A High-Level Modular Definition of the Semantics of C♯

Egon Börger [a], Nicu G. Fruja [b], Vincenzo Gervasi [a], Robert F. Stärk [b,*]

[a] *Dipartimento di Informatica, Università di Pisa, Via F. Buonarroti 2, I-56127 Pisa, Italy*

[b] *Computer Science Department, ETH Zürich, CH-8092 Zürich, Switzerland*

Abstract

We propose a structured mathematical definition of the semantics of C♯ programs to provide a platform-independent interpreter view of the language for the C♯ programmer, which can also be used for a precise analysis of the ECMA standard of the language and as a reference model for teaching. The definition takes care to reflect directly and faithfully – as much as possible without becoming inconsistent or incomplete – the descriptions in the C♯ standard to become comparable with the corresponding models for Java in [1] and to provide for implementors the possibility to check their basic design decisions against an accurate high-level model. The model sheds light on some of the dark corners of C♯ and on some critical differences between the ECMA standard and the implementations of the language.

Key words: Semantics of programming languages, Abstract State Machines, C♯, Java, .NET

1 Introduction

In this paper the method developed in [1] for a rigorous definition and analysis of Java and its implementation on the Java Virtual Machine (JVM) is applied to formalize the semantics of the entire language C♯. We provide a

* Corresponding author.

Email addresses: boerger@di.unipi.it (Egon Börger), fruja@inf.ethz.ch (Nicu G. Fruja), gervasi@di.unipi.it (Vincenzo Gervasi), staerk@inf.ethz.ch (Robert F. Stärk).

succinct, purely mathematical (thus platform-independent) model, which reflects as much as possible the intuitions and design decisions underlying the language as described in the ECMA standard [2] and in [3] and can be used as accurate and complete reference model by C♯ programmers, by implementors of the language and and by students learning it. In Sect. 8 we point to some challenging applications of the model for proving interesting theorems about C♯ and its implementations.

The model clarifies a certain number of semantically relevant issues which are not handled by the ECMA standard, wherefore we also consulted the Microsoft Press books [4–6] and the documentation in [7–11]. A series of bugs and gaps in the ECMA standard for C♯ and in its implementation in .NET and incoherences between the two were detected during our attempt to build for the language a consistent and complete yet abstract *ground model* (in the sense described in [12]). Some of them are mentioned in this paper to shed light on some dark corners of C♯, for a complete discussion we refer the reader to the companion paper [13]. As a rule we adhere to an established scientific tradition for which one of the goals of defining the meaning of programs is to accurately specify the freedom the compiler writer has for the implementation. Nevertheless we also want our model to support the practice of programming. Therefore, whenever we see for a language construct an incoherence or a to-be-closed gap between on the one side the view offered by the ECMA standard, which should support the understanding also by programmers, and on the other side the view current compilers seem to have, we give in our model a pragmatic preference to abstractly defining what the programmer is allowed to expect from the execution of his code in the current implementations of C♯ [7–9]. In each case we explicitly discuss the discovered discrepancy so that the parameters of the design decision become clear. To support the experimentation with the model a project has been started to refine the model developed here to .NET-executable AsmL code [14], similarly to the AsmGofer refinement developed by Joachim Schmid [15,16] for the Java and JVM models in [1].

To provide the programmer with a transparent view of the intricate interaction of various language features which depend on the run-time environment, our model comes as an *abstract interpreter*, which provides a simple way to reflect those run-time-related features encountered upon executing a given C♯ program. To exploit the flexibility the use of Abstract State Machines (ASMs) offers in high-level system modeling and to obtain the faithfulness and simplicity of abstract models the ASM method allows one to achieve, the interpreter takes the form of an ASM.[1] This allows us in particular to specify the static and the dynamic parts of the semantics separately, due to the ASM

[1] See Sect. 8 for more information on our choice of ASMs among the many frameworks in the literature to deal with language semantics.

classification of abstract states into a static and a dynamic part. The *dynamic semantics* of the language is captured operationally by ASM rules which describe the run-time effect of program execution on the abstract state of the program, the *static semantics* comes as a mainly declarative description of the relevant syntactical and compile-time checked language features (like typing rules, rules for definite assignment and reachability, name resolution, method resolution for overloaded methods, etc.) and of pre-processing directives (like `#define`, `#undef`, `#if`, `#else`, `#endif`, etc.), which are mostly reflected in the attributed abstract syntax tree our model starts from.

To keep the size of the models small and to facilitate the understanding of clusters of language constructs in terms of local state transformations, similarly to the decomposition of Java and the JVM in [1] we structure the C♯ programming language into *layered modules of orthogonal language features*, namely

- the imperative core, related to sequential control by while programs, built from statements and expressions over the simple types of C♯,
- classes, realizing procedural abstraction with global (module) variables and class initialization,
- object-orientation with class instances, instance methods, inheritance,
- exception handling,
- delegates together with events (including here for convenience also properties, indexers, attributes),
- concurrency (threads),
- so-called unsafe code with pointers and pointer arithmetic.

