if any[20]). For that to happen, the two *handler* and *stackCursor* frames in question have to coincide. The predicate *matchTargetHan* decides whether the handler pointed to by the *stackCursor* is *handler*. We say that two frames are the same if the address arrays of their local variables and arguments as well as their method names coincide. This is defined by means of the predicate *sameFrame*.

$$matchTargetHan((fr, n), (fr', n')) :\Leftrightarrow sameFrame(fr, fr') \land n = n'$$

$$\begin{aligned}&sameFrame((\_, locAdr', \_, argAdr', \_, meth'), (\_, locAdr'', \_, argAdr'', \_, meth'')) :\Leftrightarrow\\ &\quad locAdr' = locAdr'' \land argAdr' = argAdr'' \land meth' = meth''\end{aligned}$$

- a matching `finally`/`fault` handler whose associated `try` block protects the *pc*. The predicate *matchFinFault* selects the `finally`/`fault` handlers whose `try` blocks protect a given code index.

$$matchFinFault(pos, h) :\Leftrightarrow isInTry(pos, h) \land clauseKind(h) \in \{\texttt{finally}, \texttt{fault}\}$$

- a handler whose handler region contains *pc*: Such handlers are selected with the predicate *isInHandler* defined in Section 4.2.2;

- a `filter` handler whose `filter` region contains *pc*: These handlers are identified through the predicate *isInFilter* defined in Section 4.2.2;

The order of the last three **if** clauses in the **let** statement of the rule *Unwind* in Figure 4.2.3.1 is not important since no handler can be of all of the above last three types. It only matters that the first clause is guarded by *matchTargetHan*.

Lemma 4.2.3 *The predicates matchFinFault, isInHandler and isInFilter are pairwise disjoint.*

Proof. Using the definitions of the predicates and Lemma 4.2.1. □

The ***Ordering Assumption*** in Section 4.2.2 and the lexical nesting constraints given in [45, Partition I, §12.4.2.7] ensure that if the *stackCursor* points to a handler of one of the above types, then this handler is the first handler in the exception handling array (starting at the position indicated in the *stackCursor*) of any of the above types.

Handler Case 1 If the handler pointed to by *stackCursor* is the *handler* found in the *StackWalk*, its handler region is executed through EXECHAN: The *pc* is set to the beginning of the handler region, the exception reference *exc* (if the "wrapping" is turned on) or the wrapped exception *wrappedExc*(*exc*) (if the "wrapping" is turned off) is loaded on the *evalStack* (when EXECHAN is applied for executing `finally`/`fault` handler regions, nothing is pushed onto *evalStack*), and control switches to normal execution mode, *i.e.*, *switch* is set to *Noswitch*.

[20]Note that *handler* can be *undef* if the search in the *StackWalk* has been exited because the exception was thrown in a `filter` region.

```
EXECHAN($h$) $\equiv$
    $pc := handlerStart(h)$
    $evalStack :=$ if $clauseKind(h) \in \{$catch, filter$\}$ then
                        if $wrapNonExc$ then $[exc]$ else $wrappedExc(exc)$
                   else $[\,]$
    $switch := Noswitch$
```

Handler Case 2 If the handler pointed to by the *stackCursor* is a matching finally/fault handler, its handler region is executed with an initially empty *evalStack*. At the same time, the *stackCursor* is incremented through GOTONXTHAN to point to the next handler in *excHA*.

Handler Case 3 Let us assume that the handler pointed to by *stackCursor* is an arbitrary handler whose handler region contains *pc*.

Exceptions in handler region? The ECMA standard [45] does not specify what should happen if an exception is raised in a handler region. The experimentation in [54] led to the following rules of thumb for exceptions thrown in a handler region, in a way similar to the case of nested exceptions in filter code:

- If the exception *is taken care of* in the handler region, *i.e.*, it is successfully handled by a catch/filter handler or it is discarded (because it occurred in a filter region of a nested handler), then the handler region continues executing normally (after the exception is taken care of).
- If the exception *is not taken care of* in the handler region, i.e, it escapes the handler region, then
  - the previous pass of the exception mechanism is aborted through the macro ABORTPREVPASSREC by popping its record from the *passRecStack*

```
ABORTPREVPASSREC $\equiv$
    $passRecStack := pop(passRecStack)$
```

  - the exception is propagated further via GOTONXTHAN, *i.e.*, the *Unwind* pass continues by setting the *stackCursor* to the next handler in *excHA*.

Remark 4.2.1 *An exception can go "unhandled" without taking down the process, namely if an outer exception goes unhandled, but an inner exception is successfully handled. In fact, the execution of a handler region can only occur when the exception mechanism* EXCCLR *runs in the Unwind and Leave passes: In Unwind, handler regions of any kind are executed, whereas in Leave, only* finally *handler regions are executed. If the raised exception occurred while* EXCCLR *runs an Unwind pass for handling an outer exception, the Unwind pass of the outer exception is stopped, and the corresponding pass record is popped from the passRecStack. If the exception has been thrown while the exception mechanism runs a Leave pass for executing* finally *handlers "on the way" from a Leave instruction to its target, then this pass is stopped, and its associated pass record is popped from the passRecStack.*

Handler Case 4 The handler pointed to by the *stackCursor* is a `filter` handler whose `filter` region contains *pc*. Then the execution of this `filter` region must have triggered an inner exception whose *StackWalk* led to a call of EXITINNEREXC. In this case, the current (inner) exception is aborted, and the `filter` considered as not providing a handler for the outer exception. Formally, CONTINUEOUTEREXC pops the frame built for executing the `filter` region, pops from the *passRecStack* the pass record corresponding to the inner exception and reestablishes the pass context of the outer exception, but with the *stackCursor* pointing to the handler following the just inspected `filter` handler. The updates of the *stackCursor* in POPREC and GOTONXTHAN are done **seq**uentially such that the update in GOTONXTHAN overwrites the update in the macro POPREC. Note that by these stipulations, there is no way to exit a `filter` region with an exception. This ensures that the frame built by EXECFILTER for executing a `filter` region is used only for this purpose.

```
CONTINUEOUTEREXC ≡
  POPFRAME(0, [ ])
  POPREC seq GOTONXTHAN
```

```
POPREC ≡
  if $passRecStack = [\,]$ then
    SETRECUNDEF
    $switch := Noswitch$
  else let $(passRecStack', [rec]) = split(passRecStack, 1)$ in
    if $rec \in ExcRec$ then
      let $(exc', pass', stackCursor', handler') = rec$ in
        $exc$         $:= exc'$
        $pass$        $:= pass'$
        $stackCursor$ $:= stackCursor'$
        $handler$     $:= handler'$
    if $rec \in LeaveRec$ then
      let $(pass', stackCursor', target') = rec$ in
        $pass$        $:= pass'$
        $stackCursor$ $:= stackCursor'$
        $target$      $:= target'$
    $passRecStack := passRecStack'$
```

```
SETRECUNDEF ≡
  $exc$         $:= undef$
  $pass$        $:= undef$
  $stackCursor$ $:= undef$
  $target$      $:= undef$
  $handler$     $:= undef$
```

Remark 4.2.2 *The execution of* POPFRAME *is* safe *since the frameStack cannot be empty at the time when* CONTINUEOUTEREXC *is fired.* [45, Partition II, §12.4.2.8.1] *states that the control* *can be transferred to a* `filter` *region only through* EXCCLR. *Since pc is in a* `filter` *region, an* EXECFILTER *should have been already executed. But in this case, a new frame is pushed on the frameStack. Hence, frameStack is not empty when* CONTINUEOUTEREXC *is executed.*

Example 4.2.3 *Consider again the* `entrypoint` *body listed in Figure 4.15. Since the* `InvalidCastException` *raised in the* `filter` *region is not taken care of in the* `filter` *region, the exception, its Unwind pass, and the frame built to run the* `filter` *region are discarded. The* `NullReferenceException` *becomes again the current exception, and its StackWalk pass is continued by incrementing its stackCursor. This will point to the* `catch` *handler which turns out to be suitable for handling the exception. The StackWalk pass ends by setting handler to point to the* `catch` *handler. The Unwind pass starts, and the stackCursor is reset to point to the most deeply nested handler, i.e., the* `filter` *handler. The stackCursor is then incremented until it points to a handler of interest for the Unwind pass. The first such handler is exactly handler whose handler region is then executed.*

If the handler pointed to by the *stackCursor* is not of interest for the *Unwind* pass, the *stackCursor* is incremented to point to the next handler in *excHA*.

If the *Unwind* pass terminated the inspection of all the handlers in the frame indicated by the *stackCursor* given in the form $(frame, n)$, *i.e.*, the predicate $existsHanWithinFrame((frame, n))$ evaluates to false, then the current *frame* is popped from the *frameStack*, and the *Unwind* pass continues in the caller frame of the current *frame*. This continuation is accomplished by setting the *stackCursor* to point to the new current *frame* and the index 0 in the *excHA* of this frame.

Remark 4.2.3 *As long as the exception mechanism is running an Unwind pass, the current stackCursor is always pointing to the current frame. This contrasts with the StackWalk pass where the current stackCursor is usually pointing to a different frame in the frameStack. This remark is proved in Lemma 4.2.4.*

Lemma 4.2.4 *Let* $stackCursor = (fr, \_)$ *be the current stackCursor. If* $pass = Unwind$, *then fr is the current frame.*

Proof. By a simple induction on the run of the exception handling mechanism. □

Remark 4.2.4 *The execution of the macros* POPFRAME *and* SEARCHINVFRAME *in the* ***else*** *clause is* safe *since the current frame has a caller frame, i.e., the current frame cannot be the frame of the* `entrypoint`. *This is because* ***Backstop entry*** *guarantees that the* ***else*** *clause is not reachable if the current frame is the frame of the* `entrypoint`. *The same argument can also be invoked in case of* SEARCHINVFRAME *in the StackWalk pass.*

4.2.3.3 The *Leave* Pass

The exception mechanism gets into the *Leave* pass when, in the normal execution mode, the virtual machine executes a *Leave* instruction, which by the *Leave* constraints (see Section 4.2.2) can only happen upon the normal termination of a `try` block or of a `catch`/`filter` handler region. One has to execute the handler regions of all `finally` handlers "on the way" from the *Leave* instruction to the instruction whose program counter is given by the *Leave target* parameter. The *stackCursor* used in the *Leave* pass is initialized by the *Leave* instruction (see Figure 4.16). In the *Leave* pass, the exception mechanism EXCCLR searches for

- `finally` handlers that are "on the way" from the *pc* to the *target*, and
- *real* handlers, *i.e.*, `catch`/`filter` handlers whose handler regions are "exited on the way" from the *pc* to the *target* – more details are given below.

Handler Case 1 If the handler pointed to by *stackCursor* is a `finally` handler *h* on the way from *pc* to the *target* of the current *Leave* pass record, then the handler region of *h* is executed (see the first clause of the *Leave* rule in Figure 4.16): *pc* is set to $handlerStart(h)$, *evalStack* to $[\,]$, and control switches to normal execution mode, *i.e.*, *switch* is set to *Noswitch*. We define the predicate *isFinFromTo* to identify the `finally` handlers which are "on the way" from a code index to another code index.

$$\begin{aligned} isFinFromTo(h, pos, pos') \quad :\Leftrightarrow \quad & clauseKind(h) = \texttt{finally} \land \\ & isInTry(pos, h) \land \\ & \neg\, isInTry(pos', h) \land \neg\, isInHandler(pos', h) \end{aligned}$$

Handler Case 2 If the *stackCursor* points to a `catch`/`filter` handler whose handler region is "exited on the way" from *pc* to the *target*, then the previous pass record, *i.e.*, the top pass record on *passRecStack*, is popped off (see the second clause of the *Leave* rule). The discarded record can only be referring to an *Unwind* pass for handling an exception. By discarding this record, the mechanism terminates the handling of the corresponding exception. The below defined predicate *isRealHanFromTo* is used to determine the `catch`/`filter` handlers whose handler regions are "exited on the way" from a code index to another code index.

$$\begin{aligned} isRealHanFromTo(h, pos, pos') \quad :\Leftrightarrow \quad & clauseKind(h) \in \{\texttt{catch}, \texttt{filter}\} \land \\ & isInHandler(pos, h) \land \neg\, isInHandler(pos', h) \end{aligned}$$

That the discarded record can only be referring to an *Unwind* pass is proved in the following lemma:

Lemma 4.2.5 *If pass* $=$ *Leave and the current stackCursor points to a* `catch`/`filter` *handler, then* $top(passRecStack) = (\_, Unwind, \_, h)$ *such that* $matchTarget(stackCursor, h)$.

Proof. The proof proceeds by induction on the run of the exception mechanism and takes advantage of the fact that a `catch`/`filter` handler region can only be entered for handling an exception, *i.e.*, in an *Unwind* pass. □

Although the two **if** clauses in the **let** statement of the *Leave* pass in Figure 4.16 are executed in parallel, it is never the case that the embedded macros EXECHAN and ABORTPREVPASSREC are simultaneously executed. The reason is that no exception handler can be of both of the above types.

Lemma 4.2.6 *The predicates isFinFromTo and isRealHanFromTo are disjoint.*

Instruction	Informal description
Throw	Pops the top stack element, and throws it as an exception.
EndFilter	Terminates the execution of a `filter` region.
EndFinally	Terminates the execution of a `finally`/`fault` handler region.
Leave(*target*)	Exits a `try` block or a `catch`/`filter` handler region.

Table 4.15: The considered exception specific CIL instructions.

Proof. Using the definitions of the predicates. □

For each kind of handler, the exception mechanism EXCCLR also inspects the next handler in *excHA*. When the handlers in the current method are exhausted, by the *Leave* constraints this round of EXCCLR is terminated, and the execution proceeds at *target*: *pc* is set to *target*, the context of the previous pass record, *i.e.*, the top pass record on *passRecStack*, is reestablished, and control is passed to normal execution mode by setting *switch* to *Noswitch* (see Figure 4.16).

Example 4.2.4 *Consider again the* `entrypoint` *body in Figure 4.15. Upon executing the* `leave` *instruction at* 9*, the mechanism enters into a Leave pass with target* $= 12$ *and stackCursor initially pointing to the* `filter` *handler. The stackCursor is successively incremented until it points to the* `catch` *handler whose handler region is "exited on the way" from pc* $= 9$ *to target* $= 12$*. Therefore, the top pass record of the passRecStack, i.e., the record associated to the* `NullReferenceException` *in its Unwind pass, is popped off. The stackCursor is then incremented and points to the* `finally` *handler. As the* `finally` *handler region raises an exception, the Leave pass is aborted.*

4.2.3.4 The Semantics of the Exception Specific Instructions

This section defines the operational semantics model $\mathrm{CLR}_{\mathcal{E}}$ which refines $\mathrm{CLR}_{\mathcal{L}}$ (given in Section 4.1.3.1) to cover the changes that are needed to model exception handling. In particular, this section defines the operational semantics rules for the exception specific instructions. Table 4.15 contains the informal definitions of these instructions whose precise semantics is defined in terms of the semantics rules in Figure 4.16.

The execution scheme for $\mathrm{CLR}_{\mathcal{E}}$ is the one for $\mathrm{CLR}_{\mathcal{L}}$ extended with the exception mechanism EXCCLR.

$$\mathrm{CLR}_{\mathcal{E}} \equiv \mathit{execScheme}(\mathrm{EXECCLR}, \mathrm{SWITCHCLR}, \mathrm{EXCCLR})$$

```
$execScheme \equiv$
  if $switch = ExcMech$ then
    EXCCLR
  elseif $switch = Noswitch$ then
    EXECCLR$(code(pc))$
  else SWITCHCLR
```

Figure 4.16 The operational semantics rules for the exception specific instructions.

```
$\text{EXECCLR}(instr) \equiv$ match $instr$
  ⋮
  $Throw$        → let $ref = top(evalStack)$ in
                     if $ref \neq$ null then
                       let $ref' =$ if $actualTypeOf(ref) \preceq$ Exception then $ref$
                                    else $NewWrapExc(ref)$ in
                         $\text{LOADREC}((ref', StackWalk, (frame, 0), undef))$
                         $switch := ExcMech$
                     else $\text{RAISE}($NullReferenceException$)$

  $EndFilter$    → let $val = top(evalStack)$ in
                     if $val = 1$ then
                       $\text{FOUNDHANDLER}$
                       $\text{RESET}(stackCursor, top(frameStack))$
                     else $\text{GOTONXTHAN}$
                     $\text{POPFRAME}(0, [\,])$
                     $switch := ExcMech$

  $EndFinally$   → $switch := ExcMech$

  $Leave(target)$ → $\text{LOADREC}((Leave, (frame, 0), target))$
                    $switch := ExcMech$
```

Each of the instructions gives control to the exception mechanism EXCCLR by appropriately setting *switch* to *ExcMech*.

The *Throw* instruction pops the topmost *evalStack* element (see Remark 4.2.6 below), which is supposed to be an exception reference. It loads with the macro LOADREC to the exception mechanism the pass record associated to the given exception: *exc* is set to the exception reference (if *exc*'s type is derived from `Exception`) or to a `RuntimeWrappedException` reference pointing to *exc* (if *exc*'s type is *not* derived from `Exception`), *pass* to *StackWalk*, the *stackCursor* is initialized with $(frame, 0)$, and *handler* is set to *undef*.

In order to create a new reference to a `RuntimeWrappedException` object pointing to a given object reference, we define the macro *NewWrapExc*. Thus, given an exception reference *ref*, the macro *NewWrapExc*(*ref*) creates a new `RuntimeWrappedException` object, with the non-`Exception` as an instance field, and returns a reference to the object.

```
let $ref' = NewWrapExc(ref)$ in $P \equiv$
  let $ref' = new(ObjRef,$ RuntimeWrappedException$)$ in
    $initState(ref') := Init($RuntimeWrappedException$)$
    $mem(fieldAdr(ref',$ RuntimeWrappedException::WrappedException$)) := ref$
  seq $P$
```

If the exception mechanism is already working in a pass, *i.e.*, *pass* is not *undef*, then the current pass record is pushed with PUSHREC onto *passRecStack*.

```
LOADREC($rec$) $\equiv$
  if $rec \in ExcRec$ then
    let $(exc', pass', stackCursor', handler') = rec$ in
      $exc$         $:= exc'$
      $pass$        $:= pass'$
      $stackCursor$ $:= stackCursor'$
      $handler$     $:= handler'$
  else let $(pass', stackCursor', target') = rec$ in
    $pass$        $:= pass'$
    $stackCursor$ $:= stackCursor'$
    $target$      $:= target'$
  if $pass \neq undef$ then PUSHREC
```

```
PUSHREC $\equiv$
  if $pass = Leave$ then
    $passRecStack := push(passRecStack, (pass, stackCursor, target))$
  else $passRecStack := push(passRecStack, (exc, pass, stackCursor, handler))$
```

If the exception reference found on top of the *evalStack* by the *Throw* instruction is `null`, a `NullReferenceException` is thrown. Given a class C, the macro RAISE(C) is defined by the following code template:

```
RAISE($C$) $\equiv$
  $NewObj(C$::.ctor$)$
  $Throw$
```

This macro can be viewed as a static method defined in class `object`. Calling the macro is then like invoking the corresponding method.

In a `filter` region, exactly one *EndFilter* is allowed, namely its last instruction, which is supposed to be the only one used to normally exit the `filter` region. The *EndFilter* instruction takes an integer *val* from the stack that is supposed to be either 0 or 1.

Remark 4.2.5 *In* [45], 0 *and* 1 *are assimilated with "continue search" and "execute handler", respectively. There is a discrepancy between* [45, Partition I, §12.4.2.5]*, which states* Execution cannot be resumed at the location of the exception, except with a user-filtered handler *– therefore a "resume exception" value in addition to* 0 *and* 1 *is foreseen allowing CLR to resume the execution at the point where the handled exception has been raised— and* [45, Partition III, §3.34] *which states that the only possible return values from the filter are "exception_continue_search"*(0) *and "exception_execute_handler"*(1)*.*

If *val* is 1, then the `filter` handler to which *EndFilter* corresponds becomes the *handler* to handle the current exception in the pass *Unwind*. Remember that the `filter` handler is the handler pointed to by the *stackCursor*. The *stackCursor* is reset to be used for the pass *Unwind*:

It will point to the topmost frame on *frameStack* which is actually the faulting frame. If *val* is 0, the *stackCursor* is incremented to point to the handler following our `filter` handler. Independently of *val*, the current *frame* is discarded to reestablish the context of the faulting frame. Note that we do not explicitly pop *val* from the *evalStack* since the global dynamic function *evalStack* is updated anyway in the next step (through POPFRAME) to the *evalStack*$'$ of the faulting frame.

The *EndFinally* instruction terminates (normally) the execution of the handler region of a `finally` or of a `fault` handler. It simply transfers control to the exception mechanism EXCCLR. A *Leave* instruction loads to the exception mechanism a pass record corresponding to a *Leave* pass which has the *target* set to the instruction's code index token argument and the *stackCursor* to $(frame, 0)$.

Remark 4.2.6 *The reader might ask why the instructions Throw and EndFilter do not set the evalStack. The reason is that this set up, i.e., the emptying of evalStack, is supposed to be either a* side-effect *(the case of the Throw instruction) or ensured for a* correct *CIL (the case of the EndFilter instruction). Thus, the Throw instruction passes control to* EXCCLR *which, in a next step, will execute*[21] *a* `catch`/`finally`/`fault` *handler region or a* `filter` *code or will propagate the exception in another frame. All these "events" will "clear" the evalStack. In case of EndFilter, the evalStack must contain exactly one item (an* `int32` *which is popped off by EndFilter). Note that this has to be checked by the bytecode verification (see Figure 4.18) and not ensured by the exception handling mechanism.*

CLR$_\mathcal{E}$ also extends the rules where the instructions considered for CLR$_\mathcal{L}$ might throw exceptions. The extension is defined through the semantics rules in Figure 4.17.

The rule for the *Execute* instruction is extended with the three cases when the instruction raises one of the exceptions `DivideByZeroException`, `OverflowException`, and `ArithmeticException`. A `NullReferenceException` is raised if the target reference of a *Call*, *LoadField*, *StoreField*, *CallVirt*, *LoadFieldA*[22]. Accordingly, the macro VIRTCALL defined in Section 4.1.3.1 should be refined.

```
VIRTCALL$(tail, T::M, [ref] \cdot vals) \equiv$
  if $ref \neq$ null then
    let $T'::M = lookUp(actualTypeOf\ (ref), T::M)$ in
      if $actualTypeOf\ (ref) = T' \in ValueType$ then
        $switch := Invoke(tail, T'::M, [addressOf\ (ref)] \cdot vals)$
      else $switch := Invoke(tail, T'::M, [ref] \cdot vals)$
  else RAISE(NullReferenceException)
```

A `NullReferenceException` is raised also if the pointer popped from the *evalStack* by a *LoadInd* or *StoreInd* instruction points to an invalid address, or if the to be unboxed object of an *Unbox* instruction is the `null` reference. The *Unbox* instruction throws

[21] One can formally prove that there is such a "step" in the further run of the EXCCLR.

[22] Some versions of the CLR implementation, *e.g.*, Microsoft .NET Framework (v2.0) [3], are lazy and do not throw a `NullReferenceException` if one of the instructions *LoadFieldA*, *Call* or *CallVirt* receives `null` as the target reference, when expecting a managed pointer, as opposed to an object reference. The exception is only raised at the first attempt to dereference the managed pointer. However, being defined according to [45], our semantics model contrasts with these lazy implementations.

an `InvalidCastException` if the object on the *evalStack* is not a boxed object of the instruction's value type token argument. An *Unbox.Any* instruction with a value type token argument throws a `NullReferenceException` if the boxed object is `null`[23] and an `InvalidCastException` if the boxed object is not a boxed instance of the instruction's type token argument. An exception of the same class is also thrown by a *RefAnyVal* instruction if the instruction's type token argument is not the type stored in the typed reference. If the top stack element cannot be cast to *CastClass*' type token argument, an `InvalidCastException` is thrown.

4.2.4 The Bytecode Verification

This section is about how the exception handling influences the bytecode verification. The exception specific conditions that should be ensured by the bytecode in order to achieve type safety are depicted in Section 4.2.4.1. The type consistency checks performed for the exception specific instructions are defined in Section 4.2.4.2. The simulation of the execution paths is extended in Section 4.2.4.3 to include the exception handling.

4.2.4.1 The Bytecode Structure

In addition to the constraints stated in Section 4.1.4.2, the bytecode verification requires also that following condition holds:

Tail Call in Handlers	A tail method call should not be used to transfer control out of a `try` block, `filter` region, `catch`/`finally`/`filter`/`fault` handler region.

The reason for requiring ***Tail Call in Handlers*** is the same as that for ***Tail Call Pattern*** in Section 4.1.4.2. If the prefix `tail` is ignored when the bytecode is jitted, the caller frame would continue to execute after the callee's frame returns. Because of ***Tail Call in Handlers***, the caller frame would not matter for the exception mechanism if the callee's frame returned with an exception. Although ***Tail Call in Handlers*** is not relevant in the type safety proof, we mention it here for completeness only.

4.2.4.2 Verifying the Bytecode Instructions

As mentioned in Section 4.1.4.3, the evaluation stack before an instruction *instr* following an unconditional branch instruction should be empty, unless *instr* is the target of a forward branch instruction. As most of the exception specific instructions are regarded as unconditional branch instruction, the set *UncondBranchInstr* is extended accordingly: $\mathit{UncondBranchInstr} = \{\mathit{Return}, \mathit{Throw}, \mathit{EndFinally}, \mathit{Leave}(\_)\}$.

Bytecode verification also checks that the stack is empty upon entering a `try` block. This check can be expressed as a constraint on the first instruction of any `try` block. For this

[23]The ECMA Standard [45, Partition III, §4.33] states that a `NullReferenceException` is raised, regardless of the instruction's type token argument. This is, however, not true as the instruction with reference type token arguments acts as a *CastClass* instruction, and consequently it throws no exception.

Figure 4.17 The operational semantics rules for the exception-prone cases.

```
$\textsc{execCLR}(instr) \equiv$ match $instr$
    ⋮
  $Execute(op)$            $\rightarrow$ let $vals = take(evalStack, opNo(op))$ in
                               if $divByZeroCase(op, vals)$ then
                                 RAISE(DivideByZeroException)
                               elseif $overflowCase(op, vals)$ then
                                 RAISE(OverflowException)
                               elseif $invalidNrCase(op, vals)$
                                 RAISE(ArithmeticException)
  $Call(\_,\_,T{::}M)$      $\rightarrow$ if $T \neq$ System.Delegate then
                               let $[ref] \cdot \_ = take(evalStack, argNo(T{::}M))$ in
                                 if $ref =$ null then RAISE(NullReferenceException)
  $LoadField(\_,\_)$        $\rightarrow$ let $ref = top(evalStack)$ in
                               if $ref =$ null then RAISE(NullReferenceException)
  $StoreField(\_,\_)$       $\rightarrow$ let $[ref] \cdot \_ = take(evalStack, 2)$ in
                               if $ref =$ null then RAISE(NullReferenceException)
  $CallVirt(\_,\_,T{::}M)$  $\rightarrow$ let $[ref] \cdot \_ = take(evalStack, argNo(T{::}M))$ in
                               if $ref =$ null then RAISE(NullReferenceException)
  $LoadFieldA(\_,\_)$       $\rightarrow$ let $val = top(evalStack)$ in
                               if $val =$ null then RAISE(NullReferenceException)
  $LoadInd(\_)$             $\rightarrow$ let $adr = top(evalStack)$ in
                               if $\neg validAdr(adr)$ then RAISE(NullReferenceException)
  $StoreInd(\_)$            $\rightarrow$ let $[adr] \cdot \_ = take(evalStack, 2)$ in
                               if $\neg validAdr(adr)$ then RAISE(NullReferenceException)
  $Unbox(T)$                $\rightarrow$ let $ref = top(evalStack)$ in
                               if $ref =$ null then
                                 RAISE(NullReferenceException)
                               elseif $T \neq actualTypeOf(ref)$ then
                                 RAISE(InvalidCastException)
  $Unbox.Any(T)$            $\rightarrow$ if $T \in ValueType$ then
                               let $ref = top(evalStack)$ in
                                 if $ref =$ null then RAISE(NullReferenceException)
                                 elseif $T \neq actualTypeOf(ref)$ then
                                   RAISE(InvalidCastException)
  $RefAnyVal(T)$            $\rightarrow$ let $tr = top(evalStack)$ in
                               if $T \neq typedRefType(tr)$ then
                                 RAISE(InvalidCastException)
  $CastClass(T)$            $\rightarrow$ let $ref = top(evalStack)$ in
                               if $ref \neq$ null $\wedge\ actualTypeOf(ref) \npreceq T$ then
                                 RAISE(InvalidCastException)
  $LoadVirtFtn(\_)$         $\rightarrow$ let $ref = top(evalStack)$ in
                               if $ref =$ null then RAISE(NullReferenceException)
```

Figure 4.18 Verifying the exception specific instructions.

$$
\begin{array}{l}
\mathit{check}(\mathit{meth},\mathit{pos},\mathit{argZeroT},\mathit{evalStackT}) :\Leftrightarrow \mathit{enterTry}(\mathit{meth},\mathit{pos},\mathit{evalStackT}) \land \\
\quad \textbf{match } \mathit{code}(\mathit{pos}) \\
\qquad \vdots \\
\qquad \begin{array}{ll}
\mathit{Throw} & \rightarrow \mathit{evalStackT} \sqsubseteq_{\mathit{suf}} [\texttt{object}] \\
\mathit{EndFilter} & \rightarrow \mathit{evalStackT} \sqsubseteq_{\mathit{suf}} [\texttt{int32}] \\
\mathit{EndFinally} & \rightarrow \mathit{True} \\
\mathit{Leave}(\_) & \rightarrow \mathit{True}
\end{array}
\end{array}
$$

purpose, we define the predicate *enterTry*. Given a method *mref*, a code index *pos* and a stack state *evalStackT*, $\mathit{enterTry}(\mathit{mref},\mathit{pos},\mathit{evalStackT})$ is true if *evalStackT* is empty should exist at least one exception handler in the exception handling array of *mref* whose `try` block has the first instruction at *pos*.

$$
\begin{array}{l}
\mathit{enterTry}(\mathit{mref},\mathit{pos},\mathit{evalStackT}) :\Leftrightarrow \\
\quad \exists\, h \in \mathit{excHA}(\mathit{mref}) \;:\; (\mathit{pos} = \mathit{tryStart}(h)) \Rightarrow \mathit{evalStackT} = [\,]
\end{array}
$$

This constraint is not relevant in the type safety proof. Its goal is to simplify the JIT compilation scheme. It could happen that the evaluation of an expression is compiled into a sequence of bytecode instructions which is further jitted into a single CPU instruction. If a `try` block is allowed to be placed in the middle of an expression, the JIT compilation scheme would get complicated, as it would have to break up the expression into protected and non-protected regions. By requiring the stack to be empty upon entering every `try` block, the bytecode verification does not allow a `try` block to be placed in the execution of an expression.

The *check* predicate expressing the type consistency checks is defined in Figure 4.18 also for the exception specific instructions. Moreover, its definition also specifies the above constraint on the first instruction of any `try` block. The *Throw* instruction expects an object reference on the stack. The *EndFilter* instruction which terminates the execution of a `filter` region expects an integer on the stack. For the instructions *EndFinally* and *Leave*, nothing has to be checked.

4.2.4.3 Computing Successor Type States

All the instructions can throw exceptions. Although this assumption does not match the semantics rules defined so far, we assume so, since the special `ExecutionEngineException` may be thrown at any time during the execution of a program. Therefore, for an instruction at *pos*, the exception handlers *h* that protect *pos*, *i.e.*, for which *isInTry*(*pos*,*h*) holds, are added by the function *excHandlers* to the set of successor type states defined through the *succ* function. Upon entering a `catch` handler *h*, the stack state contains the type $\mathit{typeExc}(h)$ of exceptions *h* is handling (if $\mathit{typeExc}(h)$ is a reference type) or the type $\mathit{boxed}(\mathit{typeExc}(h))$ (if $\mathit{typeExc}(h)$ is a value type[24]), whereas upon entering a `filter`, the stack state is [`object`]. Upon entering a `finally` or a `fault` handler, the stack state is $[\,]$. By entering the `filter` region

[24]The ECMA Standard [45] does not mention anything concerning the fact that exceptions of value types are considered boxed inside `catch` handler regions.

Figure 4.19 The type state successors of the exception specific instructions.

```
$succ(meth, pos, argZeroT, evalStackT) = excHandlers(meth, pos, argZeroT) \cup$
  **match** $code(pos)$
      $\vdots$
    $Throw$          $\to \emptyset$
    $EndFilter$      $\to \emptyset$
    $EndFinally$     $\to \{(target, argZeroT, []) \mid target \in LeaveThroughFin(meth, pos)\}$
    $Leave(target)$  $\to$ **if** $\{h \in excHA(meth) \mid isFinFromTo(h, pos, target)\} = \emptyset$ **then**
                         $\{(target, argZeroT, [])\}$
                       **else** $\emptyset$
```

of a `filter` handler h, the successor code index is the index of the first instruction of the `filter` region, *i.e.*, $filterStart(h)$. Otherwise, the successor code index is the index of the first instruction of the handler region of h, *i.e.*, $handlerStart(h)$.

$excHandlers(mref, pos, argZeroT) =$
$\{ (handlerStart(h), argZeroT, [typeExc(h)]) \mid h \in excHA(mref), isInTry(pos, h),$ $clauseKind(h) = \texttt{catch}, typeExc(h) \in RefType\} \cup$

$\{ (handlerStart(h), argZeroT, [boxed(typeExc(h))]) \mid h \in excHA(mref),$ $isInTry(pos, h), clauseKind(h) = \texttt{catch}, typeExc(h) \in ValueType\} \cup$

$\{ (filterStart(h), argZeroT, [\texttt{object}]), (handlerStart(h), argZeroT, [\texttt{object}])$ $\mid h \in excHA(mref)$ such that $isInTry(pos, h), clauseKind(h) = \texttt{filter}\} \cup$

$\{ (handlerStart(h), argZeroT, []) \mid h \in excHA(mref), isInTry(pos, h), clauseKind(h) \in$ $\{\texttt{finally}, \texttt{fault}\}\}$

Example 4.2.5 *Consider the skeleton bytecode of Figure 4.20. To understand the definition of excHandlers, let us look in detail at the successors via the exception handling array of the instruction at the code index* 6*. The* possible *successors are:*

- *the instruction at* 9*, i.e., the first instruction of the* `catch` *handler region: At run-time, this handler region might be executed in the Unwind pass if an exception of the type* `NullReferenceException` *is thrown in the associated* `try` *block;*
- *the instruction at* 17*, i.e., the first instruction of the* `fault` *handler region: At run-time, this handler region might be executed in the Unwind pass if the exception is not of the type* `NullReferenceException`*;*
- *the instruction at* 24*, i.e., the first instruction of the* `filter` *region: At run-time, this region might be executed in the StackWalk pass if the exception is not of the type* `NullReferenceException`*;*
- *the instruction at* 27*, i.e., the first instruction of the* `filter` *handler region: At run-time, this handler region might be executed in the Unwind pass if this* `filter` *handler is suitable to handle the exception;*

- *the instruction at* 31*, i.e., the first instruction of the* `finally` *handler region: At run-time, this handler region might be executed in the Unwind pass if the exception is not of the type* `NullReferenceException` *and the* `filter` *handler is not suitable for handling the exception;*

Note that the code indices 27 *and* 31 *cannot be reached at run-time* directly *from* 6*. The reason is that, the exception handling mechanism has to execute the* `fault` *handler region before* 27 *and* 31 *are reached. However, to simplify the definition of excHandlers, we consider a conservative definition by including more successors than possibly reachable at run-time. This conservative approach does not affect the set of verifiable methods.*

Apart from the successors defined by the function *excHandlers*, the instructions *Throw* and *EndFilter* have no other successors. The *EndFinally* instruction terminates the execution of a `finally`/`fault` handler region. If the associated `try` block was exited with a *Leave*(*target*) instruction, control is transferred to *target* if the *EndFinally* instruction is the last "on the way" from the *Leave* instruction to its *target*. The possible targets are defined by the set *LeaveThroughFin*. A code index *target* is an element of the set *LeaveThroughFin*(*meth*, *pos*) if the instruction *code*(*pos*) is an *EndFinally* instruction of the last `finally` handler "on the way" from an instruction *Leave*(*target*) embedded into the `try` block associated to the *EndFinally* instruction to the instruction *code*(*target*). By this definition, the set *LeaveThroughFin* is empty for an *EndFinally* instruction which corresponds to a `fault` handler. This means that the *EndFinally* instruction which terminates the execution of a `fault` handler region has no successors (apart from those defined in *excHandlers*).

$$\begin{aligned}&\mathit{LeaveThroughFin}(\mathit{meth},\mathit{pos}) =\\ &\quad \{\, \mathit{target} \in \mathit{Pc} \mid \exists \mathit{pos}' \in \mathit{Pc} \text{ such that } \mathit{code}(\mathit{pos}') = \mathit{Leave}(\mathit{target}) \text{ and}\\ &\quad [h' \in \mathit{excHA}(\mathit{meth}) \mid \mathit{isFinFromTo}(h', \mathit{pos}', \mathit{target})] = [\ldots, h],\\ &\quad \text{where } h \text{ is defined by } [h' \in \mathit{excHA}(\mathit{meth}) \mid \mathit{isInHandler}(\mathit{pos}, h')] = [h, \ldots] \,\}\end{aligned}$$

As the *EndFinally* instruction empties the evaluation stack, the stack state of the successors of *EndFinally* is the empty list.

We reason below for the definition of *succ* in case of the *Leave* instruction. If there is no `finally` handler from the *Leave* instruction to its target, the successor is given by the target instruction with an empty stack state (this is because the *Leave* instruction empties the evaluation stack). If there is a `finally` handler in between, say *h*, the successor given by the instruction at *handlerStart*(*h*) is already considered in *excHandlers*.

Example 4.2.6 *Consider again the skeleton example from Figure 4.20. We look here in detail at the successors – others than those defined via the exception handling array – of some of the Leave and EndFinally instructions.*

The Leave instruction at the code index 2 *has no successor (except the successors defined via the exception handling array) since the* `finally` *handler region is "on the way" from* 2 *to the target* 40*. The successor of the Leave instruction at the code index* 15 *is the instruction at the code index* 16 *since there is no* `finally` *block "on the way" from* 15 *to the target* 16*.*

Figure 4.20 Example: successors of *Leave* and *EndFinally* instructions.

```
.try
{
  .try
  {
    .try
    {
      .try
      {
        2: leave  40
        ⋮
        8: leave  23
      } // end try block
      catch NullReferenceException
      {
        9: ...
        ⋮
        15: leave 16
      } // end handler region
      16: ...
    } // end try block
    fault
    {
      17: ...
      ⋮
      22: endfinally
    } // end handler region
  } // end try block
  23: ...
  filter
  {
    24: ...
    ⋮
    26: endfilter
  } // end filter region
  {
    27: ...
    ⋮
    30: leave 41
  } // end handler region
} // end try block
finally
{
  31: ...
  ⋮
  39: endfinally
} // end handler region
40: ...
41: ...
```

The successors of the EndFinally instruction at the code index 39 *are defined via the set* $LeaveThroughFin(meth, 39)$*. This set consists of the targets* 40 *and* 41 *since the EndFinally instruction at* 39 *is the last EndFinally instruction "on the way" from the Leave instructions at* 2 *and* 30 *to their targets.*

4.2.5 Type Safety

This section extends the type safety proof in Section 4.1.6 to include the exception handling. We show that the properties guaranteed by Theorem 4.1.1 are preserved upon adding exceptions. However, besides those properties, we consider one more type safety property:

> ***Type safety***: Every handled exception is an object reference, whose type is derived from `Exception`.

The definition of well-typed methods in Section 4.1.5 Definition 4.1.9, which serves as the basis for the type safety proof, remains unchanged. The theorem asserting type safety of well-typed methods requires the following lemmas.

Lemma 4.2.7 claims that, after the *stackCursor* reaches in the *Unwind* pass the frame containing the suitable *handler*, *pc* is in the `try` block of *handler*.

Lemma 4.2.7 *Let* $stackCursor = (fr, n)$ *and* $handler = (fr', n')$ *be the stack cursor and the target handler, respectively, where* $fr' = (\_, \_, \_, \_, \_, mref')$*. If* $pass = Unwind$ *and* $sameFrame(fr, fr')$*, then* $isInTry(pc, h)$ *where* h *is the handler pointed to by handler, i.e.,* $excHA(mref')(n')$*.*

Proof. By the definitions of *matchCatch* and *matchFilter*, we have that $isInTry(pc, h)$ holds if $sameFrame(fr, fr')$, $pass = Unwind$, and $n = 0$. That the property is preserved can be easily proved by induction on the run of the exception mechanism EXCCLR. □

The following lemma states that if the current program counter is in a `filter` region while the exception mechanism is running in an *Unwind* pass, then there exists at least one more exception whose handling has not been completed yet.

Lemma 4.2.8 *If* $pass = Unwind$ *and there exists* $h \in excHA(meth)$ *such that* $isInFilter(pc, h)$*, then* $passRecStack \neq [\,]$*.*

Proof. By $pass = Unwind$, we derive that *exc* is defined, *i.e.*, the mechanism is handling an (inner) exception. According to [45, Partition I, §12.4], control is never permitted to enter a `filter` region except through the exception mechanism. This can only happen in the *StackWalk* pass of an (outer) exception whose associated record has been saved in *passRecStack*. It then remains to show that this record has not been discarded in the meantime. But a record associated to an exception can only be discarded while the exception is handled in the corresponding suitable handler region. However, this could not have occurred since the (outer) exception is still in the *StackWalk* pass. □

Lemma 4.2.9 shows that if there are no more handlers to be inspected in a *Leave* pass, then there cannot be any `finally` handlers "on the way" from *pc* to the *target* of the *Leave* pass.

Lemma 4.2.9 *If $pass = Leave$ and $existsHanWithinFrame(stackCursor)$ is false, then*

$$\{h \in excHA(meth) \mid isFinFromTo(h, pc, target)\} = \emptyset$$

Proof. Let $\mathcal{A}$ be the set $\{h \in excHA(meth) \mid isFinFromTo(h, pc, target)\}$. Let us assume that at the beginning of the *Leave* pass, *i.e.*, when the *stackCursor* is pointing to the index 0 in *excHA*, the set $\mathcal{A}$ is non-empty. However, for every handler in $\mathcal{A}$, say h, there will eventually be a step in the run of the exception mechanism which examines h for the current *Leave* pass. When h is going to be inspected, it is actually removed from $\mathcal{A}$. The reason is that when h's handler region is executed, pc is not anymore in the `try` block of h. □

We are now ready to state the theorem asserting type safety of well-typed methods. In addition to the invariants considered in Section 4.1.6, an exception specific invariant **(exc)** is added. This ensures that if the exception mechanism is handling an exception, the exception type is a subtype of `Exception`, regardless of the flag *wrapNonExc*.

Theorem 4.2.1 (Type safety of well-typed methods) *Let $[fr'_1, \ldots, fr'_{k-1}]$ be the frameStack and fr_k be the current frame. For every $i = 1, k-1$, we denote by fr_i the succession of fr'_i.*

We assume that the type system and the bytecode structure of the methods in $(fr_i)_i$ satisfy the conditions stated in Section 4.1.4.2 and Section 4.2.4.1. Moreover, we assume that all methods of $(fr_i)_{i=1}^{k}$ are well-typed.

Let $fr_i = (pc^*, locAdr^*, \_, argAdr^*, evalStack^*, meth^*)$ be one of the frames $(fr_i)_{i=1}^{k}$. We denote by $locVal^*$ and $argVal^*$ the functions $locVal$ and $argVal$ derived based on $locAdr^*$ and $argAdr^*$, respectively. If fr_i is the current frame, we consider $initState^*$ to be the current initialization status $initState$. If fr_i is not the current frame, we denote by $initState^*$ the succession of $initState$ for fr_i. Let $(argZeroT_j, evalStackT_j)_{j \in \mathcal{D}}$ be a family of type states $meth^*$ is well-typable with.

The invariants in Theorem 4.1.1 and **(exc)** *are satisfied at run-time for every frame fr_i (for the current frame fr_k, the invariants* **(stack1)** *and* **(stack2)** *should hold if $switch = Noswitch$):*

(exc) *If ref is an exception reference in $passRecStack$, then $[\,] \vdash ref :$ `Exception`. If exc is defined, then $[\,] \vdash exc :$ `Exception`.*

Proof. The proof proceeds similarly with the proof of Theorem 4.1.1, *i.e.*, by induction on the run of the semantics model $\mathrm{CLR}_{\mathcal{E}}$ including exception handling. To extend the proof of Theorem 4.1.1, we consider here that the exception mechanism has control, *i.e.*, $switch = ExcMech$. Different cases have to be considered depending on *pass*. We only analyze here the cases when $pass = Unwind$ and $pass = Leave$.

Case 1 Assume that the exception mechanism is currently executing an *Unwind* pass of an exception, *i.e.*, $pass = Unwind$.

Subcase 1.1 If there are still handlers to be inspected in the current *frame*, let h be the handler pointed to by the *stackCursor*.

Subcase 1.1.1 Assume that *handler* is defined and h matches *handler*. By the definition of *matchTargetHan*, we have that *handler* and *stackCursor* are pointing to the same frame, *i.e.*, $sameFrame(pr_1(stackCursor), pr_1(handler))$. By Lemma 4.2.7, we get $isInTry(pc, h)$. This and the definitions of *succ* and *excHandlers* imply:

- If h is a `filter` handler, then $(handlerStart(h), argZeroT_{pc}, [\texttt{object}])$ is an element of the set $succ(meth, pc, argZeroT_{pc}, evalStackT_{pc})$;
- If h is a `catch` handler, then $(handlerStart(h), argZeroT_{pc}, [typeExc(h)])$ is in the set $succ(meth, pc, argZeroT_{pc}, evalStackT_{pc})$.

The exception mechanism runs h's handler region by switching to normal execution mode: The new pc is $handlerStart(h)$ (so, **(pc)** is preserved), the new *evalStack* is $[exc]$ (if *wrapNonExc* is *True*) or $[wrappedExc(exc)]$ (if *wrapNonExc* is *False*), and *switch* becomes *Noswitch*. Definition 4.1.9 **(wt5)** and **(pc)** ensure $handlerStart(h) \in \mathcal{D}$ independent of h's kind,

$$[\texttt{object}] \sqsubseteq_{len} evalStackT_{handlerStart(h)} \tag{4.11}$$

if h is a `filter`, and

$$[typeExc(h)] \sqsubseteq_{len} evalStackT_{handlerStart(h)} \tag{4.12}$$

if h is a `catch` and $typeExc(h) \in RefType$, and

$$[boxed(typeExc(h))] \sqsubseteq_{len} evalStackT_{handlerStart(h)} \tag{4.13}$$

if h is a `catch` and $typeExc(h) \in ValueType$.

By the induction hypothesis – that is, by **(exc)** – we have $[\,] \vdash exc : \texttt{Exception}$. So, *exc* is an object reference and, by Definition 4.1.12, $type(exc) \preceq \texttt{Exception}$. By this, (4.11) and Lemma 4.1.5, we get that **(stack1)** and **(stack2)** are preserved if h is a `filter` handler.

If h is a `catch` handler, by the definition of *matchCatch*, we have

$$\begin{aligned}&(wrapNonExc \Rightarrow actualTypeOf(exc) \preceq typeExc(h)) \wedge \\ &(\neg wrapNonExc \Rightarrow actualTypeOf(wrappedExc(exc)) \preceq typeExc(h))\end{aligned} \tag{4.14}$$

The invariants **(stack1)** and **(stack2)** follow from Lemma 4.1.5, (4.12), (4.13), and (4.14).

Hence, **(stack1)** and **(stack2)** are preserved, regardless of h's kind.

Subcase 1.1.2 The case when h is a matching `finally` is treated similarly to Subcase 2.1.1.1 except that the new *evalStack* is empty, and therefore the invariants **(stack1)** and **(stack2)** are obviously preserved.

Subcase 1.1.3 Assume that h is a `filter` whose `filter` region contains pc. By Lemma 4.2.8, *passRecStack* cannot be empty. This means that *switch* is still *ExcMech* after the exception mechanism executes a step. Consequently, nothing has to be proved for the current frame for **(stack1)** and **(stack2)**. The invariant **(pc)** simply follows from the induction hypothesis.

Subcase 1.1.4 If h is not of any of the above types, then *stackCursor* is incremented to point to the next handler. The invariants are simply preserved.

Subcase 1.2 If there are no more handlers to be examined in the current *frame*, *frame* is discarded, and the *stackCursor* is reset to point to the *frame*'s caller frame. The execution model *switch* is still *ExcMech*, and therefore the invariants **(stack1)** and **(stack2)** are not supposed to hold. The induction hypothesis implies **(pc)**.

Case 2 Consider now that the exception mechanism is in a *Leave* pass, *i.e.*, $pass = Leave$.

Subcase 2.1 Assume that there are still handlers to be examined in the current frame. Let h be the handler pointed to by the *stackCursor*.

Subcase 2.1.1 If h is a `finally` handler "on the way" from the *pc* to *target*, *i.e.*, $isFinFromTo(h, pc, target)$, then the exception mechanism runs h's handler region by switching to the normal execution mode: The new *pc* is $handlerStart(h)$, the new *evalStack* is $[\,]$, and *switch* is *Noswitch*. From the definitions of *succ* and *excHandlers*, we know that $(handlerStart(h), argZeroT_{pc}, [\,])$ is an element of $succ(meth, pc, argZeroT_{pc}, evalStackT_{pc})$. By this, **(pc)**, and Definition 4.1.9 **(wt6)**, we derive $handlerStart(h) \in \mathcal{D}$ and also $[\,] \sqsubseteq_{len} evalStackT_{handlerStart(h)}$. This means that **(pc)**, **(stack1)**, and **(stack2)** are preserved for the current *frame* (the last two invariants hold since $evalStackT_{handlerStart(h)}$ is necessarily $[\,]$).

Subcase 2.1.2 If h is real handler whose handler region is "exited on the way" from the *pc* to *target*, then a pass record is popped from *passRecStack*. So, no invariant is affected.

Subcase 2.2.2 Let us assume that there are no more handlers to be inspected in the current *frame*. The exception mechanism switches to normal execution mode: *pc* is set to *target*, *evalStack* to $[\,]$, *switch* to *Noswitch*, and the current *Leave* record is popped from *passRecStack*. The only invariants that have to be proved are **(pc)**, **(stack1)**, and **(stack2)**. By Lemma 4.2.9, $\{h \in excHA(meth) \mid isFinFromTo(h, pc, target)\}$ is empty. From the definition of *succ*, we have that $(target, argZeroT_{pc}, [\,])$ is in $succ(meth, pc, argZeroT_{pc}, evalStackT_{pc})$. From this, **(pc)**, and Definition 4.1.9 **(wt6)**, we obtain $target \in \mathcal{D}$ and $[\,] \sqsubseteq_{len} evalStackT_{target}$. Therefore, **(pc)**, **(stack1)**, and **(stack2)** are preserved for the current *frame* (the last two invariants hold since $evalStackT_{target}$ is $[\,]$). □

Remark 4.2.7 *As pointed out in the proof of Theorem 4.1.1,* ***Subcase 2.1****, the invariants* **(stack1)** *and* **(stack2)** *are only guaranteed to hold if switch* $=$ *Noswitch. The semantics of the exception handling confirms once again this aspect. If, for instance, a method call receives a* `null` *as a* `this` *pointer,* `null` *is popped from the evaluation stack and a "null" check is done. Consequently, a* `NullReferenceException` *is then raised. So, while the stack state corresponding to the call instruction remains unchanged, the evaluation stack has one less value. Obviously,* **(stack1)** *and* **(stack2)** *are not satisfied. However, this is alright since the evaluation stack is not accessed as long as switch* $=$ *ExcMech.*

4.3 The Generics

Kennedy and Syme proposed adding support for generics in the CLR [88]. Their proposal was at the basis of the generics implementation in the .NET Framework (v2.0) [3]. *Generics* are an extension to the CLR that allows to define types and methods parameterized in terms of types. That means, for example, that the same method implementation might be able to deal with both `int32` and `object`, without knowing in advance which type it is working with. Some strengths of the CLR generics are verification-time type safety, efficiency, and clarity.

A class or an interface whose declaration is parameterized by one or more *generic parameters* is called *generic type*. A method declared within a type, whose signature includes one or more generic parameters (not present in the type declaration itself) is called *generic method*. When the generic code is referenced by consumer code, the generic types and methods are *instantiated* by replacing every generic parameter with a *generic argument*.

To be useful, generic code needs to be able to call methods other than those defined by `object`. As the generic code is only valid if it works for every possible constructed generic instantiation, CLR supports a special feature called *constraints* (also known as *F-bounded polymorphism* [33]). The constraints are an optional component of a generic type or method definition. A generic type may define any number of constraints, and each constraint applies an arbitrary limitation on the types that can be used as generic arguments. By restricting the types that can be used in the construction of a generic type, the restrictions on the code that references the constrained type parameter are relaxed.

To allow conversions between related (but distinct) instances of the same generic type, the CLR supports *variance* on generic parameters, in the form of *covariant* generic arguments and *contravariant* generic arguments. For simplicity, variance is only permitted on generic interfaces. *Covariance* means that the subtype relationship of the generic type varies directly with the relationship of the generic argument, whereas the *contravariance* means that the subtype relationship of the generic type varies inversely with the relationship of the generic argument. An example of a language that provides support for covariance is Eiffel [46]. The contravariance is supported, for instance, by Java (v5.0) [67], Sather [80], Scala [13], and OCaml [12]. The .NET compliant languages not willing to support variance can ignore this feature and treat all generic parameters as non-variant.

Challenges Several versions of Eiffel turned out to be unsafe also due to the variance on generic parameters [37, 75]. Thus, in the context of generic parameter variance, virtual method calls are problematic in ensuring type safety. Let us assume, for example, that the bytecode contains a virtual call of the method $T::M$. Let $T'::M$ be the method that will be invoked at run-time. The following aspects are critical for type safety:

- due to the variance, the signature of $T'::M$ may not match the signature of $T::M$; Consequently, the following potential risks for type safety may arise: (1) the arguments of $T::M$ ensured to be of $T::M$'s argument types might be treated in $T'::M$ as being of another types, namely of $T'::M$'s argument types; (2) the return value of $T'::M$ guaranteed to be of $T'::M$'s return type might be handled in the frame containing the virtual call of $T::M$ as being of another type, namely of $T::M$'s return type.

- if T::M is a generic method, T'::M should also be generic, but its constraint types may not match the corresponding constraint types of T::M. This is a potential risk for type safety as T::M's generic arguments, restricted by T::M's constraints, might be treated in T'::M as being bounded by other constraints, namely by T'::M's constraint types. As the latter constraints might not be satisfied by T::M's generic arguments, type safety would be violated.

Another critical issue concerns the validity of the run-time instantiation of generic types and methods. Let us assume, for example, that the types $(U_i)_{i=0}^{n-1}$ satisfy at verification time the constraints associated to the type $C\langle U_0, \ldots, U_{n-1}\rangle$. As any U_i can involve generic parameters, one has to make sure that after $(U_i)_{i=0}^{n-1}$ get instantiated at run-time with the generic arguments $(V_i)_{i=0}^{n-1}$, also $(V_i)_{i=0}^{n-1}$ satisfy the constraints.

Of high risk for type safety turns out to be the boxing of nullable types. Through nullable types, CLR provides support for nullable forms of all value types. A *nullable type* is a *generic value type* (that implements no interface) of the form `Nullable`$\langle T\rangle$, where T can only be instantiated with value types, other than nullable types. Let *NullableType* denote the (recursively defined) set of nullable types $\{Nullable\langle T\rangle \mid T \in ValueType \setminus NullableType\}$.

A nullable type `Nullable`$\langle T\rangle$ combines a value of the underlying type T with a boolean null indicator. More precisely, an instance of a type `Nullable`$\langle T\rangle$ has two instance fields: `HasValue`, of type `bool` (however, remember that the booleans are treated as `int32` values), indicating whether the *underlying value* is defined, and `Value`, of type T, carrying the defined value (if any). Boxing nullable types results always in situations when the dynamic (run-time) types of the boxed values do not match the static (verification) types predicted by the bytecode verification. Surprisingly, the cause of this problem, *i.e.*, the special semantics of boxing nullable types, is not documented in the ECMA Standard [45]. The only information we have originates from [109, 42].

Section plan The remainder of this section is organized as follows. Section 4.3.1 gives a formalization of the extension with generics of the CLR type system. Section 4.3.2 defines CLR$_G$ by refining the operational semantics model CLR$_E$. Section 4.3.3 deals with the changes to the bytecode verification. Finally, Section 4.3.4 relates the type system with generics and the operational semantics model CLR$_G$ in theorem asserting type safety of well-typed methods.

4.3.1 The Polymorphic Type System

This section extends the type system defined in Section 4.1.2 by formally introducing the generic types and methods described above. The universe *Type* is extended with generic parameters, elements of *GenericParam*, as shown in Table 4.16. Table 4.17 gathers the selector functions which we define to deal with generics[25].

In the bytecode, the generic parameters are referred to as `!`i, whereas the method generic parameters are addressed as `!!`i. Thus, !i denotes the i-th type generic parameter (numbered from left-to-right in the relevant type declaration). Similarly, `!!`i designates the i-th method generic parameter (numbered from left-to-right in the relevant method declaration). The universe *GenericParam* is defined relative to a given (possibly generic) type or method[26].

Universe of types			Typical use
Type	=	*RefType* ∪ *ValueType* ∪ *GenericParam*	T, T', T''
GenericParam			$!i$, $!!i$, $X, X_1, \ldots, X_n$

Table 4.16: Type system extension with generics.

Function definition	Function name
$genParamNo : Map(GenericType \cup MRef, \mathbb{N})$	generic type/method's arity
$genArg : Map(GenericType \cup MRef, List(Type \setminus PointerType))$	generic argument list
$constr_i^T : Type \setminus PointerType$	T's i-th constraint
$constr_i^{mref} : Type \setminus PointerType$	$mref$'s i-th constraint
$(var_i^{I'n})_{i=0}^{n-1} : List(\{+, -, \emptyset\})$	$I'n$'s variances array

Table 4.17: Generics specific selector functions.

Let *GenericType* stand for the universe of generic types. Only the classes and interfaces can be generic: $GenericType \subseteq Class \cup Interface$. Typically, $T'n$ represents the (*raw*) name of a generic type T with n generic parameters. As a convention, we assume that *GenericType* has only instantiated generic types and not generic types' raw names.

Every generic parameter can have an optional constraint consisting of a type[27]. Every generic argument which replaces a generic parameter should be compatible *when boxed* with the type given in the corresponding constraint (see Definition 4.3.4 for the compatibility relation for verification types). This means, for instance, that every value type can be used as generic argument for a generic parameter constrained by `object`.

A covariant generic parameter is marked with "+" in the interface declaration, whereas "−" is used to denote a contravariant generic parameter. For the sake of notation, we mark with "∅" the non-variant parameters.

The subtype relation $\preceq$ of Definition 4.1.1 is extended to generic types. The relation is defined for both *open generic types*, *i.e.*, instantiations of generic types that have not been supplied all generic arguments for their generic parameters (because they are open to accepting more generic arguments), and *closed generic types*, *i.e.*, instantiations of generic types which have been supplied all of their generic arguments. The definition captures the intuitive notion of generic parameters variance. In this definition, $T \circ [U_i/X_i]_{i=0}^{n-1}$ designates the type T, where each generic parameter X_i is substituted by the type U_i.

Definition 4.3.1 (Subtype relation) *The* subtype relation $\preceq$ *is the least reflexive and transitive relation such that*

- *if* $T \in Class \cup Interface$ *and* T' *is* `object`*, or*

[25]Several examples illustrating the definitions of the selector functions are included in [59, 60].

[26]As a matter of fact, *GenericParam* should be defined as a map assigning to every (possibly generic) type/method the set of its generic parameters. However, as the parameters' reference is bounded to the relevant generic type/method bodies, we take the liberty to define *GenericParam* as a universe, making sure that no confusion can occur.

[27]This differs slightly from [45], which allows a constraint to have more than one type. The restriction does not reduce the complexity, but simplifies the exposition.

- *if $T \in$ ValueType and T' is* `System.ValueType`*, or*
- *if $T \in$ ObjClass $\setminus$ GenericType and T' is a base class of T or an interface implemented by T, or*
- *if $T \in$ Interface $\setminus$ GenericType and T' is an interface implemented by T, or*
- *if $T \in$ ValueClass $\setminus$ GenericType and T' is an interface implemented by T, or*
- *if T is $C\langle U_0, \ldots, U_{n-1}\rangle$, where $C'n$ is a generic object class with the generic parameters $(X_i)_{i=0}^{n-1}$, T'' is a base class of $C'n$ or an interface implemented by $C'n$, and T' is $T'' \circ [U_i/X_i]_{i=0}^{n-1}$, or*
- *if T is $I\langle U_0, \ldots, U_{n-1}\rangle$, where $I'n$ is a generic interface with the generic parameters $(X_i)_{i=0}^{n-1}$, T'' is an interface implemented by $I'n$, and T' is $T'' \circ [U_i/X_i]_{i=0}^{n-1}$, or*
- *if T is $C\langle U_0, \ldots, U_{n-1}\rangle$, where $C'n$ is a generic value class with the generic parameters $(X_i)_{i=0}^{n-1}$, T'' is an interface implemented by $C'n$, and T' is $T'' \circ [U_i/X_i]_{i=0}^{n-1}$, or*
- *if T is $I\langle U_0, \ldots, U_{n-1}\rangle$ and T' is $I\langle V_0, \ldots, V_{n-1}\rangle$, and for every $i = 0, n-1$ the following conditions hold:*
  - *if $var_i^{I'n} = \emptyset$ or $V_i \in$ ValueType or $V_i \in$ GenericParam, then $U_i = V_i$;*
  - *if $var_i^{I'n} = +$, then $U_i \preceq V_i$;*
  - *if $var_i^{I'n} = -$, then $V_i \preceq U_i$;*

then $T \preceq T'$.

Since it is possible for different methods to have identical signatures when the declaring types are instantiated and consequently ambiguous references, *the method references have the return type and parameter types uninstantiated.*

Example 4.3.1 *Let us consider the skeleton of the generic class C'2 in Figure 4.21. One can easily notice that the generic instantiation $C\langle$*`string`*, *`string`*$\rangle$ contains two generic methods, which can only be distinguished through their uninstantiated signatures, namely $!0\ C\langle \mathtt{string}, \mathtt{string}\rangle :: M\langle Z\rangle(!0, !!0)$ and $!1\ C\langle \mathtt{string}, \mathtt{string}\rangle :: M\langle Z\rangle(!1, !!0)$. To simplify a few aspects of our formal development, we assume that the environment specific functions retType and paramTypes are defined with respect to these uninstantiated signatures. This means, for instance, that it holds:*

$$retType(!0\ C\langle \mathtt{string}, \mathtt{string}\rangle :: M\langle Z\rangle(!0, !!0)) = !0$$

$$paramTypes(!0\ C\langle \mathtt{string}, \mathtt{string}\rangle :: M\langle Z\rangle(!0, !!0)) = [!0, !!0]$$

To get the instantiated return type and parameter types of a generic method reference, we define $inst : Map(MRef, \{[U_i/X_i]_{i=0}^{n} \mid (U_i, X_i) \in Type \times GenericParam$ for every $i = 0, n\})$, which assigns to every generic method the *type substitution* (or simply *substitution* when it is obvious that we are talking about types) that should be applied to the method return type and

Figure 4.21 Example: uninstantiated method return type and parameter types.

```
.class public auto ansi C'2⟨ X, Y ⟩ extends System.Object
{
        .field public !0 F
        .method instance !0 M ⟨ ( I⟨ !0 ⟩ ) Z⟩ (!0 i, !!0 j)
        {
          ...
        }
        .method instance !1 M⟨Z⟩ (!1 i, !!0 j)
        {
          ...
          5: box !!0
          ...
        }
        ...
}


.class interface private abstract auto ansi I'1⟨X⟩
{
        ...
}
```

signature. However, due to technical considerations, the "instantiation" function *inst* is also defined for non-generic methods defined by both generic and non-generic types. If the method is not generic and is declared by a non-generic type, *inst* returns the *empty substitution* ϵ. If the method is not generic, but is declared by a generic type, then *inst* yields the generic type's underlying instantiation. Similarly, if the method is generic and declared by a non-generic type, then *inst* generates the generic method's underlying instantiation. Finally, if the method is generic and declared by a generic type, then *inst* returns the substitution formed by composing the generic type's and method's underlying instantiations. Note that the same generic parameter may occur in both the range and domain of a type substitution.

Definition 4.3.2 *Given a method T::M, we define*

$$inst(T{::}M) = \begin{cases} \epsilon, & \textit{if } T \notin \textit{GenericType and} \\ & T{::}M \textit{ is not generic;} \\ [genArg(T)(i)/\,!\,i]_{i=0}^{genParamNo(T)-1}, & \textit{if } T \in \textit{GenericType and} \\ & T{::}M \textit{ is not generic;} \\ [genArg(T{::}M)(i)/\,!\,!\,i]_{i=0}^{genParamNo(T::M)-1}, & \textit{if } T \notin \textit{GenericType and} \\ & T{::}M \textit{ is generic;} \\ [genArg(T)(i)/\,!\,i]_{i=0}^{genParamNo(T)-1}. & \textit{if } T \in \textit{GenericType and} \\ [genArg(T{::}M)(i)/\,!\,!\,i]_{i=0}^{genParamNo(T::M)-1}, & T{::}M \textit{ is generic.} \end{cases}$$

Example 4.3.2 *To illustrate the intuition behind the definition of inst, consider again the generic class C'2 defined in Figure 4.21. As C*⟨`string`, `string`⟩ *:: M*⟨`object`⟩ *is generic and declared by a generic type, we have:*

$$\begin{aligned} & \mathit{inst}(\texttt{!1}\ C\langle\texttt{string},\texttt{string}\rangle :: M\langle\texttt{object}\rangle(\texttt{!1},\texttt{!!0})) \\ = \ & [\texttt{string}/\texttt{!0},\texttt{string}/\texttt{!1}] \circ [\texttt{object}/\texttt{!!0}] \\ = \ & [\texttt{string}/\texttt{!0},\texttt{string}/\texttt{!1},\texttt{object}/\texttt{!!0}] \end{aligned}$$

Technical Conventions Similarly as the return type *retType* and parameter types *paramTypes* (and implicitly also *argTypes*), the local signature *locTypes* is considered uninstantiated in our approach. In contrast, *constr* and *fieldType* are considered to return results instantiated according to the underlying context, unless explicitly stated otherwise.

Example 4.3.3 *Let us consider again the generic class $C'2$ in Figure 4.21. Following the above technical conventions, the constraint type of the generic parameter* `!!0` *associated to* `!0` *$C\langle\texttt{string},\texttt{string}\rangle :: M\langle\texttt{object}\rangle(\texttt{!0},\texttt{!!0})$ is $I\langle\texttt{string}\rangle$, as opposed to $I\langle\texttt{!0}\rangle$. Also, the declared type of the field $C\langle\texttt{string},\texttt{string}\rangle::F$ is* `string`*, as opposed to* `!0`*.*

4.3.2 The Bytecode Semantics

In this section, we refine the semantics of the bytecode language with generics. Besides the polymorphic type system and technical conventions stipulated in Section 4.3.1, there is one more extension concerning the static semantics. We consider that the function *code* yields instructions which are *not* instantiated relative to the underlying context. This is for purely technical reasons. When an instruction is executed at run-time, the possible open generic types referenced by the instruction are priory instantiated. As we would like to keep the correspondence between the uninstantiated instructions on which the bytecode verification is performed and the instantiated instructions being executed, we consider that *code* returns uninstantiated instructions.

Example 4.3.4 *To illustrate the above assumption, let us have another look at the class $C'2$ in Figure 4.21. In the second method, the instruction corresponding to the index 5 returned by the function code is $Box(\texttt{!!0})$, regardless of the instantiation which is applied to the method. Formally,*

$$\mathit{code}(C\langle\texttt{string},\texttt{string}\rangle :: M\langle\texttt{object}\rangle(\texttt{!0},\texttt{!!0}))(5) = \mathit{Box}(\texttt{!!0})$$

The operational semantics model $\mathrm{CLR}_{\mathcal{E}}$ for the bytecode language with exceptions is refined by $\mathrm{CLR}_{\mathcal{G}}$ for the bytecode language with generics. The above convention on the function *code* is reflected in the refinement of the execution scheme *execScheme*:

```
$execScheme \equiv$
  if $switch = ExcMech$ then
    EXCCLR
  elseif $switch = Noswitch$ then
    EXECCLR$(code(pc) \circ inst(meth))$
  else SWITCHCLR
```

The set of bytecode instructions is extended with the prefix *Constrained*, informally described in Table 4.18, which is permitted only on *CallVirt* instructions. The operational semantics rules for the *CallVirt* instruction prefixed by *Constrained* and the operational semantics rules, defined in Sections 4.1.3.1 and 4.2.3.4, affected by the run-time specialization of generic types/methods are defined, respectively, refined in Figures 4.3.2, 4.3.2, and 4.3.2.

Every instruction is executed under the assumption that the current method *meth* is instantiated (specialized), *i.e.*, every generic parameter of the enclosing type or method is replaced by the corresponding generic argument. Consequently, the substitution given by $inst(meth)$ is applied to every instruction. If neither the enclosing class $classNm(meth)$ nor *meth* is generic, then $inst(meth)$ is, according to Definition 4.3.2, the empty substitution ϵ. It then turns out that the semantics rules defined in this section (except of that for the prefixed *CallVirt* instruction) are refinements for the rules defined in Section 4.1.3.1 and 4.2.3.4.

There are some non-obvious refinements triggered by the extension with generics: the boxing of nullable types, the unboxing to nullable types, and the dynamic method dispatch.

Boxing nullable types It occurs when the *Box* instruction's type token argument is a type `Nullable`$\langle T\rangle$ or a generic parameter which is instantiated (by means of $inst(meth)$) with a type `Nullable`$\langle T\rangle$. When an instance *val* of a nullable type `Nullable`$\langle T\rangle$ is boxed, the below defined macro BOXNULLABLE is executed. If *val* is not defined, *i.e.*, *val*(`HasValue`) is *False*, then the result of boxing is a `null` reference. If *val* is defined, *i.e.*, *val*(`HasValue`) is *True*, then *val*'s underlying value *val*(`Value`) is boxed (through the *NewBox* macro) and not the `Nullable` instance *val* itself[28]. In particular, this means that "Nullable" is suppressed in the run-time type of the newly created object. As we will point out in Section 4.3.3, this makes type safety a concern.

```
BOXNULLABLE($val$, Nullable$\langle T\rangle$) $\equiv$
  let $ref$ = if $val$(HasValue) then $NewBox(val(\texttt{Value}), T)$
              else null in
    $evalStack := pop(evalStack) \cdot [ref]$
```

Unboxing back to nullable types If the type token argument of an *Unbox* or *Unbox.Any* instruction is a nullable type `Nullable`$\langle T\rangle$ or a generic parameter instantiated with a type `Nullable`$\langle T\rangle$, then $\mathrm{CLR}_{\mathcal{G}}$ allocates and initializes a new `Nullable`$\langle T\rangle$ instance:

- if the top stack element, say *ref*, is not `null`, then `Value` is set to whatever T value has been boxed inside *ref*, and `HasValue` becomes *True*.
- if *ref* is the `null` reference, then `HasValue` is set to *False* and `Value` becomes the default value of type T.

If the value *ref* is not a boxed object of type T, as opposed to the type `Nullable`$\langle T\rangle$, then an `InvalidCastException` is thrown.

[28]The reason behind this design is to avoid have a non-`null` box containing a `null`, along the same lines as one cannot construct types of the form `Nullable`$\langle$`Nullable`$\langle T\rangle\rangle$.

Instruction	Informal description
$Constrained(T'').CallVirt(T', T::M)$	Call the virtual method $T::M$, of return type T', on a generic parameter T''.

Table 4.18: The generic specific instructions.

As usually, the *Unbox* instruction pushes a pointer to the newly created `Nullable`$\langle T\rangle$ instance onto the stack, whereas the *Unbox.Any* instruction loads the instance.

To unbox back to nullable types, we define the macro UNBOXNULLABLE. This takes as arguments the to-be-unboxed object, the nullable type, and a boolean indicating whether an *address* is required to be pushed on the stack (the case of the *Unbox* instruction) or a *value* (the case of the *Unbox.Any* instruction).

Remark 4.3.1 *Note that the creation of the nullable instance represents, besides the value class instance creation with NewObj, a new case when the allocated address is considered for the local memory pool locPool. The reason behind this modelling decision is the one pointed out in Remark 4.1.10, namely the risk of "loosing" the typing of the allocated address.*

```
$\text{UNBOXNULLABLE}(ref, \texttt{Nullable}\langle T\rangle, reqAdr) \equiv$
  $\textbf{if } ref = \texttt{null} \lor actualTypeOf(ref) \preceq T \textbf{ then}$
    $\textbf{let } adr = new(Adr, \texttt{Nullable}\langle T\rangle) \textbf{ in}$
      $locPool := locPool \cup \{(adr, \texttt{Nullable}\langle T\rangle)\}$
      $\textbf{if } reqAdr \textbf{ then } evalStack := pop(evalStack) \cdot [adr]$
      $\textbf{else } evalStack := pop(evalStack) \cdot [memVal(adr, \texttt{Nullable}\langle T\rangle)]$
      $\textbf{let } (val, val') = \textbf{ if } ref = \texttt{null} \textbf{ then } (0, defVal(T))$
                          $\textbf{else } (1, memVal(addressOf(ref), T)) \textbf{ in}$
        $\text{WRITEMEM}(fieldAdr(adr, \texttt{Nullable}\langle T\rangle::\texttt{HasValue}), \texttt{bool}, val)$
        $\text{WRITEMEM}(fieldAdr(adr, \texttt{Nullable}\langle T\rangle::\texttt{Value}), T, val')$
  $\textbf{else } \text{RAISE}(\texttt{InvalidCastException})$
```

Dynamic method dispatch The ECMA Standard [45] does not specify what is the effect of the generic parameter variance on the dynamical method lookup. The definition of *lookUp* in Section 4.1.2 is refined to take the variance into account.

Definition 4.3.3 (Method lookup) *For a possibly generic*[29] *method $T'::M\langle T_0, \ldots, T_{n-1}\rangle$ and a type T, we define:*

[29]If the method is not generic, the generic arguments $(T_i)_i$ should simply be omitted.

Figure 4.22 The refined operational semantics rules.

```
$\textsc{execCLR}(instr) \equiv$ **match** $instr$
  ⋮
  $LoadLoc(n) \circ inst(meth)$        $\rightarrow$ **let** $T = locTypes(n) \circ inst(meth)$ **in**
                                          $evalStack := evalStack \cdot [memVal(locAdr(n), T)]$
                                          $pc := pc + 1$
  $StoreLoc(n) \circ inst(meth)$       $\rightarrow$ **let** $(evalStack', [val]) = split(evalStack, 1)$ **in**
                                          $\textsc{WriteMem}(locAdr(n), locTypes(n) \circ inst(meth), val)$
                                          $evalStack := evalStack'$
                                          $pc := pc + 1$
  $LoadArg(n) \circ inst(meth)$        $\rightarrow$ **let** $T = argTypes(n) \circ inst(meth)$ **in**
                                          $evalStack := evalStack \cdot [memVal(argAdr(n), T)]$
                                          $pc := pc + 1$
  $StoreArg(n) \circ inst(meth)$       $\rightarrow$ **let** $(evalStack', [val]) = split(evalStack, 1)$ **in**
                                          $\textsc{WriteMem}(argAdr(n), argTypes(n) \circ inst(meth), val)$
                                          $evalStack := evalStack'$
                                          $pc := pc + 1$
  $Call(tail, \_, T{::}M) \circ inst(meth)$ $\rightarrow$ **if** $T \circ inst(meth) =$ System.Delegate **then**
                                          **let** $[ref, ref'] = take(evalStack, argNo(T{::}M))$ **in**
                                            **if** $M =$ Combine **then**
                                              $\textsc{DelegateCombine}(T \circ inst(meth), ref, ref')$
                                            **if** $M =$ Remove **then**
                                              $\textsc{DelegateRemove}(T \circ inst(meth), ref, ref')$
                                            **if** $M =$ op_Equality **then**
                                              $\textsc{DelegateEqual}(ref, ref')$
                                        **else let** $(evalStack', [ref] \cdot vals) = split(evalStack, argNo(T{::}M))$ **in**
                                          **if** $ref \neq$ null **then**
                                            **if** $T \circ inst(meth) \in DelegateClass$ **then**
                                              **if** $M =$ _length **then**
                                                $evalStack := evalStack' \cdot [length(invocationList(ref))]$
                                              **if** $M =$ _invoke **then**
                                                $\textsc{DelegateCall}(ref, vals)$
                                            **else** $evalStack := evalStack'$
                                                 $switch \quad := Invoke(tail, T{::}M \circ inst(meth), [ref] \cdot vals)$
                                          **else** $\textsc{Raise}($NullReferenceException$)$
  $CallVirt(tail, \_, T{::}M) \circ inst(meth)$ $\rightarrow$ **let** $(evalStack', vals) = split(evalStack, argNo(T{::}M))$ **in**
                                          $evalStack := evalStack'$
                                          $\textsc{VirtCall}(tail, T{::}M \circ inst(meth), vals)$
  $Constrained(T'').$
  $CallVirt(\_, T{::}M) \circ inst(meth)$ $\rightarrow$ **let** $(evalStack', [adr] \cdot vals) = split(evalStack, argNo(T{::}M))$ **in**
                                     $evalStack := evalStack'$
                                     **if** $T'' \circ inst(meth) \in RefType$ **then**
                                       **let** $ref = memVal(adr, T'' \circ inst(meth))$ **in**
                                         $\textsc{VirtCall}(False, T{::}M \circ inst(meth), [ref] \cdot vals)$
                                     **elseif** $T'' \circ inst(meth) \in ValueClass \wedge$
                                     $T''{::}M \circ inst(meth)$ implements $T{::}M \circ inst(meth)$ **then**
                                       $switch := Invoke(False, T''{::}M \circ inst(meth), [adr] \cdot vals)$
                                     **else let** $ref = NewBox(memVal(adr, T'' \circ inst(meth)), T'' \circ inst(meth))$ **in**
                                       $\textsc{VirtCall}(False, T{::}M \circ inst(meth), [ref] \cdot vals)$
  $Return \circ inst(meth)$            $\rightarrow$ **if** $retType(meth) \circ inst(meth) =$ void **then** $\textsc{Result}([\,])$
                                     **else** $\textsc{Result}([top(evalStack)])$
```

Figure 4.23 The refined operational semantics rules (continued).

```
$\textsc{execCLR}(instr) \equiv$ match $instr$
  $\vdots$
  $NewObj(C::\texttt{.ctor}) \circ inst(meth) \rightarrow$ let $vals = take(evalStack, argNo(C::\texttt{.ctor}) - 1)$ in
      if $C \circ inst(meth) \in DelegateClass$ then
        $\textsc{DelegateCreate}(C \circ inst(meth), vals)$
      elseif $C \circ inst(meth) \in ObjClass$ then
        $\textsc{ObjectCreate}(C::\texttt{.ctor} \circ inst(meth), vals)$
      else $\textsc{ValueClassCreate}(C::\texttt{.ctor} \circ inst(meth), vals)$
  $LoadField(T, C::F) \circ inst(meth) \rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
      if $val \neq \texttt{null}$ then
        let $val' = memVal(fieldAdr(val, C::F \circ inst(meth)), T \circ inst(meth))$ in
          $evalStack := evalStack' \cdot [val']$
          $pc := pc + 1$
      else $\textsc{Raise}(\texttt{NullReferenceException})$
  $StoreField(T, C::F) \circ inst(meth) \rightarrow$ let $(evalStack', [val, val']) = split(evalStack, 2)$ in
      if $val \neq \texttt{null}$ then
        $\textsc{WriteMem}(fieldAdr(val, C::F \circ inst(meth)), T, val')$
        $evalStack := evalStack'$
        $pc := pc + 1$
      else $\textsc{Raise}(\texttt{NullReferenceException})$
  $LoadFieldA(\_, C::F) \circ inst(meth) \rightarrow$ let $(evalStack', [val]) = split(evalStack, 1)$ in
      if $val \neq \texttt{null}$ then
        $evalStack := evalStack' \cdot [fieldAdr(val, C::F \circ inst(meth))]$
        $pc := pc + 1$
      else $\textsc{Raise}(\texttt{NullReferenceException})$
  $MkRefAny(T) \circ inst(meth) \rightarrow$ let $(evalStack', [adr]) = split(evalStack, 1)$ in
      let $tr = new(TypedRef)$ in
        $typedRefAdr(tr) := adr$
        $typedRefType(tr) := T \circ inst(meth)$
        $evalStack := evalStack' \cdot [tr]$
        $pc := pc + 1$
  $Box(T) \circ inst(meth) \rightarrow pc := pc + 1$
      if $T \circ inst(meth) \in ValueType$ then
        let $(evalStack', [val]) = split(evalStack, 1)$ in
          if $T \circ inst(meth) \in NullableType$ then
            $\textsc{BoxNullable}(val, T \circ inst(meth))$
          else let $ref = NewBox(val, T \circ inst(meth))$ in
            $evalStack := evalStack' \cdot [ref]$
  $Unbox(T) \circ inst(meth) \rightarrow$ let $(evalStack', [ref]) = split(evalStack, 1)$ in
      if $T \circ inst(meth) \in NullableType$ then
        $\textsc{UnboxNullable}(ref, T \circ inst(meth), True)$
      elseif $ref \neq \texttt{null} \wedge actualTypeOf(ref) \preceq T \circ inst(meth)$ then
        $evalStack := evalStack' \cdot [addressOf(ref)]$
        $pc := pc + 1$
      elseif $ref = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
      else $\textsc{Raise}(\texttt{InvalidCastException})$
```

```
$lookUp(T, T'::M\langle T_0, \dots, T_{n-1}\rangle) =$
  if $T$ declares a non-abstract method $M \wedge$
    $((T = T') \vee (T' \in ObjClass \Rightarrow T::M$ overrides $T'::M)) \wedge$
    $(T' \in Interface \setminus GenericType \Rightarrow T::M$ implements $T'::M) \wedge$
    $(T' = I\langle U_0, \dots, U_{m-1}\rangle \in Interface \cap GenericType \Rightarrow$
      $T::M$ implements $I\langle V_0, \dots, V_{m-1}\rangle::M \wedge$
        $\forall j = 0, m-1 \; : \; (var_j^{I'm} = + \Rightarrow V_j \preceq U_j) \wedge$
                          $(var_j^{I'm} = - \Rightarrow U_j \preceq V_j) \wedge$
                          $(var_j^{I'm} = \emptyset \Rightarrow U_j = V_j))$
  then $T::M\langle T_0, \dots, T_{n-1}\rangle$
  elseif $T$ = object then $undef$
  else $lookUp(T'', T'::M\langle T_0, \dots, T_{n-1}\rangle)$ where $T''$ is the direct base class of $T$
```

The most significant change occurs when *lookUp* is applied to a method declared by a generic interface. In such a case, the method selected by *lookUp* may not override or implement the original method. As one can see in Example 4.3.6, this aspect makes type safety a concern.

As stated at the end of Section 4.3.1, the generic method references have the signatures uninstantiated. Therefore, for example, the return type (included in the signature) of a method which overrides/implements another method may not match the return type (included in the signature) of the overridden/implemented method. To cover this aspect, the conditions **[override/implement]**, defined in Section 4.1.2, are refined below. Moreover, **[override/implement]** now also includes the conditions that should be satisfied by the constraint types of the overriding/implementing method and overridden/implemented method.

[override/implement] If $T::M \in MRef$ overrides/implements $T'::M \in MRef$, then:

(at) $argNo(T::M) = argNo(T'::M)$ and for every $i = 1, argNo(T::M) - 1$,

$$argTypes(T::M)(i) \circ inst(T::M) = argTypes(T'::M)(i) \circ inst(T'::M)$$

(rt) $retType(T::M) \circ inst(T::M) = retType(T'::M) \circ inst(T'::M)$

(ct) $genParamNo(T::M) = genParamNo(T'::M)$ and for every $j = 0, genParamNo(T'::M) - 1$,

$$constr_j^{T::M} \text{ is not defined or } constr_j^{T'::M} \preceq constr_j^{T::M}$$

Remark 4.3.2 *Concerning* **(ct)**, [45, Partition II, §9.9] *states that any constraint type specified by the overriding/implementing method should be "no more restrictive" than the corresponding constraint type specified in the overridden/implemented method. However, this does not match the .NET Framework (v2.0) implementation* [3]*, where the bytecode verification checks whether one of the following conditions is satisfied: either the constraint type in the overriding method is not defined, i.e., as it would have been* `object`*, or the (assumably instantiated) constraint types coincide.*

Figure 4.24 The refined operational semantics rules (continued).

```
EXECCLR$(instr) \equiv$ match $instr$
  ⋮
  $Unbox.Any(T) \circ inst(meth)$ $\rightarrow$ let $(evalStack', [ref]) = split(evalStack, 1)$ in
      if $T \circ inst(meth) \in ValueType$ then
        if $T \circ inst(meth) \in NullableType$ then
          UNBOXNULLABLE$(ref, T \circ inst(meth), False)$
        elseif $ref \neq$ null $\wedge\ actualTypeOf\,(ref) = T \circ inst(meth)$ then
          $evalStack := evalStack' \cdot [memVal(addressOf\,(ref), T \circ inst(meth))]$
          $pc := pc + 1$
        elseif $ref =$ null then RAISE(NullReferenceException)
        elseif $actualTypeOf\,(ref) \neq T \circ inst(meth)$ then
          RAISE(InvalidCastException)
      elseif $ref =$ null $\vee\ actualTypeOf\,(ref) \preceq T \circ inst(meth)$ then
        $pc := pc + 1$
      elseif RAISE(InvalidCastException)
  $RefAnyVal(T) \circ inst(meth)$ $\rightarrow$ let $(evalStack', [tr]) = split(evalStack, 1)$ in
      if $T \circ inst(meth) = typedRefType(tr)$ then
        $evalStack := evalStack' \cdot [typedRefAdr(tr)]$
        $pc := pc + 1$
      else RAISE(InvalidCastException)
  $CastClass(T) \circ inst(meth)$ $\rightarrow$ let $ref = top(evalStack)$ in
      if $ref =$ null $\vee\ actualTypeOf\,(ref) \preceq T \circ inst(meth)$ then
        $pc := pc + 1$
      else RAISE(InvalidCastException)
  $IsInstance(T) \circ inst(meth)$ $\rightarrow$ let $(evalStack', [ref]) = split(evalStack, 1)$ in
      $pc := pc + 1$
      if $ref \neq$ null $\wedge\ actualTypeOf\,(ref) \npreceq T \circ inst(meth)$ then
        $evalStack := evalStack' \cdot$ [null]
  $LoadFtn(T::M) \circ inst(meth)$ $\rightarrow$ let $mp = new(MethPtr, T::M \circ inst(meth))$ in
      $evalStack := evalStack \cdot [mp]$
      $pc := pc + 1$
  $LoadVirtFtn(T::M) \circ inst(meth)$ $\rightarrow$ let $(evalStack', [ref]) = split(evalStack, 1)$ in
      if $ref \neq$ null then
        let $T' = actualTypeOf(ref)$ in
          let $mp = new(MethPtr, lookUp(T', T::M \circ inst(meth)))$ in
            $evalStack := evalStack' \cdot [mp]$
            $pc := pc + 1$
      else RAISE(NullReferenceException)
```

Let us now look in detail at the *Constrained* prefix, permitted only on a *CallVirt* instruction. The $Constrained(T'').CallVirt(T', T::M)$ instruction[30] calls the virtual method $T::M$, of return type T', on a value of a generic parameter T''. It pops the necessary number of arguments from the *evalStack*. The first value is expected to be a pointer, evaluated to an address, say *adr*. There are three cases that can occur. If T'' is instantiated with a reference type, then *adr* is dereferenced and passed as the `this` pointer. If T'' is instantiated with a value class that implements $T::M$, then *adr* is passed as the `this` pointer to the invocation of the method implemented by the

[30]A *Constrained* call cannot be tail as at most one prefix is allowed on the *CallVirt* instructions.

Figure 4.25 Example: constrained call.

```
class C⟨T⟩                .method public hidebysig instance void M() cil managed
{                         {
   public T F;                0: ldarg         0
   public void M()            1: ldflda        !0 class C'1⟨!0⟩::F
   {                          2: constrained. !0
     F.ToString();            3: callvirt      instance string System.Object::ToString()
   }                          4: pop
}                             5: ret
                          }
```

value class. If T'' is instantiated with a value class which does not implement $T::M$[31], then *adr* is dereferenced, boxed, and passed as the `this` pointer to the virtual call of $T::M$. Note that, in the macro VIRTCALL, the function *lookUp* is applied to $T::M \circ inst(meth)$, *i.e.*, to the method $T::M$ where *only* the generic parameters occurring in T and possibly in the generic argument list of M are replaced by the generic arguments indicated in $inst(meth)$.

The *Constrained* prefix has been introduced to support the compilers generating generic code. As the *CallVirt* instruction is not allowed on value classes, the compilers should effectively perform the `this` pointer transformation outlined above at compile time, depending on the type of the pointer and the method being called. One cannot do this transformation at compile time, however, when the pointer is a generic type (which is obviously unknown at compile time). The *Constrained* prefix is needed to get rid of this inconvenience: It allows compilers to make a virtual call in an uniform way, *i.e.*, regardless of whether the pointer is a value type or reference type.

Example 4.3.5 *Figure 4.25 contains a C♯ generic class and the bytecode for its method, where the virtual call F.`ToString`() is compiled into a CallVirt instruction prefixed by Constrained.*

4.3.3 The Bytecode Verification

In this section, we refine the bytecode verification specification to accommodate the generics. Section 4.3.3.1 specifies the restrictions on the verification type system, required to guarantee type safety. The verification specific functions *check* and *succ* are refined in Sections 4.3.3.2 and 4.3.3.3, respectively.

4.3.3.1 The Verification Type System

The verification types are described by the same universe *VerificationType* in Table 4.9, where the universe *Type* has been refined as shown in Table 4.16. The compatibility relation for verification types is refined as follows:

Definition 4.3.4 (Compatible verification types) *The relation $\sqsubseteq$, for verification types, is the least reflexive and transitive relation such that*

[31]This is the case when the method to be invoked is declared by the base class of the value class, *i.e.*, `System.ValueType`, or `object`, and not overridden by the value class.

- *if T is Null and $T' \in ObjClass \cup Interface \cup BoxedType$, or*
- *if $T \in ObjClass \cup Interface \cup BoxedType$ and T' is* `object`*, or*
- *if $T \in BoxedType$ and T' is* `System.ValueType`*, or*
- *if $T \in ObjClass \setminus GenericType$ and T' is a base class of T or an interface implemented by T, or*
- *if $T \in Interface \setminus GenericType$ and T' is an interface implemented by T, or*
- *if T is boxed(T''), where $T'' \in ValueType \setminus GenericType$ and T' an interface implemented by T'', or*
- *if T is boxed(X), where X is a generic parameter constrained by T', or*
- *if T is $C\langle U_0, \ldots, U_{n-1}\rangle$, where $C'n$ is a generic object class with the generic parameters $(X_i)_{i=0}^{n-1}$, T'' is a base class of $C'n$ or an interface implemented by $C'n$, and T' is $T'' \circ [U_i/X_i]_{i=0}^{n-1}$, or*
- *if T is $I\langle U_0, \ldots, U_{n-1}\rangle$, where $I'n$ is a generic interface with the generic parameters $(X_i)_{i=0}^{n-1}$, T'' is an interface implemented by $I'n$, and T' is $T'' \circ [U_i/X_i]_{i=0}^{n-1}$, or*
- *if T is boxed$(C\langle U_0, \ldots, U_{n-1}\rangle)$, where $C'n$ is a generic value class with the generic parameters $(X_i)_{i=0}^{n-1}$, T'' is an interface implemented by $C'n$, and T' is $T'' \circ [U_i/X_i]_{i=0}^{n-1}$, or*
- *if T is $I\langle U_0, \ldots, U_{n-1}\rangle$ and T' is $I\langle V_0, \ldots, V_{n-1}\rangle$, and for every $i = 0, n-1$ the following conditions hold:*
  - *if $var_i^{I'n} = \emptyset$ or $V_i \in ValueType$ or $V_i \in GenericParam$, then $U_i = V_i$;*
  - *if $var_i^{I'n} = +$, then $U_i \preceq V_i$;*
  - *if $var_i^{I'n} = -$, then $V_i \preceq U_i$;*

then $T \sqsubseteq T'$.

Definition 4.1.8 of compatible delegate classes and method references is refined as a result of the convention that *argTypes* and *retType* are uninstantiated.

Definition 4.3.5 *A delegate class DC is* compatible *with a method mref if the following conditions are matched:*

- *$argTypes(DC::$*`Invoke`*$)(i) \sqsubseteq argTypes(mref)(i) \circ inst(mref)$, for every $i = 1, argNo(mref) - 1$*
- *$retType(DC::$*`Invoke`*$) = retType(mref) =$* `void` *or $retType(mref) \circ inst(mref) \sqsubseteq retType(DC::$*`Invoke`*$)$.*

It is generally believed that one has to use contravariance when (static) type safety is required, but that covariance is more natural, flexible and reflects the program designer's natural wish to model the specialization of the classes. Although it is loosing some expressiveness, the CLR manages to combine the two approaches in a way that, as we will prove in Section 4.3.4, guarantees type safety. Thus, [45] imposes several requirements on the instance methods declared by a generic interface which is co-/contra-variant in at least one generic parameter. The requirements should ensure that an overriding method follows the *substitution principle*, *i.e.*, it has a "covariant" return type, "contravariant" parameter and constraint types. The usage of a "covariant" parameter type, as opposed to a "contravariant" one, cannot be statically verified (see Example 4.3.6) and run-time checks should have been inserted. This is, however, among the issues the generics design was supposed to get rid of.

The above requirements are captured in Definition 4.3.9. This definition necessitates Definitions 4.3.7 and 4.3.8, which in turn require the notion of *negation of a variances array* $(var_i)_i$:

Definition 4.3.6 (Negated variances array) *The negation $-(var_i)_i$ of an array $(var_i)_i$ of variances is defined by*

$$-var_i = \begin{cases} + , & \text{if } var_i = - \\ - , & \text{if } var_i = + \\ \emptyset , & \text{if } var_i = \emptyset \end{cases}$$

The notion of a *valid type with respect to a variances array* is specified as follows.

Definition 4.3.7 (Valid type) *The predicate validType checks the validity of a type T' with respect to an array (var_i) of variances.*

$$\begin{array}{l}
validType(T', (var_i)) :\Leftrightarrow \\
\quad T' \notin GenericType \cap GenericParam \ \vee \\
\quad T' \text{ is a generic parameter } \ !!j \text{ of the enclosing method } \ \vee \\
\quad T' \text{ is a generic parameter } !j \text{ of the enclosing type } \wedge \ var_j \in \{+, \emptyset\} \ \vee \\
\quad T' = T\langle U_0, \ldots, U_{n-1}\rangle \in GenericType \ \wedge \\
\qquad \forall k = 0, n-1 \ : \\
\qquad\quad (var_k^{T'n} = + \Rightarrow validType(U_k, (var_i))) \wedge \\
\qquad\quad (var_k^{T'n} = - \Rightarrow validType(U_k, -(var_i))) \wedge \\
\qquad\quad (var_k^{T'n} = \emptyset \Rightarrow validType(U_k, (var_i)) \wedge validType(U_k, -(var_i)))
\end{array}$$

Remark 4.3.3 [45, Partition II, §9.7] *states that $T' = T\langle U_0, \ldots, U_{n-1}\rangle$ in the above definition should refer to a "closed" generic type. This does not make a lot of sense, since in this case the definition has nothing to do with the array of variances. This remark and the experiments we have run with CLR indicate that T' should not necessarily be a closed type.*

Definition 4.3.8 stipulates when *a method is valid with respect to a variances array*. A method is valid if its return type “behaves covariantly”, whereas its argument types and possibly constraint types “behave contravariantly”. In other words, the covariant generic parameters can only appear in ”getter” positions in the type definition, *i.e.*, in return types of methods, the contravariant generic parameters can only appear in ”setter” positions in the type definition, *i.e.*, in argument and constraint types of methods, and non-variant generic parameters can appear anywhere.

Definition 4.3.8 (Valid method) *The predicate validMeth checks the validity of the declaration of the method mref with respect to the array* (var_i) *of variances.*

$$\begin{aligned}
&validMeth(mref, (var_i)) :\Leftrightarrow\\
&\quad validType(retType(mref), (var_i)) \ \wedge\\
&\quad \forall j = 1, argNo(mref) - 1 \ : \ validType(argTypes(j), -(var_i)) \ \wedge\\
&\quad \forall j = 0, genParamNo(mref) - 1 \ : \ validType(constr_j^{mref}, -(var_i))
\end{aligned}$$

Finally, the declaration of a generic interface is *valid* if all the instance methods declared by the interface and all the implemented interface types are valid with respect to the variances array of the given interface.

Definition 4.3.9 (Valid interface declaration) *The predicate validDecl checks the validity of the generic interface* $I'n$ *declaration.*

$$\begin{aligned}
&validDecl\,(I'n) :\Leftrightarrow\\
&\quad \forall I'n{::}M \in MRef \ : \ validMeth(I'n{::}M, (var_i^{I'n})) \wedge\\
&\quad \forall J \text{ implemented by } I'n \ : \ validType(J, (var_i^{I'n}))
\end{aligned}$$

Remark 4.3.4 [45, Partition II, §9.7] *is unclear about the definition of valid interface declarations. It requires that “every inherited interface declaration” should be valid with respect to the variances array. Firstly, the interfaces are not “inherited”, but implemented. Secondly, it is not about the interface “declaration”, but about the interface type present in the* `extends` *clause of the given interface.*

Example 4.3.6 *Consider the bytecode skeleton in Figure 4.26. The generic interface* $I'1$ *is covariant on its generic parameter:* $var^{I'1} = [+]$. *By this and* $OC' \preceq OC$, *we have* $I\langle OC'\rangle \preceq I\langle OC\rangle$ *according to Definition 4.3.1. On the other hand, by the same definition, we get* $OC'' \preceq I\langle OC'\rangle$.

Let us now assume that $I\langle OC\rangle{::}M$ *is called with the* `this` *pointer of type* OC'' *(allowed since* $OC'' \preceq I\langle OC'\rangle \preceq I\langle OC\rangle$*) and the argument of type* OC. *The method invoked at runtime is* $OC''{::}M$ *since* $lookUp(OC'', I\langle OC\rangle{::}M) = OC''{::}M$. *This means that, the argument of type* OC *is used in* $OC''{::}M$ *as an object of type* OC'*: see, for example, the access at index* 5 *of the field* $OC'{::}F$. *This is definitely unsafe. Fortunately, such an example is rejected by the bytecode verification as the declaration of* $I'1{::}M$ *is not valid according to Definition 4.3.8: The parameter type* `!0` *is not a valid type with respect to* $var^{I'1} = [+]$.

Figure 4.26 Example: unsafe example rejected by the verification.

```
.class interface private abstract auto ansi I'1⟨+X⟩
{
       .method public hidebysig newslot abstract virtual
          instance void M(!0 k) cil managed
       {
       }
}


.class private auto ansi OC extends System.Object
{
       ...
}


.class private auto ansi OC' extends class OC
{
       .field public int32 F
       ...
}


.class private auto ansi OC'' implements class I'1⟨ class OC'⟩
{
       .method virtual instance void M (class OC' i)
       {
               ...
               4 : ldarg 1
               5 : ldfld int32 OC'::F
               ...
       }
}
```

4.3.3.2 Verifying the Bytecode Instructions

The specification by means of *check* of the type-consistency checks is refined in Figure 4.27. Note that the *check* predicate is applied to instructions that might involve open generic types, in particular generic parameters, as the verification obviously occurs before the run-time specialization of generics.

Following the technical convention regarding the definition of *paramTypes* (and implicitly affecting also *argTypes*), stipulated in Section 4.3.2, the definition of *check* for the instructions *Call* and *CallVirt* is refined accordingly. Concerning the definition of *check* for an instruction $\mathit{CallVirt}(\_, T::M)$ prefixed by $\mathit{Constrained}(T'')$, the following checks are performed. The stack must contain a managed pointer to a value of type T''. Additionally, all the verification rules of the *CallVirt* instruction apply after the `this` pointer transformation as described above. That is equivalent to demanding that $\mathit{boxed}(T'')$ must be a subtype of T.

Figure 4.27 The refined instruction verification.

```
$check(meth, pos, argZeroT, evalStackT) :\Leftrightarrow$
  match $code(pos)$
    ⋮
    $Call(tail, \_, T::M)$ → (if $tail$ then $length(evalStackT) = argNo(T::M)$
                             else $\neg underflow(evalStackT, argNo(T::M))) \wedge$
                             let $[T'] \cdot types = take(evalStackT, argNo(T::M))$ in
                               $types \sqsubseteq_{suf} paramTypes(T::M) \circ inst(T::M) \wedge$
                               if $T \in ObjClass \wedge M =$ .ctor then $initCtorCompat(meth, T', T)$
                               else $argZeroCompat(T', T)$
    $CallVirt(tail, \_, T::M)$ → if $tail$ then $evalStackT \sqsubseteq_{len} argTypes(T::M) \circ inst(T::M)$
                             else $evalStackT \sqsubseteq_{suf} argTypes(T::M) \circ inst(T::M)$
    $Constrained(T'')$.
    $CallVirt(\_, T::M)$ → $boxed(T'') \sqsubseteq T \wedge$
                             $evalStackT \sqsubseteq_{suf} [T''\&] \cdot [argTypes(T::M)(i) \circ inst(T::M)]_{i=1}^{argNo(T::M)-1}$
```

Figure 4.28 The refined computation of type state successors.

```
$succ(meth, pos, argZeroT, evalStackT) =$
  match $code(pos)$
    ⋮
    $Call(tail, T', T::M)$ → if $tail$ then $\emptyset$
                             else let $evalStackT' = drop(evalStackT, argNo(T::M)) \cdot void(T' \circ inst(T::M))$ in
                               if $M =$ .ctor then
                                 let $evalStackT'' = evalStackT' \circ [classNm(meth)/UnInit]$ in
                                   $\{(pos+1, classNm(meth), evalStackT'')\}$
                               else $\{(pos+1, argZeroT, evalStackT')\}$
    $CallVirt(tail, T', T::M)$ → if $tail$ then $\emptyset$
                             else
                               $\{(pos+1, argZeroT, drop(evalStackT, argNo(T::M)) \cdot void(T' \circ inst(T::M)))\}$
    $Constrained(\_)$.
    $CallVirt(T', T::M)$ → $\{(pos+1, argZeroT, drop(evalStackT, argNo(T::M)) \cdot void(T' \circ inst(T::M)))\}$
```

4.3.3.3 Computing Successor Type States

Figure 4.28 contains the refinement of the function *succ*. Similarly as the *check* function, *succ* is refined for the instructions *Call* and *CallVirt*. The definition of *succ* for a *CallVirt* instruction prefixed by *Constrained* is the same as that for a *CallVirt* instruction with no prefix, *i.e.*, with neither *Tail* nor *Constrained*.

4.3.4 Type Safety

In this section, we prove that the bytecode language with generics is type safe. Besides the properties guaranteed so far for well-typed methods, we also consider and prove two generic specific type safety properties:

> ***Type safety***: The generic arguments of any generic type or method satisfy the corresponding declared constraints.

The definition of well-typed methods in Section 4.1.5 Definition 4.1.9 remains unmodified, assuming that the functions *check* and *succ* are refined, as indicated in Sections 4.3.3.2 and 4.3.3.3, respectively.

In the sequel, we assume that, besides the conditions on the type system and bytecode structure stipulated in Sections 4.1.4.2 and 4.2.4.1, the following are required to hold:

- every method should be well-typed in the sense of Definition 4.1.9;
- every generic interface declaration should be valid in the sense of Definition 4.3.9;
- the generic arguments of all the generic types and method references should satisfy at verification time the corresponding constraints. In particular, this means that every generic argument used in the *inst* functions satisfies the corresponding constraints.

Let us now revisit the potential risks for type safety, informally described at the beginning of Section 4.3, involved with the support for generics. Thus, although the signature and constraint types of a virtual called method may not match the signature and constrained types of the invoked method, respectively, type safety is preserved through a conservative assumption, namely by restricting the set of interfaces to those with valid declarations. The validity of the run-time specialization of generic types/methods, in the sense that the generic arguments satisfy the correspond constraints, is proved in Proposition 4.3.1, based on the definition of the verification type compatibility $\sqsubseteq$.

Of particular interest is the mismatch between the run-time type and the verification type which occurs upon boxing a nullable value. Consider that a *Box* instruction has a generic parameter `!0` in the scope as a type token argument. So, the value being boxed should be of some type `!0` (perhaps declared by the enclosing method) that cannot be known at verification time. The verification rules state that the result of *Box*(`!0`) should be of type *boxed*(`!0`) (see the definition of *succ* for the *Box* instruction). But, as noticed in Section 4.3.2, if `!0` is instantiated with a nullable type `Nullable`$\langle T\rangle$ as the generic argument for the generic parameter `!0`, this would not be true. The run-time type *actualTypeOf* would indicate a boxed object of type T, yet the verification type indicates (indirectly) a boxed object of type `Nullable`$\langle T\rangle$. Although this looks as a *type hole*, type safety is maintained, in particular through the design of nullable types as value classes that do not implement interfaces. The key points are that the boxed types are *abstract types* (used by the bytecode verification) that cannot be referred to by name in the bytecode, and the supertypes of a boxed nullable type are supertypes of any boxed nullable type. Therefore, there is no instance member of the boxed nullable type representing the verification type predicted by the bytecode verification, which is not an instance member of the run-time type.

The type safety proof takes advantage of the following lemmas. Lemma 4.3.1 shows that the subtype relationship of a given type varies *directly* with the relationship of the covariant generic parameters and *inversely* with the relationship of the contravariant generic parameters.

Lemma 4.3.1 *If the types T, $(U_i)_{i=0}^{n-1}$, $(V_i)_{i=0}^{n-1}$ and the array $(var_i)_{i=0}^{n-1}$ of variances are such that validType$(T, (var_i)_{i=0}^{n-1})$ and*

$$\forall i = 0, n-1 \;:\; (var_i = + \Rightarrow V_i \sqsubseteq U_i) \wedge (var_i = - \Rightarrow U_i \sqsubseteq V_i) \wedge (var_i = \emptyset \Rightarrow V_i = U_i)$$

then $T \circ [V_i / \,!\, i]_{i=0}^{n-1} \sqsubseteq T \circ [U_i / \,!\, i]_{i=0}^{n-1}$.

Proof. By induction on the structure of the possibly generic type T. Definition 4.3.7 is also applied. □

Lemma 4.3.2 proves that, if $T' \sqsubseteq T''$, then the instantiation of the generic parameters (corresponding to an enclosing generic class and/or method) occurring in T' and T'' with generic arguments satisfying the corresponding constraints preserves the subtype relation between T' and T''.

Lemma 4.3.2 *Let T' and T'' be two types such that $T' \sqsubseteq T''$. Assume that they occur in the declaration of a generic type $T'n$, possibly in the declaration of a generic method $T'n{::}M$ with m generic parameters. Let $(U_i)_{i=0}^{n-1}$ and $(V_j)_{j=0}^{m-1}$ be generic arguments for $T'n$ and $T'n{::}M$, respectively, assumed to satisfy the constraints:*

$$\begin{array}{ll} \mathit{boxed}(U_i) \sqsubseteq \mathit{constr}_i^{T'n} \circ [U_i / \,!\, i]_{i=0}^{n-1}, & \text{for every } i = 0, n-1 \\ \mathit{boxed}(V_j) \sqsubseteq \mathit{constr}_j^{T'n::M} \circ ([U_i / \,!\, i]_{i=0}^{n-1} \cdot [V_j / \,!\,!\, j]_{j=0}^{m-1}), & \text{for every } j = 0, m-1 \end{array}$$

It then holds $T' \circ [U_i / \,!\, i]_{i=0}^{n-1} \cdot [V_j / \,!\,!\, j]_{j=0}^{m-1} \sqsubseteq T'' \circ [U_i / \,!\, i]_{i=0}^{n-1} \cdot [V_j / \,!\,!\, j]_{j=0}^{m-1}$.

Proof. By induction on the structure of the (possibly generic) type T'. Definition 4.3.4 is also applied. □

The following two lemmas relate the argument types, respectively the return type of the method of a virtual call, and the argument types, respectively the return type of the method determined through a lookup and invoked at run-time.

Lemma 4.3.3 *If $T''{::}M = \mathit{lookUp}(T, T'{::}M)$ holds for a type T, then*

- *If $T \in \mathit{ValueType}$, then*
  - *if $T'' \neq T$, then $\mathit{boxed}(T) \sqsubseteq \mathit{argTypes}(T''{::}M)(0) \sqsubseteq \mathit{argTypes}(T'{::}M)(0)$*
  - *if $T'' = T$, then $\mathit{argTypes}(T''{::}M)(0) = T\&$ and $\mathit{boxed}(T) \sqsubseteq \mathit{argTypes}(T'{::}M)(0)$*

  If $T \in \mathit{RefType}$, then $T \sqsubseteq \mathit{argTypes}(T''{::}M)(0) \sqsubseteq \mathit{argTypes}(T'{::}M)(0)$
- *$\mathit{argNo}(T'{::}M) = \mathit{argNo}(T''{::}M)$ and for every $i = 1, \mathit{argNo}(T'{::}M) - 1$*

$$\mathit{argTypes}(T'{::}M)(i) \circ \mathit{inst}(T'{::}M) \sqsubseteq \mathit{argTypes}(T''{::}M)(i) \circ \mathit{inst}(T''{::}M)$$

Proof. By Definitions 4.3.8, 4.3.9, and 4.3.3, Lemmas 4.3.1 and 4.3.2, and **(at)**. □

Lemma 4.3.4 *If $T''{::}M = \mathit{lookUp}(T, T'{::}M)$ holds for a type T, then*

$$\mathit{retType}(T''{::}M) \circ \mathit{inst}(T''{::}M) \sqsubseteq \mathit{retType}(T'{::}M) \circ \mathit{inst}(T'{::}M)$$

Proof. By Definitions 4.3.8, 4.3.9, and 4.3.3, Lemmas 4.3.1 and 4.3.2, and **(rt)**. □

Lemma 4.3.5 shows that a generic method determined through a lookup has the same generic arguments as the original method and the generic arguments satisfy the corresponding constraints.

Lemma 4.3.5 *If $T''::M$ and $T'::M$ are some generic methods and T is some type such that $T''::M = lookUp(T, T'::M)$ and $boxed(genArg(T'::M)(i)) \sqsubseteq constr_i^{T'::M}$ holds for every $i = 0, genParamNo(T'::M) - 1$, then $genParamNo(T''::M) = genParamNo(T'::M)$, and for every $i = 0, genParamNo(T'::M) - 1$*

$$boxed(genArg(T'::M)(i)) = boxed(genArg(T''::M)(i)) \sqsubseteq constr_i^{T''::M}$$

Proof. By Definitions 4.3.8, 4.3.9, and 4.3.3, Lemmas 4.3.1 and 4.3.2, and **(ct)**. □

We assume that, if a generic type or method occurs instantiated or referenced, respectively, the generic arguments satisfy at verification time the corresponding constraints. As the generic arguments might be open generic types, the following question arise: *Do they satisfy the constraints also after the run-time instantiation?* The following proposition answers positively to this question.

Proposition 4.3.1 (Preservation of constraints under type substitution) *We assume that the (not necessarily closed) type $T\langle T_0, \ldots, T_{n-1}\rangle$ occurs in the declaration of a generic type $S'm$, possibly in the declaration of a generic method $S'm::M$ with p generic parameters. Moreover, assume that $(T_i)_{i=0}^{n-1}$ satisfy $T'n$'s constraints.*

If $S'm$ and $S'm::M$ are instantiated at run-time with the generic arguments $(U_j)_{j=0}^{m-1}$ and $(V_k)_{k=0}^{p-1}$, assumed to satisfy the constraints of $S'm$ and $S'm::M$, respectively, then $(T_i \circ \rho)_{i=0}^{n-1}$ satisfy $T'n$'s constraints, where ρ is the substitution $\rho = [U_j/\,!j]_{j=0}^{m-1} \cdot [V_k/\,!\,!k]_{k=0}^{p-1}$ defined in the scopes of $S'm$ and $S'm::M$.

A similar result also holds for a referenced generic method, i.e., $T::M\langle T_0, \ldots, T_{n-1}\rangle$ instead of $T\langle T_0, \ldots, T_{n-1}\rangle$.

Proof. For this proof, we assume that $constr^{T'n}$ denotes the uninstantiated (raw) constraints, exactly as given in the declaration of $T'n$. The following relations hold for every $i = 0, n-1$:

$$\begin{array}{rcl} boxed(T_i \circ \rho) & = & boxed(T_i) \circ \rho \\ & \sqsubseteq & (constr_i^{T'n} \circ [T_i/X_i]_{i=0}^{n-1}) \circ \rho \\ & = & constr_i^{T'n} \circ ([T_i/X_i]_{i=0}^{n-1} \circ \rho) \\ & = & constr_i^{T'n} \circ [T_i \circ \rho/X_i]_{i=0}^{n-1} \end{array}$$

The relation $\sqsubseteq$ in the second row above follows from Lemma 4.3.2 and the fact that $(T_i)_{i=0}^{n-1}$ satisfy $T'n$'s constraints. Finally, $(T_i \circ \rho)_{i=0}^{n-1}$ satisfy the constraints of $T'n$.

The proof for the case of a referenced generic method is similar. □

The definition of the typing judgment $\vdash$, given in Definition 4.1.12, remains unchanged. However, in the context of boxing nullable types, $\vdash$ does not suffice anymore. Therefore, we introduce a slightly different judgment $\models$ which, unlike $\vdash$, also considers the special cases when the dynamic and static types disagree.

Definition 4.3.10 *Let val be a value, T a verification type and $[fr_1, \dots, fr_n]$ a list of frames. The typing $[fr_1, \dots, fr_n] \models val : T$ holds if one of the following conditions is satisfied:*

- $[fr_1, \dots, fr_n] \vdash val : T$;
- *there exists $T' \in ValueType$ such that*

$$actualTypeOf(val) = T' \text{ and } boxed(\texttt{Nullable}\langle T'\rangle) \sqsubseteq T$$

Although Definition 4.3.10 does not explicitly include a *subsumption rule* for $\models$, this is implied and can be easily proved:

Lemma 4.3.6 *Let T and T' be two types such that $T \sqsubseteq T'$. If $[fr_1, \dots, fr_n] \models val : T$, then $[fr_1, \dots, fr_n] \models val : T'$.*

Proof. By Definition 4.3.10 and Lemma 4.1.5. □

The next lemma claims that $\vdash$ and $\models$ coincide for each type other than boxed nullable types.

Lemma 4.3.7 *If T is a type such that $[fr_1, \dots, fr_n] \models val : T$ and there exists no $T' \in ValueType$ such that $T = boxed(\texttt{Nullable}\langle T'\rangle)$, then $[fr_1, \dots, fr_n] \vdash val : T$.*

Proof. By *reductio ad absurdum*: Let us assume that $[fr_1, \dots, fr_n] \nvdash val : T$. From Definition 4.3.10, we derive that there exists $T'' \in ValueType$ such that

$$actualTypeOf\,(val) = T'' \text{ and } boxed(\texttt{Nullable}\langle T'\rangle) \sqsubseteq T \tag{4.15}$$

As the nullable types are value classes that do not implement interfaces, by Definition 4.3.4, we know that one of the following conditions should hold: $T = boxed(\texttt{Nullable}\langle T''\rangle)$ or $T = \texttt{object}$ or $T = \texttt{System.ValueType}$. The first conditions cannot be satisfied as, by the hypothesis, we know that there exists no $T' \in ValueType$ such that $T = boxed(\texttt{Nullable}\langle T'\rangle)$. If the second or the third condition holds, then, by (4.15), we have $[fr_1, \dots, fr_n] \vdash val : T$ according to Definition 4.1.12. Finally, the initial assumption is contradicted. □

As the generic methods have the return types uninstantiated, the notion *succession of a frame*, given in Definition 4.1.14, should be refined.

Definition 4.3.11 (Succession of a frame) *Let the list $[fr_1, \dots, fr_k]$ be the frameStack and $fr_i = (pc^*, locAdr^*, locPool^*, argAdr^*, evalStack^*, meth^*)$ be an arbitrary frame in the frameStack. We denote by pos and mref the program counter and the method of the callee frame of fr_i (for fr_k, the callee frame is frame). The succession of fr_i is defined as follows:*

- *If mref returns a value, say val, satisfying* $[fr_1, \ldots, fr_i] \vdash val : retType(mref) \circ inst(mref)$*, then* $(pc^* + 1, locAdr^*, locPool^*, argAdr^*, evalStack^* \cdot [val], meth^*)$ *is called* the succession *of* fr_i *in frameStack.*
- *If retType(mref) is* `void`*, then* the succession *of* fr_i *in frameStack is given by* $(pc^* + 1, locAdr^*, locPool^*, argAdr^*, evalStack^*, meth^*)$*.*

The type safety proof for well-typed methods is now extended to generics. We redefine the invariants which are affected by generics, and, additionally, we consider a generics specific invariant **(constr)**. The invariant **(stack2)** ensures that the values on the *evalStack* are of the types assigned in the stack state, unless the values are boxed values corresponding to appropriate boxed nullable types. Following the conventions that *locTypes* and *argTypes* are uninstantiated, the invariants **(loc)** and **(arg)** are reconsidered. The invariant **(constr)** ensures that the generic arguments of any generic method satisfy the declared constraints.

Theorem 4.3.1 (Type safety of well-typed methods) *Let* $[fr'_1, \ldots, fr'_{k-1}]$ *be the frameStack and* fr_k *be the current frame. For every* $i = 1, k-1$*, we denote by* fr_i *the succession of* fr'_i*.*

We assume that the type system and the bytecode structure of the methods of $(fr_i)_i$ *satisfy the conditions stated in Section 4.1.4.2 and Section 4.2.4.1. Moreover, we assume that every generic interface declaration is valid and the generic arguments of all the generic types and method references satisfy at verification time the corresponding constraints. Furthermore, we assume that all methods of* $(fr_i)_{i=1}^k$ *are well-typed.*

Let $fr_i = (pc^*, locAdr^*, \_, argAdr^*, evalStack^*, meth^*)$ *be one of the frames* $(fr_i)_{i=1}^k$*. We denote by* $locVal^*$ *and* $argVal^*$ *the functions locVal and argVal derived based on* $locAdr^*$ *and* $argAdr^*$*, respectively. If* fr_i *is the current frame, we consider* $initState^*$ *to be the current initialization status initState. If* fr_i *is not the current frame, we denote by* $initState^*$ *the succession of initState for* fr_i*. Let* $(argZeroT_j, evalStackT_j)_{j \in \mathcal{D}}$ *be the family of type states* $meth^*$ *is typable with.*

The invariants **(pc)***,* **(stack1)***,* **(init)***,* **(field)** **(box)** *and* **(del)** *of Theorem 4.1.1 and the following invariants are satisfied at run-time for every frame* fr_i *(for the current frame,* fr_k*, the invariants* **(stack1)** *and* **(stack2)** *should* only *hold if switch = Noswitch):*

(stack2) $[fr_1, \ldots, fr_i] \models evalStack^*(j) : evalStackT_{pc^*}(j) \circ inst(meth^*)$*, for every* $j = 0, length(evalStack^*) - 1$.

(loc) $[fr_1, \ldots, fr_i] \vdash locVal^*(n) : locTypes(meth^*)(n) \circ inst(meth^*)$*, for every* $n = 0, locNo(meth^*) - 1$.

(arg) *If* $classNm(meth^*) \in ObjClass$ *and* $methNm(meth^*) =$ `.ctor`*, then it holds* $[fr_1, \ldots, fr_i] \vdash argVal^*(0) : argZeroT_{pc^*}$*; otherwise,* $[fr_1, \ldots, fr_i] \vdash argVal^*(0) : argTypes(meth^*)(0)$.

If $argNo(meth^*) \geq 2$*, then* $[fr_1, \ldots, fr_i] \vdash argVal^*(n) : argTypes(meth^*)(n) \circ inst(meth^*)$*, for every* $n = 1, argNo(meth^*) - 1$.

(constr) *If $meth^*$ is a generic method, $boxed(genArg(meth^*)(n)) \sqsubseteq constr_n^{meth^*}$, for every $n = 0, genParamNo(meth^*) - 1$.*

Proof. The proof proceeds by induction on the run of the operational semantics model $\mathrm{CLR}_{\mathcal{G}}$. We only prove here the case of virtual method calls.

Case $code(meth)(pc) = CallVirt(T', T{::}M) \circ inst(meth)$.

Since *meth* is well-typed, we get $check(meth, pc, evalStackT_{pc})$ from Definition 4.1.9 **(wt5)**. According to the definition of *check* in Figure 4.27, $evalStackT_{pc} \sqsubseteq_{suf} argTypes(T{::}M) \circ inst(T{::}M)$. By **(constr)** and Lemma 4.3.2, we have [32]

$$evalStackT_{pc} \circ inst(meth) \sqsubseteq_{suf} (argTypes(T{::}M) \circ inst(T{::}M)) \circ inst(meth) \tag{4.16}$$

By (4.16) and by the induction hypothesis – that is, by the invariants **(stack1)** and **(stack2)** – there exists a list of values L, two lists of types L' and L'' and the values $(val_j)_{j=0}^{argNo(T::M)-1}$ such that: $evalStack = L \cdot [val_j]_{j=0}^{argNo(T::M)-1}$,

$$\begin{array}{l} evalStackT_{pc} = L' \cdot L'',\ length(L'') = argNo(T{::}M), \\ \text{for every } j = 0, length(evalStack) - argNo(T{::}M) - 1 \\ [fr_1, \ldots, fr_k] \models L(j) : L'(j) \circ inst(meth) \end{array} \tag{4.17}$$

and

$$\text{for every } j = 0, argNo(T{::}M) - 1 \qquad [fr_1, \ldots, fr_k] \models val_j : L''(j) \circ inst(meth) \tag{4.18}$$

By (4.16), (4.17), (4.18) and Lemma 4.3.6, $[fr_1, \ldots, fr_k] \models val_j : (argTypes(T{::}M)(j) \circ inst(T{::}M)) \circ inst(meth)$ for every $j = 0, argNo(T{::}M) - 1$. By this and Lemma 4.3.7, we get for every $j = 0, argNo(T{::}M) - 1$:

$$[fr_1, \ldots, fr_k] \vdash val_j : (argTypes(T{::}M)(j) \circ inst(T{::}M)) \circ inst(meth) \tag{4.19}$$

When the bytecode instruction $CallVirt(tail, T', T{::}M) \circ inst(meth)$ is executed, the macro $\mathrm{VIRTCALL}(tail, T{::}M \circ inst(meth), [val_j]_{j=0}^{argNo(T::M)-1})$ is invoked. According to [45, Partition I, §12.1.6.2.4], the *CallVirt* instruction is not valid with value class method token arguments. This implies $T \notin ValueClass$. By this and (4.19), we have that val_0 is an object reference.

We only analyze here the case when val_0 is not `null` and *tail* is *False*. If that is the case, in the frame of new current method $T''{::}M = lookUp(T''', T{::}M \circ inst(meth))$, where $T''' = actualTypeOf\,(val_0)$, the arguments $(argVal(j))_{j=0}^{argNo(T''::M)-1}$ are set to $(val_j)_{j=0}^{argNo(T::M)-1}$. Note that, by Lemma 4.3.3, we have $argNo(T{::}M) = argNo(T''{::}M)$.

Regarding $T''{::}M$, we only focus here on proving the invariants **(arg)** and **(constr)** as the proof of the others goes along the lines of the proof **Case 2** in Theorem 4.1.1. Taking into account Lemma 4.3.3, we make the following case distinction.

Let us first assume that $T''' \in ValueType$.

[32]Note that $evalStackT_{pc}$ might involve generic parameters.

- If $T'' \neq T'''$, then $\mathit{boxed}(T''') \sqsubseteq \mathit{argTypes}(T''::M)(0)$. According to the definition of VIRTCALL, the `this` pointer of $T''::M$ is val_0. By the definition of $\vdash$, we have $[\mathit{fr}_1, \ldots, \mathit{fr}_k] \vdash \mathit{val}_0 : \mathit{argTypes}(T''::M)(0)$. This and the relation $\mathit{argTypes}(T''::M)(0) = \mathit{argTypes}(T''::M)(0) \circ \mathit{inst}(T''::M)$ (the type of the `this` pointer is instantiated, as opposed to the other argument types) imply the invariant **(arg)** for the first argument of $T''::M$.

- If $T'' = T'''$, then $\mathit{argTypes}(T''::M)(0) = T'''\&$. Conforming with the definition of VIRTCALL, the first argument of $T''::M$ is $\mathit{addressOf}(\mathit{val}_0)$. By the definition of $\vdash$ and $\mathit{argTypes}(T''::M)(0) = T'''\&$, we get $[\mathit{fr}_1, \ldots, \mathit{fr}_k] \vdash \mathit{addressOf}(\mathit{val}_0) : \mathit{argTypes}(T''::M)(0)$. Similarly as above, this ensures the invariant **(arg)** for the `this` pointer of $T''::M$.

If $T''' \in \mathit{RefType}$, then $T''' \sqsubseteq \mathit{argTypes}(T''::M)(0)$. The first argument of $T''::M$ is val_0. By the definition of $\vdash$, we have $[\mathit{fr}_1, \ldots, \mathit{fr}_k] \vdash \mathit{val}_0 : \mathit{argTypes}(T''::M)(0)$. This implies the invariant **(arg)** for the `this` pointer of $T''::M$.

So, in all the cases, the invariant **(arg)** for the `this` pointer of $T''::M$ is ensured.

By Lemma 4.3.3, the following relations hold for every $j = 1, \mathit{argNo}(T::M) - 1$:

$$\begin{array}{rl} & (\mathit{argTypes}(T::M)(j) \circ \mathit{inst}(T::M)) \circ \mathit{inst}(\mathit{meth}) \\ = & \mathit{argTypes}(T::M \circ \mathit{inst}(\mathit{meth}))(j) \circ \mathit{inst}(T::M \circ \mathit{inst}(\mathit{meth})) \\ \sqsubseteq & \mathit{argTypes}(T''::M)(j) \circ \mathit{inst}(T''::M) \end{array} \tag{4.20}$$

The invariant **(arg)** for the other arguments follows from (4.19), (4.20), and Lemma 4.3.6.

Let us now prove **(constr)**. We assume that $T''::M$ is a generic method (otherwise **(constr)** is obviously satisfied). This implies that also the method $T::M$ is generic. By Lemma 4.3.5, we get $\mathit{genParamNo}(T''::M) = \mathit{genParamNo}(T::M)$ and for every $j = 0, \mathit{genParamNo}(T::M) - 1$:

$$\mathit{boxed}(\mathit{genArg}(T::M \circ \mathit{inst}(\mathit{meth}))(j)) = \mathit{boxed}(\mathit{genArg}(T''::M)(i)) \sqsubseteq \mathit{constr}_j^{T''::M}$$

Proposition 4.3.1 applied for the generic method $T::M$ and the above relations imply **(constr)**.

We also want to show **(stack2)** for the succession, defined according to Definition 4.3.11, of the new current *frame* in $[\mathit{fr}'_1, \ldots, \mathit{fr}'_k]$. If $\mathit{retType}(T''::M) =$ `void`, then, by Lemma 4.3.4, $\mathit{retType}(T::M) =$ `void`, and therefore **(stack2)** for the succession follows from the induction hypothesis, *i.e.*, **(stack2)** for fr_k.

Let us now assume that $\mathit{retType}(T''::M)$ is not `void`. Let *val* be the value returned by $T''::M$, assumed to satisfy

$$[\mathit{fr}'_1, \ldots, \mathit{fr}'_k, \mathit{fr}_k, \mathit{frame}] \vdash \mathit{val} : \mathit{retType}(T''::M) \circ \mathit{inst}(T''::M) \tag{4.21}$$

Lemma 4.1.7, the constraint ***Return Type*** and (4.21) imply

$$[\mathit{fr}'_1, \ldots, \mathit{fr}'_k, \mathit{fr}_k] \vdash \mathit{val} : \mathit{retType}(T''::M) \circ \mathit{inst}(T''::M) \tag{4.22}$$

To prove **(stack2)** for the succession of *frame*, it suffices to show

$$[\mathit{fr}'_1, \ldots, \mathit{fr}'_k, \mathit{fr}_k] \models \mathit{val} : \mathit{retType}(T::M) \circ \mathit{inst}(T::M) \tag{4.23}$$

By Lemma 4.3.4, it follows that

$$\mathit{retType}(T''{::}M) \circ \mathit{inst}(T''{::}M) \sqsubseteq \mathit{retType}(T{::}M) \circ \mathit{inst}(T{::}M) \tag{4.24}$$

Finally, (4.23), and implicitly also **(stack2)** for the succession of *frame*, follows from (4.22), (4.24), Lemma 4.1.5, and Definition 4.3.10. □

4.4 Related Work

Sections 4.4.1 and 4.4.2 survey the literature on CIL's formal studies, in particular type safety proofs, and the closely related *Java Virtual Machine* (JVM) [91]'s type safety, respectively. The application of the ASM method for the specification of programming languages and their implementation is reviewed in Section 4.4.4.

4.4.1 The CIL

Regarding CIL's type safety, there are only two publications we are aware of. Gordon and Syme have developed a *type system* for a small fragment of CIL, named Baby IL (*BIL*) [66]. Their work has been *partially extended* with *generics* by Yu, Kennedy and Syme for an even smaller CIL subset, called *BILG* [123]. We provide below a detailed comparison between these papers and our work.

BIL's type safety The main theorem proved by Gordon and Syme [66], based upon Syme's method [117] for writing functional specifications (subject to theorem proving in HOL), asserts soundness of the BIL's type system and not the soundness of the bytecode verification. Being focused on the type system, they develop simple evaluation rules, in a *big-step* style of operational semantics [85]. This approach has, however, some downsides. For example, it seems to be impossible to determine in their framework if and when a location (*e.g.*, a field or argument) is initialized.

Unlike [66], our framework involves a *complete* (with respect to the considered bytecode language) semantics model, which represents an important by-product of this thesis. To the best of our knowledge, our work is the *first* that defines a semantics model for the verifiable CIL bytecode.

We briefly mention here the most important aspects omitted in their formal specifications:

- *local variables*: As Gordon and Syme do not provide a formal description for a method *frame*, leaving out local variables simplifies considerably their approach. As we have seen in Definition 4.1.12, considering local variables increases the complexity of the typing judgment the type safety proof highly relies on.
- *arithmetic operations*: For the semantics of the arithmetic operations, our framework reflects the implementation of the real CLR, where the evaluation stack is *strongly* typed. Accordingly, we consider the values to be tagged (see Section 4.1.2). As Gordon and Syme abstract away from any details concerning the evaluation stack, the importance of a typed evaluation stack cannot be spotted in their framework.
- *typed references*: Since typed references embed pointers, the verification has to treat them prudently in order to avoid dangling pointers. It is not immediately obvious whether the restrictions imposed in Section 4.1.4.2 guarantee the type-safe execution of programs with typed references.
- *tail method calls*: Tail calls require care, because the verification has to prevent pointers to the current frame being passed as arguments. It is not straightforward at all whether the constraints on the bytecode structure defined Sections 4.1.4.2 and 4.2.4.1 ensure type-safe tail calls.

- *exceptions*: BIL is exception-free. One of our main contributions is the investigation of the fairly elaborate exception handling mechanism, which significantly increased the complexity of our work through the complicated control flow graph underlying the runs of the mechanism and through the nesting of the pass records. The analysis of the exception mechanism's runs necessitates attention since, as we have seen in Theorem 4.2.1, there are certain runs of the mechanism, characterized by $switch \neq Noswitch$, which include steps where the invariants **(stack1)** and **(stack2)** are *not* satisfied. However, these (hidden) steps do not affect CLR's type safety since the *evalStack*, involved in **(stack1)** and **(stack2)**, is not accessed as long as *switch* is not *Noswitch*.

- *delegates*: BIL does not include *method pointers*. Therefore, the delegates, *i.e.*, the feature that allows for type-safe method pointers, are not treated either. As [66] does not specify the program counter *pc*, it is not clear whether [66]'s approach can be easily adjusted to specify, for example, the constraint ***Delegate Pattern*** in Section 4.1.4.2. Moreover, BIL would need to be considerably extended (*e.g.*, *Dup*, *LoadFtn*, *LoadVirtFtn*, branch instructions) to cover the delegate handling.

- *boxed types* and *interfaces*: Although BIL includes instructions for boxing and unboxing, [66], contrary to [45], does not include boxed types. Moreover, without considering boxed types, [66]'s approach is not flexible enough to include interfaces.

The subtleties of the *local memory pool* are avoided by [66]'s framework. Actually, they *cannot* be spotted as long as the typing of the `this` pointer of a `.ctor` is not an issue in [66]. This is the case since the `.ctor`s considered in [66] are oversimplified and do not have a `this` pointer.

Gordon and Syme do not take into account details concerning the methods' *evaluation stack*. That means, on one hand, that **(stack1)** and **(stack2)** cannot be analyzed, and therefore cannot be ensured with their framework. On the other hand, as the exception mechanism deals extensively with evaluation stacks, it is imperatively necessary to specify the *evalStack* if one wants to include exceptions.

Another important drawback of [66]'s type system is that it does not analyze the special *initialization rules for objects*. In this sense, one of our merits is the extension of the verification type system by the type *UnInit*, though [45, Partition III, §1.8.1.2.1] does *not* mention it. Another important contribution of our framework is augmenting the bytecode verification with the *argZeroT* component to track type information concerning the initialization status of object references. As BIL does not include the *Return* instruction and assumes, as originally suggested by [76], that *each class has exactly one* `.ctor` (which does not even have a `this` pointer), it would just not be possible to implement the object initialization rules in [66]'s framework according to [45]. Recall that a `this` pointer of an object class `.ctor` is fully initialized if a constructor of *the same class* or of the base class has been invoked.

As the theorem proved by Gordon and Syme applies only to a small bytecode language, they write [66, §1, pag. 3]:

> *A soundness proof for the whole of CIL would be an impressive achievement.*

Given that we have only omitted a *very few*, *yet non-critical*, CIL features, we can certainly claim that this thesis has attained the respective achievement.

BILG's type safety Yu, Kennedy and Syme use judgment forms, in a *big-step* style operational semantics, to define the generics semantics, in a fashion similar to Featherweight GJ [78]. They focus on aspects of the *generics implementation*, *e.g.*, specialization of generic code up to data representation, efficient support for run-time types. As the exceptions are *not* part of BILG, [123] has to specify for certain constructs a different semantics than [45], *e.g.*, the [123]'s *IsInstance* instruction does not return – as it was supposed to – the `null` pointer if the run-time test fails. Moreover, as their main goal was not BILG's type safety, the formal specifications do not include the main sources for potential type safety violations: *variance on generic parameters* and *constraint types*. The critical issue concerning the mismatch between the static and dynamic types in case of boxing nullable types cannot be uncovered in [123]'s approach as long as the *Box* and *Unbox* instructions are not included. That means, in particular, that the *boxed types* are not considered either. Consequently, the constraint types cannot be added to their approach.

Bannwart and Müller [20] present a Hoare-style logic (*axiomatic semantics*) for a sequential kernel bytecode language, *similar* to CIL. The logic is part of a larger project that aims at developing a *proof-transforming compiler* (somehow similar to certifying compilers [36] in Proof-Carrying Code [96]), that translates a source program together with a proof of some of its properties to the bytecode level. Their logic considers that the bytecode programs are accepted by the bytecode verification. The logic is sound with respect to the defined operational semantics. They apply the logic in a weakest precondition style and show how source proofs can be translated to the bytecode logic. The bytecode language they consider does not include managed pointers, delegates, value classes, tail calls, typed references, and generics.

4.4.2 The JVM

The JVM is the run-time environment *designed* to execute compiled Java programs. Considerable research effort has been devoted to formally specify the JVM bytecode semantics, verification, and type safety; see, for example, [116, 52, 115, 114, 53, 51, 89, 103, 104, 99, 39]. A detailed review of the huge literature can been found in [73]. Of all these publications, the work by Stärk *et al.* [115] is the most closely related and had the most impact on this thesis. Besides proving Java's and JVM's type safety, [115] (following [113]) also points out a serious problem of the JVM bytecode verification: There are legal Java programs which are correctly compiled (with at that time Sun's JDK (v1.2) and (v1.3) compilers) into a JVM bytecode that is rejected by the bytecode verification (of JDK (v1.2) and (v1.3)). To fix this problem, [115] (following [114]) propose to restrict the definite assignment analysis rules of the Java compiler such that these programs are no longer allowed. Further, [115] (following [114]) specifies a compilation scheme and proves that the bytecode programs the Java compiler generates, according to the compilation scheme, from the *restricted subset* of valid Java programs pass the bytecode verification.

As the CLR is *designed* to support a large diversity of languages, it is understandable that it deals with many constructs that do not have an equivalent in JVM, which, despite of several attempts [93, 69, 1] to be used as target for languages other than Java, has been *designed* precisely with the Java language in mind. The most important CLR features that are lacking in JVM are the *tail method calls*, the constructs supporting *pointers* (including method pointers in the form of *delegates*), *value classes*, *boxing*, *unboxing*, and *typed references*. As detailed comparisons of the two virtual machines already exist [68, 108], we limit ourselves to point out here only a few important aspects concerning the semantics and bytecode verification that differ in JVM and CLR.

The main challenge of the JVM bytecode verification, *i.e.*, the *subroutine* verification, is not an issue in the CLR. Moreover, in the JVM, the *object initialization* verification is performed differently than in the CLR.

Subroutine verification A JVM subroutine can be viewed as a procedure at the bytecode level. If a sequence of bytecode instructions occurs more than once in a method body, the Java compiler can build a subroutine with this sequence and call it at the convenient positions. The main application of the subroutines is for the Java `finally` blocks. Every such a block is compiled to a subroutine. At each exit point inside a `try` block and its associated `catch` handler region, the subroutine corresponding to the `finally` handler is called. After the `finally` block completes – as long as it completes by executing past the last statement in the `finally`, not by throwing an exception or executing a return, continue, or break – the subroutine itself returns. The execution continues just past the point where the subroutine has been called in the first place, so the `try` block can be exited in the appropriate manner. One problem faced by the JVM bytecode verification when checking subroutines is concerning the *polymorphism on local variables*. The subroutines may have multiple call points, whose local variables may each hold values of different types. One should expect that the local variables not used inside the subroutine have the same type before and after the execution of the subroutine.

The CLR does not use subroutines to implement the `finally` handlers. The CLR's solution seems to be more efficient: As we have seen in Section 4.2.2, the CLR *explicitly* supports `finally` handlers in CIL. Moreover, the CLR, unlike JVM, has *typed* local variables. This approach does not have any downside, except for carrying around some metadata. However, as observed in Section 4.2, the CLR exception handling mechanism becomes very complex by supporting `filter` handlers. Another construct that has no equivalent in JVM is the CLR `fault` handler. A semantics model for the JVM exception handling mechanism has been defined in [26, 115].

Object initialization verification The JVM object creation process is accomplished in two steps as follows. A new reference is created by a `new` instruction (which does not have a constructor token argument), duplicated, and passed *explicitly* to a constructor invocation. One reference (to an *uninitialized* object) remains on the stack, and one reference is ”consumed” as the `this` pointer by the constructor invocation. After the constructor has been invoked, the reference left on the stack points to a properly initialized object. Bytecode verification is difficult since the constructors operate by side-effect: Instead of taking an uninitialized object and returning an initialized object, they simply take an uninitialized object, leave a reference to uninitialized object on the stack, and return *nothing*. To find out what references in the current state are ensured to point to the same uninitialized object that is passed as `this` pointer to a constructor, the JVM verification identifies every uninitialized object by the program counter of the `new` instruction that created it.

The CLR gets rid of this problem by adding a constructor token argument to the `newobj` instruction. Thus, in CLR, the same instruction creates a new object reference and initializes it by invoking the constructor pointed to by the token argument. However, in the constructor, the `this` pointer is considered uninitialized and the CLR verification has to track its initialization status.

Another important verification issue, the local variable initialization, is treated differently by JVM and CLR. While the CLR requires that the verifiable methods *automatically* zeroes the local variables, the JVM bytecode verification performs a definite assignment analysis, in the sense of the one presented in Chapter 3 Section 3.4, and tracks type information for every local variable.

Unlike CLR, the JVM does not have generic instructions. To encode the kind of the performed operation, prefixes are added to the JVM instructions. For example, the JVM instruction `iadd` performs the addition of two integers, whereas `dadd` adds two doubles. It is then not necessary for any JVM semantics model to consider *tagged* values.

The JVM does not provide support for generic types and generic methods though Java (v5.0) supports them. The design of the generics in Java, initially proposed in [29], takes a different approach than the CLR. More precisely, the generics feature is compiled ”away” by the Java compiler. This design has some drawbacks. For instance, generic types can only be instantiated with reference types and not with primitive types, and type information is not preserved at run-time as in the CLR.

4.4.3 The Variance on Generic Parameters

The variance on generic parameters has its origins in the POOL-I language presented by America and van der Linden [19]. It has produced a lot of research into the nature of the inheritance relationship and subtyping. Two excellent overviews of the corresponding literature can be found in [35, 79].

There have been proposals to add variance for Java. One of the first designs for generics in Java, the work of Cartwright and Steele Jr. [34] include *definition-site* variance annotations. Igarashi and Viroli formally specified a system of *use-site* variance [79]. The latter has been adopted in Java (v5.0) through the *wildcard* mechanism [120].

As the C$^\sharp$'s generic parameters are currently non-variant, Emir *et al.* [47] propose a system of type-safe variance for C$^\sharp$ that supports the declaration of covariant and contravariant parameters. They also introduce type constraints that generalize the *F-bounded polymorphism* [33] of Java [78, 76] and the bounded method generic parameters of Scala and include work on equational constraints [87].

A controversial feature of the Eiffel type system, the variance is in depth analyzed in [63], which also reports on a comprehensive comparison of generics in six programming languages (C++, Standard ML, Haskell, Eiffel, Java, C$^\sharp$). Cook [37] identifies some problems in Eiffel, in particular the mismatch of the called and invoked methods' signatures caused by the variance. Cook also proposes solutions to fix these problems. The solution the above problem is approximately the one that [45] adopted, namely methods are required to have return types behaving ”covariantly” and argument types behaving ”contravariantly”. Howard *et al.* [75] propose a solution which enables compilers to spot all potential run-time type violations caused by variance and forces programmer to resolve them. The resulting language rules statically guarantee type safety and only require local analysis. However, in the absence of a full semantics model for Eiffel, [75] cannot provide a formal proof that the proposed policy guarantees type safety.

4.4.4 The ASM Method

This section gives a short critical assessment of the ASM method and provides pointers to publications for further comparisons to alternative approaches. Besides reviewing the work on JVM's type safety, [73] also includes an evaluation of the ASM based investigations. As a broad comparison of the ASM method with respect to other system design and analysis frameworks has already been done in [21, 64], we only point here some advantages of the ASM method over [66]'s approach.

With Gordon and Syme [66]'s evaluation rules, it is not convenient at all to model irregular control flows, in particular exception handling[33]. To simplify the evaluation and typing rules, [66] decides to create two instructions which assemble certain branch instructions. As a result, a syntax for these two constructs is chosen. This technique is, however, not suitable when considering the whole CIL. As Gordon and Syme [66] mention, "*the technique may not scale well to express control flow such as arbitrary branching within a method and exception handling*". In particular, the *code containment*, *i.e.*, the invariant that *pc* will always point to a valid code index, cannot be analyzed, and therefore cannot be ensured in [66]'s framework. As demonstrated by this thesis, the ASM method can be successfully applied to model irregular control flows.

Another advantage of using the ASM method is the possibility of validating the corresponding ASM model. Thus, the model can be refined, made executable by means of an implementation in AsmL [49], and then validated by its simulation. To validate our operational semantics model, *i.e.*, to test its *internal correctness* and its *conformance* to the Microsoft implementation (v2.0) [3], we have developed an AsmL implementation [92] of our ASM specifications. Furthermore, we have also validated our specifications for the bytecode verification [50].

4.5 Summary and Future Work

Except for static members, arrays and multi-threading – features that do not raise any challenging problems – we have proved type safety of the whole CIL.

We have structured the CIL bytecode language into three bytecode sublanguages, by isolating orthogonal parts of CIL, namely lightweight CIL, exception handling, and generics. Each sublanguage in the sequence extends its predecessor. We proved type safety for each sublanguage as follows. To specify the static and dynamic semantics, for each sublanguage, we defined an ASM interpreter, which is a conservative (purely incremental) extension of its predecessor. We have then provided a formal specification of the static and dynamic consistency checks performed by the bytecode verification. The dynamic checks have been defined by stepwise refinement, analogously to the layering of the semantics models. On top of these checks, we developed precise specifications of the bytecode verification algorithm and of the well-typed methods. We then proved type safety of each sublanguage as follows. We first showed that the well-typed methods are type-safe, *i.e.*, the execution, according to the semantics models, of legal and well-typed methods does not yield any run-time type errors and maintains the program in a good state, where certain structural constraints are satisfied. Finally, we demonstrated that the bytecode verification is sound, *i.e.*, the accepted methods are well-typed, and complete, *i.e.*,

[33]Note that *denotational semantics* is not suitable either to model irregular control flows (see [81] for a solution to cope with this issue).

it accepts every well-typed method and computes a most specific family of type states.

Potential future work includes a few extensions of the bytecode verification. As the bytecode verification is conservative, it might reject non-well-typed programs though their execution is type safe, regardless any input values. To limit the number of these cases, we would like to refine the bytecode verification. One refinement could be, for example, to extend the verification such that it also verifies methods whose local variables are not automatically initialized. For that, the verification should perform a definite assignment analysis similar to the one performed by the JVM verification. Another possible refinement is concerning the method pointers. Thank to the strong constraint ***Delegate Pattern***, the bytecode verification can check the *NewObj* instructions which create delegates. This restriction can be weakened by inheriting the verification type system *VerificationType* with an *abstract type*. This special type would actually be defined as a pair consisting of the following components: a *verification type* (as defined before) and a *method signature*. The type component of the pair would be essential when verifying a delegate creation. Thus, for the *NewObj* instructions applied with delegate `.ctor`s, one would need check that the type of the target object on the stack is compatible with the type component and the delegate class is compatible with the signature component. This extension of the verification type system would also make possible the verification of the currently non-verifiable indirect calls, *i.e.*, calls via method pointers.

Future work might also encompass a formal study of the .NET security model and its properties, but also a correctness proof of a compilation scheme C$^\sharp$–CIL, similar to the Java–JVM scheme developed in [115]. Also, the automatic verification of our type safety result remains an important challenge for future research.

5 Conclusion

Work for something because it is good, not just because it stands a chance to succeed.
Václav Havel

Until a while ago, C and C++ programming were the real standard in industry, and the types, if present, were not much more than ways to name memory offsets. For instance, a C structure is really just a big sequence of bits with names to access precise offsets, *i.e.*, fields, from the base address. References to structures could be used to point at incompatible instances, data could be indexed into and used freely. Although C++ was already an important step in the right direction, there generally was not any run-time system enforcing that memory access follows the type system rules at run-time. In unmanaged languages, like C and C++, one can easily violate type safety, with direct security implications, through the possibility to easily access memory unimpeded. As this programming approach has proven to be error prone by leading to hard bugs, a trend toward type-safe languages has been started. Thus, over time, type-safe languages, like C$^\sharp$ and Java, employing a static type analysis and dynamic detections of operations that could lead to type errors, gained popularity. Given the areas of applications these languages are designed for, it is crucial to guarantee that they meet the safety-related expectations, *i.e.*, they are indeed type-safe.

In this thesis, we have formally proved type safety of large and representative subsets of C$^\sharp$ and of the CIL language to which any .NET compliant language, in particular C$^\sharp$, compiles. The analyzed subsets are reasonably rich and contain all the features which separately or together might have led to violations of type safety. It must be emphasized that all the formal models the proofs rely on are not only executable in theory: They have been validated through AsmL prototypes [84, 92, 50] generated from the formal specifications.

The abstract frameworks in this thesis have made it possible to clarify a number of important aspects which are ambiguously specified or not handled at all by the official specifications of C$^\sharp$ and CLR. In this sense, we pointed out and filled several gaps in the two official documents, and we indicated a number of inconsistencies between different implementations of C$^\sharp$ and CLR and their official specifications. It is important to notice that most of the conditions inherited from the language specifications, in particular even the well-formedness conditions, are actually needed somewhere in the proofs. This inspires, in particular, confidence in the appropriateness of the developed abstract frameworks.

The formal specifications defined in this thesis serve not only as a basis for the type safety proofs, but also as an excellent starting point for further investigations of various aspects of C$^\sharp$ and CLR. Thus, one can analyze, for instance, stronger safety guarantees for C$^\sharp$ and CIL programs, or ways in which one can optimize programs in a type-safe manner. Moreover, the abstract and modular structure of the framework developed for CLR makes it possible to integrate new algorithms as well as new type systems into the existing formalization.

For formal proofs related to other languages, the experience achieved through the type safety proofs in this thesis, especially from the one developed in Chapter 4 for CIL, can be of help. One can learn, for instance:

- how to *"divide and conquer"* the type system, static and dynamic semantics of a language;
- how to separate the description of conceptually orthogonal constructs by dividing them into sublanguages; *modularity* is a key issue, since, unless the language changes excessively, the modifications in the formal specifications and proofs tend to be of local nature;
- how to unify and integrate the formal specifications of similar constructs by appropriate *parameterizations* (*e.g.*, smart grouping of bytecode instructions makes possible factoring out common properties and implicitly reduce the work load significantly);
- how to specify and evaluate *variations* of specific features (*e.g.*, expression evaluation, boxing and unboxing) by varying macros, rules and/or domains together with their operations.

The research underlying this thesis suggests, in particular, that, for the development of a programming language, the ASM method can be successfully applied for:

- defining an model as *executable specification* of critical language constructs or layers;
- *generating test cases* for the implementing code from the abstract model;
- using the abstract model as *oracle for test evaluations* and for comparing model test runs with code test runs;
- using the abstract model as *internal documentation* for future language extensions.

Bibliography

[1] JGNAT - GNAT-based Ada compiler for the Java Virtual Machine. Web pages at http://www.adacore.com.

[2] Microsoft .NET Framework 1.1 Software Development Kit. Web pages at http://msdn.microsoft.com/netframework/howtoget/.

[3] Microsoft .NET Framework 2.0 Software Development Kit. Web pages at http://msdn.microsoft.com/netframework/.

[4] .NET Framework Class Library, The RuntimeWrappedException Class. Web pages at http://msdn2.microsoft.com/en-us/library/system.runtime.compilerservices.runtimewrappedexception.aspx.

[5] .NET Framework Class Library, The System.Delegate Class. Web pages at http://msdn2.microsoft.com/en-us/library/system.delegate.aspx.

[6] .NET Framework Class Library, The System.Exception Class. Web pages at http://msdn2.microsoft.com/en-us/library/system.exception.aspx.

[7] .NET Framework Class Library, The System.RuntimeTypeHandle Structure. Web pages at http://msdn2.microsoft.com/en-us/library/system.runtimetypehandle.aspx.

[8] .NET Framework Class Library, The System.ThreadAbortException Class. Web pages at http://msdn2.microsoft.com/library/system.threading.threadabortexception.aspx.

[9] .NET Framework Class Library, The System.TypedReference Structure. Web pages at http://msdn2.microsoft.com/en-us/library/system.typedreference.aspx.

[10] .NET Framework Class Library, The System.ValueType Class. Web pages at http://msdn2.microsoft.com/en-us/library/system.valuetype.aspx.

[11] The Mercury Project .NET. Documentation available at http://www.cs.mu.oz.au/research/mercury/dotnet.html.

[12] The Objective Caml Language. Web pages at http://caml.inria.fr/ocaml/.

[13] The Scala Programming Language. Web pages at http://scala.epfl.ch/.

[14] The Haskell Home Page. Web pages at http://haskell.org/, 1990.

[15] Mono Compiler for C$^\sharp$. Web pages at http://www.go-mono.com/CSharp_Compiler, 2003.

[16] Rotor–Shared Source Common Language Infrastructure (SSCLI). Web pages at http://msdn.microsoft.com/net/sscli/, 2003.

[17] .NET Languages. Web pages at http://www.dotnetpowered.com/languages.aspx, 2006.

[18] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer–Verlag, 1998.

[19] Pierre America and Frank van der Linden. A Parallel Object-oriented Language with Inheritance and Subtyping. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications, Minneapolis, Minnesota, USA*, pages 161–168. ACM Press, 1990.

[20] Fabian Y. Bannwart and Peter Müller. A Logic for Bytecode. In F. Spoto, editor, *BYTECODE '05: Proceedings of the 1st Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Edinburgh, Scotland*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[21] Egon Börger. High Level System Design and Analysis Using Abstract State Machines. In *FM-Trends '98: Proceedings of the Workshop on Current Trends in Applied Formal Method, Boppard, Germany*, pages 1–43. Springer–Verlag, 1999.

[22] Egon Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(2-3):237–257, 2003.

[23] Egon Börger, Igor Durdanovic, and Dean Rosenzweig. Occam: Specification and Compiler Correctness - Part I: The Primary Model. In *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy*, volume A-56 of *IFIP Transactions*, pages 489–508. North-Holland, 1994.

[24] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A High–Level Modular Definition of the Semantics of C$^\sharp$. *Theoretical Computer Science*, 336(2–3):235–284, 2005.

[25] Egon Börger and Dean Rosenzweig. A Mathematical Definition of Full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[26] Egon Börger and Wolfram Schulte. A Practical Method for Specification and Analysis of Exception Handling-A Java/JVM Case Study. *IEEE Transactions on Software Engineering*, 26(9):872–887, 2000.

[27] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High–Level System Design and Analysis*. Springer, 2003.

[28] Egon Börger and Robert F. Stärk. Exploiting Abstraction for Specification Reuse: The Java/C$^\sharp$ Case Study. In F. S. de Boer et al., editors, *FMCO '03: Proceedings of the 2nd Symposium on Formal Methods for Components and Objects, Leiden, The Netherlands*, volume 3188 of *Lecture Notes in Computer Science*, pages 42–76. Springer–Verlag, 2004.

[29] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Generic Java Specification. Draft available at `http://citeseer.ist.psu.edu/bracha98gj.html`, 1998.

[30] Kim B. Bruce. Safe Type Checking in a Statically-typed Object-oriented Programming Language. In *POPL '93: Proceedings of the 20th Symposium on Principles of Programming Languages, Charleston, South Carolina, USA*, pages 285–298. ACM Press, 1993.

[31] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *OOPSLA '93: Proceedings of the 8th Conference on Object-oriented Programming Systems, Languages, and Applications, Washington, DC, USA*, pages 29–46. ACM Press, 1993.

[32] Chris Brumme. The Exception Model. WebLog at `http://blogs.msdn.com/cbrumme/`, October 2003.

[33] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded Polymorphism for Object-oriented Programming. In *FPCA '89: Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, London, United Kingdom*, pages 273–280. ACM Press, 1989.

[34] Robert Cartwright and Jr. Guy L. Steele. Compatible Genericity with Run-time Types for the Java Programming Language. In *OOPSLA '98: Proceedings of the 13th Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada*, pages 201–215. ACM Press, 1998.

[35] Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[36] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A Certifying Compiler for Java. In *PLDI '00: Proceedings of the Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada*, pages 95–107. ACM, 2000.

[37] William R. Cook. A Proposal for Making Eiffel Type–Safe. *Computer Journal*, 32(4):305–311, 1989.

[38] Tyson Dowd, Fergus Henderson, and Peter Ross. Compiling Mercury to the .NET Common Language Runtime. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[39] Stéphane Doyon and Mourad Debbabi. Verifying Object Initialization in the Java Bytecode Language. In *SAC '00: Proceedings of the Symposium on Applied Computing, Como, Italy*, pages 821–830. ACM Press, 2000.

[40] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe – Probably. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming, Jyväskylä, Finnland*, volume 1241 of *Lecture Notes In Computer Science*, pages 389–418. Springer–Verlag, 1997.

[41] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes In Computer Science*, pages 41–82. Springer–Verlag, 1999.

[42] Joe Duffy. Boxing Nullable and Verification. WebLog at `http://www.bluebytesoftware.com/blog/`, October, 2005.

[43] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2003.

[44] ECMA International. C$^\sharp$ *Language Specification, Standard ECMA–334*, 2001.

[45] ECMA International. *Common Language Infrastructure (CLI), Standard ECMA–335*, third edition, 2005.

[46] ECMA International. *Eiffel: Analysis, Design and Programming Language, Standard ECMA–367*, second edition, 2006.

[47] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and Generalized Constraints for C$^\sharp$ Generics. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming, Nantes, France*, 2006.

[48] Vladimir Fedorov. Single Pass Intermediate Language Verification Algorithm, 2004. Microsoft patent no. EP1376342, available at `http://gauss.ffii.org/PatentView/EP1376342`.

[49] Microsoft Research Foundations of Software Engineering Group. Abstract State Machine Language (AsmL). Web pages at `http://research.microsoft.com/foundations/AsmL/`.

[50] Markus Frauenfelder. An Implementation of a Bytecode Verifier for the .NET CLR, 2006. Master Thesis supervised by Nicu G. Fruja.

[51] Stephen N. Freund. *Type Systems for Object-oriented Intermediate Languages*. PhD thesis, Stanford University, 2000.

[52] Stephen N. Freund and John C. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[53] Stephen N. Freund and John C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3–4):271–321, 2003. Special issue on bytecode verification.

[54] Nicu G. Fruja. Experiments with the .NET CLR Exception Handling Mechanism. Available at http://www.inf.ethz.ch/personal/fruja/publications/clrexctests.pdf, 2004. Example programs to determine the meaning of CLR features not specified by the ECMA Standard for CLI.

[55] Nicu G. Fruja. Specification and Implementation Problems for C$\sharp$. In W. Zimmermann and B. Thalheim, editors, *ASM '04: Proceedings of the 11th Workshop on Abstract State Machines, Wittenberg, Germany*, volume 3052 of *Lecture Notes in Computer Science*, pages 127–143. Springer–Verlag, 2004.

[56] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C$\sharp$. In Vaclav Skala and Piotr Nienaltowski, editors, *.NET '04: Proceedings of the 2nd .NET Technologies Workshop, Pilsen, Czech Republic*, pages 81–88, 2004.

[57] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C$\sharp$ (extended version). *Journal of Object Technology*, 3(9):29–52, 2004.

[58] Nicu G. Fruja. A Modular Design for the .NET CLR Architecture. Technical Report 493, ETH Zürich, 2005.

[59] Nicu G. Fruja. Type Safety of Generics for the .NET Common Language Runtime. Technical Report 519, ETH Zürich, 2006.

[60] Nicu G. Fruja. Type Safety of Generics for the .NET Common Language Runtime. In Peter Sestoft, editor, *ESOP '06: Programming Languages and Systems, Proceedings of the 15th European Symposium on Programming, Vienna, Austria*, volume 3924 of *Lecture Notes In Computer Science*, pages 325–341. Springer–Verlag, 2006.

[61] Nicu G. Fruja and Egon Börger. Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis. *Journal of Object Technology*, 5(3):5–34, 2006.

[62] Nicu G. Fruja and Robert F. Stärk. The Hidden Computation Steps of Turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *ASM '03: Proceedings of the 10th Workshop on Abstract State Machines, Taormina, Italy*, volume 2589 of *Lecture Notes in Computer Science*, pages 244–262. Springer–Verlag, 2002.

[63] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *OOPSLA '03: Proceedings of the 18th Conference on Object-oriented Programing, Systems, Languages, and Applications, Anaheim, California, USA*, pages 115–134. ACM Press, 2003.

[64] Sabine Glesner. ASMs versus Natural Semantics: A Comparison with New Insights. In E. Börger, A. Gargantini, and E. Riccobene, editors, *ASM '03: Proceedings of the 10th Workshop on Abstract State Machines, Taormina, Italy*, volume 2589 of *Lecture Notes In Computer Science*, pages 293–308. Springer–Verlag, 2003.

[65] Andrew Gordon, Don Syme, Jonathon Forbes, and Vance P. Morrison. Verifier to Check Intermediate Language, 2003. Microsoft patent no. 6560774, available at http://www.freepatentsonline.com/6560774.html.

[66] Andrew D. Gordon and Don Syme. Typing a Multi-Language Intermediate Code. Technical Report 106, Microsoft Research, 2000.

[67] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[68] K. John Gough. Stacking them up: a Comparison of Virtual Machines. In *ACSAC '01: Proceedings of the 6th Australasian Conference on Computer Systems Architecture, Queensland, Australia*, pages 55–61. IEEE Computer Society, 2001.

[69] K. John Gough and Diane Corney. Evaluating the Java Virtual Machine as a Target for Languages other than Java. In *JMLC '00: Proceedings of the Joint Modular Languages Conference on Modular Programming Languages, Zürich, Switzerland*, volume 1897 of *Lecture Notes In Computer Science*, pages 278–290. Springer–Verlag, 2000.

[70] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.

[71] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In E. Börger, G. Jäger, H. Kleine-Büning, S. Martini, and M. M. Richter, editors, *CSL '92: Selected Papers from the 6th Workshop on Computer Science Logic, San Miniato, Italy*, pages 274–308. Springer–Verlag, 1993.

[72] Yuri Gurevich and Lawrence S. Moss. Algebraic Operational Semantics and Occam. In *CSL '89: Proceedings of the 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany*, volume 40 of *Lecture Notes In Computer Science*, pages 176–192. Springer–Verlag, 1990.

[73] Pieter H. Hartel and Luc Moreau. Formalizing the Safety of Java, the Java Virtual Machine, and Java Card. *ACM Computer Surveys*, 33(4):517–558, 2001.

[74] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. C$^\sharp$ *Language Specification 1.2*. MSDN, 2003.

[75] Mark Howard, Éric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stapf, Karine Arnout, and Markus Keller. Type–safe Covariance: Competent Compilers Can Catch all Catcalls. Draft available at `http://se.ethz.ch/~meyer/`, 2003.

[76] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. In *OOPSLA '99: Proceedings of the 14th Conference on Object-oriented Programming, Systems, Languages, and Applications, Denver, Colorado, USA*, pages 132–146. ACM Press, 1999.

[77] Atsushi Igarashi and Benjamin Pierce. On Inner Classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object–Oriented Programming, Sophia Antipolis and Cannes, France*, volume 1850 of *Lecture Notes In Computer Science*, pages 129–153. Springer–Verlag, 2000.

[78] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[79] Atsushi Igarashi and Mirko Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, 2006.

[80] International Computer Science Institute, University of California. *Sather*. Documentation available at `www.icsi.berkeley.edu/~sather`.

[81] Bart Jacobs and Erik Poll. A Monad for Basic Java Semantics. In *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology, Iowa City, IA, USA*, volume 1816 of *Lecture Notes in Computer Science*, pages 150–164. Springer–Verlag, 2000.

[82] Jon Jagger, Nigel Perry, and Peter Sestoft. C$^\sharp$ *Annotated Standard*. Elsevier, 2007.

[83] Horatiu V. Jula. ASM semantics for C$^\sharp$ 2.0. In D. Beauquier, A. Slissenko, and E. Börger, editors, *ASM '05: Proceedings of the 12th Workshop on Abstract State Machines, Paris, France*, 2005.

[84] Horatiu V. Jula and Nicu G. Fruja. An Executable Specification of C$^\sharp$. In D. Beauquier, A. Slissenko, and E. Börger, editors, *ASM '05: Proceedings of the 12th Workshop on Abstract State Machines, Paris, France*, pages 275–287, 2005.

[85] Gilles Kahn. Natural Semantics. In *STACS '87: Proceedings of the 4th Symposium on Theoretical Aspects of Computer Sciences, Passau, Germany*, pages 22–39. Springer–Verlag, 1987.

[86] Jonathan Keljo. Personal Communication, September 2004.

[87] Andrew Kennedy and Claudio V. Russo. Generalized Algebraic Data Types and Object-oriented Programming. In *OOPSLA '05: Proceedings of the 20th Conference on Object-oriented Programming, Systems, Languages, and Applications, San Diego, California, USA*, pages 21–40. ACM Press, 2005.

[88] Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI '01: Proceedings of the Conference on Programming Language Design and Implementation, Snowbird, Utah, USA*, pages 1–12. ACM, 2001.

[89] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Technical University Munich, 2003.

[90] Philipp W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[91] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition, 1999.

[92] Christian Marrocco. An Executable Specification of the .NET CLR, 2005. Diploma Thesis supervised by Nicu G. Fruja.

[93] Oberon Microsystems. Component Pascal Language Report. Web pages `http://www.oberon.ch/resources`, 1999.

[94] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[95] Marcin Mlotkowski. *Specification and Optimization of the Smalltalk Programs*. PhD thesis, University of Wroclaw, 2001.

[96] George C. Necula. Proof-carrying Code. In *POPL '97: Proceedings of the 24th Symposium on Principles of Programming Languages, Paris, France*, pages 106–119. ACM Press, 1997.

[97] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe – Definitely. In *POPL '98: Proceedings of the 25th Symposium on Principles of Programming Languages, San Diego, California, USA*, pages 161–170. ACM Press, 1998.

[99] Robert O'Callahan. A Simple, Comprehensive Type System for Java Bytecode Subroutines. In *POPL '99: Proceedings of the 26th Symposium on Principles of Programming Languages, San Antonio, Texas, USA*, pages 70–78. ACM Press, 1999.

[100] Larry Paulson and Tobias Nipkow. Isabelle – Generic Theorem Proving Environment. Cambridge University and Technical University Munich.

[101] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[102] Matt Pietrek. A Crash Course on the Depths of Win32^{TM} Structured Exception Handling. *Microsoft Systems Journal*, January 1997.

[103] Cornelia Pusch. Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, Amsterdam, The Netherlands*, volume 1579 of *Lecture Notes In Computer Science*, pages 89–103. Springer–Verlag, 1999.

[104] Zhenyu Qian. Constraint-based Specification and Dataflow Analysis for Java Bytecode Verification. Technical report, Kestrel, 1997.

[105] Norbert Schirmer. Java Definite Assignment in Isabelle/HOL. In *FTfJP '03: Proceedings of the Workshop on Formal Techniques for Java–like Programs, Darmstadt, Germany*, pages 13–21, 2003.

[106] Joachim Schmid. Executing ASM Specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.

[107] Joachim Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, 2002.

[108] Sam Shiel and Ian Bayley. A Translation-Facilitated Comparison Between the Common Language Runtime and the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 141(1):35–52, 2005.

[109] S. Somasegar. Nulls not Missing Anymore. WebLog at `http://blogs.msdn.com/somasegar/`, August, 2005.

[110] Robert F. Stärk. The Theoretical Foundations of LPTP (A Logic Program Theorem Prover). *Journal of Logic Programming*, 36(3):241–269, 1998.

[111] Robert F. Stärk. Formal Specification and Verification of the C$\sharp$ Thread Model. *Theoretical Computer Science*, 343(3):482–508, 2005.

[112] Robert F. Stärk and Egon Börger. An ASM Specification of C$\sharp$ Threads and the .NET Memory Model. In W. Zimmermann and B. Thalheim, editors, *ASM '04: Proceedings of the 11th Workshop on Abstract State Machines, Wittenberg, Germany*, volume 3052 of *Lecture Notes in Computer Science*, pages 38–60. Springer–Verlag, 2004.

[113] Robert F. Stärk and Joachim Schmid. The Problem of Bytecode Verification in Current Implementations of the JVM. Technical report, ETH Zürich, 2001.

[114] Robert F. Stärk and Joachim Schmid. Completeness of a Bytecode Verifier and a Certifying Java-to-JVM Compiler. *Journal of Automated Reasoning*, 30(3–4):323–361, 2003. Special issue on bytecode verification.

[115] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation*. Springer–Verlag, 2001.

[116] Raymie Stata and Martìn Abadi. A Type System for Java Bytecode Subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[117] Don Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.

[118] Don Syme. Proving Java Type Soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes In Computer Science*, pages 83–118. Springer–Verlag, 1999.

[119] Alfred Tarski. A Lattice Theoretical Fix-point Theorem and its Applications. *Pacific Journal of Mathematics*, 2(5):285–309, 1955.

[120] Mads Torgersen, Erik Ernst, and Christian P. Hansen. Wild FJ. In *FOOL '05: Proceedings of the 12th Workshop on Foundations of Object-Oriented Languages, Long Beach, California, USA*, 2005.

[121] David von Oheimb and Tobias Nipkow. Machine–Checking the Java Specification: Proving Type-Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes In Computer Science*, pages 119–156. Springer–Verlag, 1999.

[122] Charles Wallace. The Semantics of the C++ Programming Language. In *Specification and Validation Methods*, pages 131–164. Oxford University Press, Inc., 1995.

[123] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of Generics for the .NET Common Language Runtime. In *POPL '04: Proceedings of the 31st Symposium on Principles of Programming Languages, Venice, Italy*, pages 39 – 51. ACM, 2004.

List of Figures

List of Tables

A
Appendix

A.1 The C$^\sharp_{\mathcal{S}}$ Operational Semantics Rules

The operational semantics model is defined in terms of an ASM interpreter (for C$^\sharp_{\mathcal{S}}$ programs), named EXECCSHARP:

```
EXECCSHARP ≡
    EXECCSHARPEXP
    EXECCSHARPSTM
```

The main transition rule consists of two subrules executed *in parallel*: EXECCSHARPEXP defines one execution step in the *evaluation of expressions*, and EXECCSHARPSTM specifies one execution step in the *execution of statements*. Upon executing one of the subrules, the updates implied by the *first* evaluation rule matching the current context are fired.

The macro *context*(*pos*) is used to define the *context* of the currently to be handled expression or statement or intermediate result, which has to be matched against the syntactically possible cases, in the *textual order* of the rules, to select the appropriate computation step.

Figure A.1 The operational semantics rules for the C$^\sharp_{\mathcal{S}}$'s expressions.

```
EXECCSHARPEXP ≡ match $context(pos)$
  $lit$                                  → YIELD($valueOfLiteral(lit)$)
  $loc$                                  → YIELDINDIRECT($locAdr(loc)$)

  $^{\alpha}exp\ bop\ ^{\beta}exp'$              → $pos := \alpha$
  $^{\blacktriangleright}val\ bop\ ^{\beta}exp'$              → $pos := \beta$
  $^{\alpha}val\ bop\ ^{\blacktriangleright}val'$              → if $\mathcal{ST}(pos) \notin Delegate$ then
                                             if $divisionByZero(bop, val')$ then FAILUP(DivideByZeroException)
                                             else YIELDUP($opResVal(bop, val, val')$)
                                         elseif $bop = +$ then DELEGATECOMBINE($\mathcal{ST}(pos), val, val'$)
                                         elseif $bop = -$ then DELEGATEREMOVE($\mathcal{ST}(pos), val, val'$)
                                         elseif $bop = ==$ then DELEGATEEQUAL($\mathcal{ST}(pos), val, val'$)

  $^{\alpha}exp\ ?\ ^{\beta}exp'\ :\ ^{\gamma}exp''$      → $pos := \alpha$
  $^{\blacktriangleright}val\ ?\ ^{\beta}exp'\ :\ ^{\gamma}exp''$      → if $val$ then $pos := \beta$ else $pos := \gamma$
  $^{\alpha}$true ? $^{\blacktriangleright}val\ :\ ^{\gamma}exp''$     → YIELDUP($val$)
  $^{\alpha}$false ? $^{\beta}exp'\ :\ ^{\blacktriangleright}val$    → YIELDUP($val$)
```

Figure A.2 The operational semantics rules for the $C^{\sharp}_{\mathcal{S}}$'s expressions (continued).

```
$loc = {}^{\alpha}exp$                      $\to pos := \alpha$
$loc = {}^{\blacktriangleright}val$          $\to$ $\textsc{WriteMem}(locAdr(loc), locType(loc), val)$
                                            $\textsc{YieldUp}(val)$

${}^{\alpha}vexp\ bop = {}^{\beta}exp$        $\to pos := \alpha$
${}^{\blacktriangleright}adr\ bop = {}^{\beta}exp$  $\to pos := \beta$
${}^{\alpha}adr\ bop = {}^{\blacktriangleright}val$ $\to$ if $\mathcal{ST}(\alpha) \notin Delegate$ then
                                              let $val' = memVal(adr, \mathcal{ST}(\alpha))$ in
                                                if $divisionByZero(bop, val)$ then
                                                  $\textsc{FailUp}$(DivideByZeroException)
                                                else let $val'' = convertVal(\mathcal{ST}(\alpha), opResVal(bop, val', val))$ in
                                                  $\textsc{WriteMem}(adr, \mathcal{ST}(\alpha), val'')$
                                                  $\textsc{YieldUp}(val'')$
                                            elseif $bop = +$ then $\textsc{DelegateCombine}(\mathcal{ST}(\alpha), val', val)$
                                            elseif $bop = -$ then $\textsc{DelegateRemove}(\mathcal{ST}(\alpha), val', val)$

ref ${}^{\alpha}vexp$                         $\to pos := \alpha$
ref ${}^{\blacktriangleright}adr$             $\to \textsc{YieldUp}(adr)$
out ${}^{\alpha}vexp$                         $\to pos := \alpha$
out ${}^{\blacktriangleright}adr$             $\to \textsc{YieldUp}(adr)$

$()$                                        $\to \textsc{Yield}([\,])$
$({}^{\alpha_1}arg_1, \ldots, {}^{\alpha_n}arg_n)$  $\to pos := \alpha_1$
$({}^{\alpha_1}valadr_1, \ldots, {}^{\blacktriangleright}valadr_n)$  $\to \textsc{YieldUp}([valadr_1, \ldots, valadr_n])$
$(\ldots {}^{\blacktriangleright}valadr_i, {}^{\alpha_{i+1}}arg_{i+1} \ldots)$  $\to pos := \alpha_{i+1}$

${}^{\alpha}exp$ is $T$                      $\to pos := \alpha$
${}^{\blacktriangleright}val$ is $T$          $\to$ if $\mathcal{ST}(pos) \in ValueType$ then $\textsc{YieldUp}(\mathcal{ST}(pos) \preceq T)$
                                            else $\textsc{YieldUp}(val \neq$ null $\wedge\ actualTypeOf(val) \preceq T)$

${}^{\alpha}exp$ as $T$                      $\to pos := \alpha$
${}^{\blacktriangleright}val$ as $T$          $\to$ if $\mathcal{ST}(pos) \in ValueType$ then
                                            elseif $(val \neq$ null $\wedge\ actualTypeOf(val) \preceq T)$ then $\textsc{YieldUp}(val)$
                                            else $\textsc{YieldUp}$(null)

$(T)\ {}^{\alpha}exp$                        $\to pos := \alpha$
$(T)\ {}^{\blacktriangleright}val$            $\to$ if $\mathcal{ST}(pos) \in ValueType$ then
                                              if $\mathcal{ST}(pos) \in PrimitiveType \wedge T \in PrimitiveType$ then
                                                $\textsc{YieldUp}(convertVal(T, val))$
                                              elseif $T = \mathcal{ST}(pos)$ then $\textsc{YieldUp}(val)$
                                              if $T \in RefType$ then $\textsc{YieldUpBox}(\mathcal{ST}(pos), val)$
                                            elseif $T \in RefType$ then
                                              if $val =$ null $\vee\ actualTypeOf(val) \preceq T$ then $\textsc{YieldUp}(val)$
                                              else $\textsc{FailUp}$(InvalidCastException)
                                            elseif $val \neq$ null $\wedge\ T = actualTypeOf(val)$ then
                                              $\textsc{YieldUp}(memVal(valueAdr(val), T))$
                                            elseif $val =$ null then $\textsc{FailUp}$(NullReferenceException)
                                            else $\textsc{FailUp}$(InvalidCastException)

${}^{\alpha}exp.T{::}F$                      $\to pos := \alpha$
${}^{\blacktriangleright}valadr.T{::}F$       $\to$ if $\mathcal{ST}(pos) \in ValueType \wedge valadr \notin Adr$ then $\textsc{YieldUp}(valadr(T{::}F))$
                                            elseif $valadr \neq$ null then $\textsc{YieldUpIndirect}(fieldAdr(valadr, T{::}F))$
                                            else $\textsc{FailUp}$(NullReferenceException)

${}^{\alpha}exp.T{::}F = {}^{\beta}exp'$      $\to pos := \alpha$
${}^{\blacktriangleright}valadr.T{::}F = {}^{\beta}exp'$  $\to pos := \beta$
${}^{\alpha}valadr.T{::}F = {}^{\blacktriangleright}val'$  $\to$ if $valadr \neq$ null then
                                              $\textsc{WriteMem}(fieldAdr(valadr, T{::}F), fieldType(T{::}F), val')$
                                              $\textsc{YieldUp}(val')$
                                            else $\textsc{FailUp}$(NullReferenceException)
```

Figure A.3 The operational semantics rules for the C$\sharp_{\mathcal{S}}$'s expressions (continued).

```
$\texttt{new}\ C$   $\rightarrow$ let $ref = new(ObjRef, C)$ in
        $actualTypeOf(ref) := C$
        forall $C'::F \in instFields(C)$ do
          let $adr = fieldAdr(ref, C'::F)$ and $T = fieldType(C'::F)$ in
            $\textsc{WriteMem}(adr, T, defVal(T))$
        $\textsc{Yield}(ref)$

$^{\alpha}exp.T::M\ ^{\beta}(args)$   $\rightarrow pos := \alpha$
$^{\blacktriangleright}valadr.T::M\ ^{\beta}(args)$   $\rightarrow pos := \beta$
$^{\alpha}valadr.T::M\ ^{\blacktriangleright}(valadrs)$ $\rightarrow$ if $valadr \neq \texttt{null}$ then $\textsc{VirtCall}(T::M, valadr, valadrs)$
      else $\textsc{FailUp}(\texttt{NullReferenceException})$

$\texttt{new}\ D(^{\alpha}exp.T::M)$   $\rightarrow pos := \alpha$
$\texttt{new}\ D(^{\blacktriangleright}val.T::M)$   $\rightarrow$ if $val = \texttt{null}$ then $\textsc{FailUp}(\texttt{NullReferenceException})$
      else let $d = new(ObjRef, D)$ in
        $actualTypeOf(d) := D$
        if $callKind(T::M) = \texttt{Virtual}$ then
          let $T''::M = lookUp(actualTypeOf(val), T::M)$ in
            $invocationList(d) := [val, T''::M]$
        else let $ref' =$ if $T \in Struct$ then $NewBox(\mathcal{ST}(pos), val)$ else $val$ in
          $invocationList(d) := [ref', T::M]$
        $\textsc{YieldUp}(d)$

$\texttt{new}\ D(^{\alpha}exp)$   $\rightarrow pos := \alpha$
$\texttt{new}\ D(^{\blacktriangleright}ref)$   $\rightarrow$ if $ref = \texttt{null}$ then $\textsc{FailUp}(\texttt{NullReferenceException})$
      else let $d = new(ObjRef, D)$ in
        $actualTypeOf(d) := D$
        $invocationList(d) := invocationList(ref)$
        $\textsc{YieldUp}(d)$
```

Figure A.4 The operational semantics rules for the C$\sharp_{\mathcal{S}}$'s statements.

```
$\textsc{ExecCsharpStm} \equiv$ match $context(pos)$
  $^{\alpha}exp;$   $\rightarrow pos := \alpha$
  $^{\blacktriangleright}val;$   $\rightarrow \textsc{YieldUp}(\texttt{Norm})$
  $\texttt{break};$   $\rightarrow \textsc{Yield}(\texttt{Break})$
  $\texttt{continue};$   $\rightarrow \textsc{Yield}(\texttt{Continue})$
  $\texttt{goto}\ lab;$   $\rightarrow \textsc{Yield}(\texttt{Goto}(lab))$

  $\texttt{if}\ (^{\alpha}exp)\ ^{\beta}stm\ \texttt{else}\ ^{\gamma}stm'$   $\rightarrow pos := \alpha$
  $\texttt{if}\ (^{\blacktriangleright}val)\ ^{\beta}stm\ \texttt{else}\ ^{\gamma}stm'$   $\rightarrow$ if $val$ then $pos := \beta$ else $pos := \gamma$
  $\texttt{if}\ (^{\alpha}\texttt{true})\ ^{\blacktriangleright}\texttt{Norm}\ \texttt{else}\ ^{\gamma}stm'$   $\rightarrow \textsc{YieldUp}(\texttt{Norm})$
  $\texttt{if}\ (^{\alpha}\texttt{false})\ ^{\beta}stm\ \texttt{else}\ ^{\blacktriangleright}\texttt{Norm}$ $\rightarrow \textsc{YieldUp}(\texttt{Norm})$

  $\texttt{while}\ (^{\alpha}exp)\ ^{\beta}stm$   $\rightarrow pos := \alpha$
  $\texttt{while}\ (^{\blacktriangleright}val)\ ^{\beta}stm$   $\rightarrow$ if $val$ then $pos := \beta$
        else $\textsc{YieldUp}(\texttt{Norm})$
  $\texttt{while}\ (^{\alpha}\texttt{true})\ ^{\blacktriangleright}\texttt{Norm}$   $\rightarrow pos := up(pos)$
        $\textsc{ClearValues}(up(pos))$
  $\texttt{while}\ (^{\alpha}\texttt{true})\ ^{\blacktriangleright}\texttt{Break}$   $\rightarrow \textsc{YieldUp}(\texttt{Norm})$
  $\texttt{while}\ (^{\alpha}\texttt{true})\ ^{\blacktriangleright}\texttt{Continue}$   $\rightarrow pos := up(pos)$
        $\textsc{ClearValues}(up(pos))$
  $\texttt{while}\ (^{\alpha}\texttt{true})\ ^{\blacktriangleright}abr$   $\rightarrow \textsc{YieldUp}(abr)$
```

Figure A.5 The operational semantics rules for the $C^\sharp_S$'s statements (continued).

```
$T\ loc$;                                   → YIELD(Norm)

$lab$ : $^{\alpha}stm$                       → $pos := \alpha$
$lab$ : $^{\blacktriangleright}$Norm         → YIELDUP(Norm)

{ }                                         → YIELD(Norm)
{$^{\alpha}stm$ ...}                         → $pos := \alpha$
{... $^{\blacktriangleright}$Norm}           → YIELDUP(Norm)
{... $^{\blacktriangleright}$Norm $^{\alpha}stm$ ...}  → $pos := \alpha$
{... $^{\blacktriangleright}$Goto($L$) ...}  → let $\alpha$ = GOTOTARGET($first(up(pos)), L$)
                                                if $\alpha \neq undef$ then
                                                  $pos := \alpha$
                                                  CLEARVALUES($up(pos)$)
                                                else YIELDUP(Goto($L$))

return $^{\alpha}exp$;                       → $pos := \alpha$
return $^{\blacktriangleright}val$;          → YIELDUP(Return($val$))
return;                                     → YIELD(Return)

Return                                      → if ($pos$ is the position of $meth$'s body) $\wedge$ $frameStack \neq [\,]$ then
                                                EXITMETHOD(Norm)
Return($val$)                               → if ($pos$ is the position of $meth$'s body) $\wedge$ $frameStack \neq [\,]$ then
                                                EXITMETHOD($val$)

$^{\blacktriangleright}$Norm;                → YIELDUP(Norm)

throw $^{\alpha}exp$;                        → $pos := \alpha$
throw $^{\blacktriangleright}ref$;           → if $ref$ = null then FAILUP(NullReferenceException)
                                              else YIELDUP(Exc($ref$))
throw;                                      → YIELD(Exc($top(excStack)$))

try $^{\alpha}block$ catch ($E\ x$) $stm$    → $pos := \alpha$
try $^{\blacktriangleright}$Norm catch ($E\ x$) $stm$  → YIELDUP(Norm)
try $^{\blacktriangleright}$Exc($ref$)       → if $\exists\, i = 1,n$ : $actualTypeOf(ref) \preceq E_i$ then
catch($E_1\ x_1$) $^{\alpha_1}stm_1$             let $j = \min\,\{i = 1,n \mid actualTypeOf(ref) \preceq E_i\}$ in
...                                               $pos := \alpha_j$
catch($E_n\ x_n$) $^{\alpha_n}stm_n$               $excStack := push(excStack, ref)$
                                                  WRITEMEM($locAdr(x_j)$, object, $ref$)
                                              else YIELDUP(Exc($ref$))
try $^{\blacktriangleright}abr$              → YIELDUP($abr$)
catch($E_1\ x_1$) $stm_1$
...
catch($E_n\ x_n$) $stm_n$
try Exc($ref$) catch($E\ x$) $^{\blacktriangleright}res$ → $excStack := pop(excStack)$
                                              YIELDUP($res$)

try $^{\alpha}block$ finally $^{\beta}block'$    → $pos := \alpha$
try $^{\blacktriangleright}res$ finally $^{\beta}block'$  → $pos := \beta$
try $res$ finally $^{\blacktriangleright}$Norm   → YIELDUP($res$)
try $res$ finally $^{\blacktriangleright}$Exc($ref$)  → YIELDUP(Exc($ref$))

Exc($ref$)                                  → if ($pos$ is the position of $meth$'s body) $\wedge$ $frameStack \neq [\,]$ then
                                                EXITMETHOD(Exc($ref$))
{... $^{\blacktriangleright}abr$ ...}         → YIELDUP($abr$)
... $^{\blacktriangleright}abr$ ...           → if $up(pos) \neq undef \wedge propagatesAbr(up(pos))$ then
                                                YIELDUP($abr$)
```

The macro FAILUP is applied to throw exceptions. Given an exception class E, it executes the statement `throw new` $E()$; at the parent position. It accomplishes this by invoking an internal method, named `Throw`E (whose body is assumed to contain only the above `throw` statement), with that effect for each E.

```
FAILUP($E$) ≡ INVOKEMETHOD($ExcSupport$::Throw$E$, [ ])
```

```
DELEGATECOMBINE($D, ref, ref'$) ≡
  if $ref$ = null then YIELDUP($ref'$)
  elseif $ref'$ = null then YIELDUP($ref$)
  else let $d = new(ObjRef, D)$ in
    $actualTypeOf(d) := D$
    $invocationList(d) := invocationList(ref) \cdot invocationList(ref')$
    YIELDUP($d$)
```

```
DELEGATEREMOVE($D, ref, ref'$) ≡
  if $ref$ = null then YIELDUP(null)
  elseif $ref'$ = null then YIELDUP($ref$)
  else let $L = invocationList(ref)$ and $L' = invocationList(ref')$ in
    if $L = L'$ then YIELDUP(null)
    elseif $sublist(L', L)$ then
      let $d = new(ObjRef, D)$ in
        $actualTypeOf(d) := D$
        $invocationList(d) := prefix(L', L) \cdot suffix(L', L)$
        YIELDUP($d$)
    else YIELDUP($ref$)
```

```
DELEGATEEQUAL($ref, ref'$) ≡
  if $ref$ = null ∨ $ref'$ = null then YIELDUP($ref = ref'$)
  else let $L = invocationList(ref)$ and $L' = invocationList(ref')$ in
    YIELDUP($length(L) = length(L') \wedge \forall i = 0, length(L) - 1 \; : \; (L(i) = L'(i))$)
```

```
YIELDUPBOX($T, val$) ≡
  let $ref = newBox(T, val)$ in
    YIELDUP($ref$)
```

```
INVOKEMETHOD($T$::$M, valadrs$) ≡
  if $T \in Delegate$ then
    if $M$ = _length then YIELDUP($length(invocationList(valadrs(0)))$)
    if $M$ = _invoke then CALLDELEGATE($valadrs(0), valadrs(1), drop(valadrs, 2)$)
  else
    $frameStack := push(frameStack, (meth, up(pos), locAdr, values))$
    $meth$      $:= T::M$
    $pos$       $:= \alpha$, where $^{\alpha}body(T::M)$
    $values$    $:= \emptyset$
    INITLOCALS($T::M, valadrs$)
```

```
CALLDELEGATE($ref, i, valadrs$) ≡
  let $(ref', T::M) = invocationList(ref)(i)$ in
    let this = if $T \in Struct$ then $valueAdr(ref')$ else $ref'$ in
      INVOKEMETHOD($T::M$, [this] $\cdot valadrs$)
```

```
$\textsc{InitLocals}(T{::}M, valadrs) \equiv$
  forall $loc \in localVars(T{::}M)$ do
    let $adr = new(Adr, locType(T{::}M, loc))$ in
      $locAdr(loc) := adr$
  forall $loc \in valueParams(T{::}M)$ do
    let $adr = new(Adr, paramType(T{::}M, loc))$ in
      $locAdr(loc) := adr$
      $\textsc{WriteMem}(adr, paramType(T{::}M, loc), valadrs(paramIndex(T{::}M, loc)))$
    forall $loc \in refParams(T{::}M) \cup outParams(T{::}M)$ do
      $locAdr(loc) := valadrs(paramIndex(T{::}M, loc))$
```

```
$\textsc{ClearValues}(\alpha) \equiv$
  $values(\alpha) := undef$
  if $first(\alpha) \neq undef$ then $\textsc{ClearValuesSeq}(first(\alpha))$
```

```
$\textsc{ClearValuesSeq}(\alpha) \equiv$
  $\textsc{ClearValues}(\alpha)$
  if $next(\alpha) \neq undef$ then $\textsc{ClearValuesSeq}(next(\alpha))$
```

$propagatesAbr(\alpha) :\Leftrightarrow label(\alpha) \notin \{Block, \texttt{while}\}$

```
$\textsc{GotoTarget}(\alpha, lab) =$
  if $label(\alpha) = Lab(lab)$ then $\alpha$
  elseif $next(\alpha) = undef$ then $undef$
  else $\textsc{GotoTarget}(next(\alpha), lab)$
```

```
$\textsc{ExitMethod}(res) \equiv$
  let $(oldMeth, oldPos, oldLocals, oldValues) = top(frameStack)$ in
    $meth$       $:= oldMeth$
    $pos$        $:= oldPos$
    $locAdr$     $:= oldLocals$
    $frameStack$ $:= pop(frameStack)$
    $values$     $:= oldValues \oplus \{oldPos \mapsto res\}$
```

Curriculum Vitae

Nicu Georgian Fruja

January 13, 1977	Born in Ramnicu-Sarat, Romania
1983 – 1987	Primary school, Ramnicu-Sarat
1987 – 1991	Secondary school, Ramnicu-Sarat
1991 – 1995	High school, Ramnicu-Sarat
1995 – 1999	Studies in Applied Mathematics, University of Bucharest, Romania
1998 – 1999	Exchange Studies in Applied Statistics, Technical University Munich, Germany
1999 – 2001	Master Studies in Applied Statistics and Optimization, University of Bucharest, Romania
1999 – 2001	Software Developer, National Institute for Statistics and Economic Studies, Romania
2000 – 2001	Teaching Assistant, Department of Probabilities, Statistics, and Operations Research, University of Bucharest, Romania
2001 – 2006	Research and Teaching Assistant, Computer Science Department, ETH Zurich, Switzerland