This yields a sequence of sublanguages $\mathrm{C}\sharp_{\mathcal{I}}$, $\mathrm{C}\sharp_{\mathcal{C}}$, $\mathrm{C}\sharp_{\mathcal{O}}$, $\mathrm{C}\sharp_{\mathcal{E}}$, $\mathrm{C}\sharp_{\mathcal{D}}$, $\mathrm{C}\sharp_{\mathcal{T}}$, $\mathrm{C}\sharp_{\mathcal{U}}$ which altogether describe the entire language C♯. Each language L in the sequence extends its predecessor and for each one we build a submachine $\textsc{ExecCsharp}_L$ which is a conservative (purely incremental) extension of its predecessor. The model ExecCsharp for the entire language C♯ is a composition of all submachines.

ExecCsharp ≡
  $\textsc{ExecCsharp}_I$
  $\textsc{ExecCsharp}_C$
  $\textsc{ExecCsharp}_O$
  $\textsc{ExecCsharp}_E$
  $\textsc{ExecCsharp}_T$
  $\textsc{ExecCsharp}_D$
  $\textsc{ExecCsharp}_U$

This approach supports a systematic piecemeal introduction of the numerous language constructs in teaching C♯ (or similar programming languages).

To keep the definition of the models succinct, we avoid tedious and routine rep-

etitions concerning language constructs which can be reduced in well-known ways to the core constructs in our models. Whenever instead of a direct formalization of a construct we use a syntactical translation to constructs dealt with in the core model, we have to justify that the translation is correct with respect to the semantics of the construct as intended by the standard. The ASM model we define provides a basis to rigorously formulate and mathematically prove the intended equivalence.[2] Since such a justification follows well-known patterns, it is skipped in this paper, but to remind the reader of the problem we usually mention it.

The handling of truly concurrent threads, not limited to interleaving or similar simple scheduling techniques, is closely related to the underlying memory model. Since the description of this memory model goes much beyond this paper, the submodel $C\sharp_{\mathcal{T}}$ and its further analysis is postponed to a separate paper [17].

By and large one can correctly understand an ASM as pseudo-code operating over abstract data (structures in the sense of logic). Therefore we skip a detailed definition of ASMs, which is available in textbook form in Chapter 2 of the AsmBook [18]. Since our paper is not a tutorial or manual on C♯, we restrict our explanations of language constructs to features a reader will appreciate who is already knowledgeable about the basic concepts of object-oriented programming. In a technical report [19] also the remaining details which are skipped in this paper are spelt out completely, together with further explanations and examples.

The paper is structured by the modularization we propose for the language description. The basic framework of our model is introduced in Sect. 2 together with the interpreter for the imperative kernel $C\sharp_{\mathcal{I}}$ of the language. Successively one more section is added for each model refinement to capture the related language extension. In general each section has a first part where the static assumptions of the model are formulated, followed by a second part which contains the dynamics expressed by the ASM transition rules operating on the corresponding state components. In general at each layer the interpreter consists of two submachines, one defining expression evaluation and one defining statement execution.

[2] One has to define an extension of the core model by a direct formalization of the construct in question and then to prove that this model is equivalent to the core model modulo the syntactical translation of the construct.

$$
\begin{array}{lll}
\mathit{Exp} & ::= & \mathit{Lit} \mid \mathit{Vexp} \mid \mathit{Uop}\ \mathit{Exp} \mid \mathit{Exp}\ \mathit{Bop}\ \mathit{Exp} \mid \mathit{Exp}\ \text{`?'}\ \mathit{Exp}\ \text{`:'}\ \mathit{Exp} \\
& & \mid (\ \mathit{Type}\ \text{`)'}\ \mathit{ExpSexp} \mid \text{`('}\ \mathit{Exp}\ \text{`)'} \mid \texttt{`checked'}\ \text{`('}\ \mathit{Exp}\ \text{`)'} \\
& & \mid \texttt{`unchecked'}\ \text{`('}\ \mathit{Exp}\ \text{`)'} \\
\mathit{Vexp} & ::= & \mathit{Loc} \\
\mathit{Sexp} & ::= & \mathit{Vexp}\ \texttt{`='}\ \mathit{Exp} \mid \mathit{Vexp}\ \mathit{Aop}\ \mathit{Exp} \mid \mathit{Vexp}\ \texttt{`++'} \mid \mathit{Vexp}\ \texttt{`--'} \\
\mathit{Uop} & ::= & \texttt{`+'} \mid \texttt{`-'} \mid \texttt{`!'} \mid \texttt{`\~{}'} \\
\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{`!='} \mid \texttt{`\&'} \mid \texttt{`\^{}'} \mid \texttt{`|'} \\
\mathit{Aop} & ::= & \texttt{`*='} \mid \texttt{`/='} \mid \texttt{`\%='} \mid \texttt{`+='} \mid \texttt{`-='} \mid \texttt{`<<='} \mid \texttt{`>>='} \mid \texttt{`\&='} \mid \texttt{`\^{}='} \mid \texttt{`|='} \\
\mathit{Stm} & ::= & \text{`;'} \mid \mathit{Sexp}\ \text{`;'} \mid \texttt{`break'}\ \text{`;'} \mid \texttt{`continue'}\ \text{`;'} \mid \texttt{`goto'}\ \mathit{Lab}\ \text{`;'} \\
& & \mid \texttt{`if'}\ \text{`('}\ \mathit{Exp}\ \text{`)'}\ \mathit{Stm}\ \texttt{`else'}\ \mathit{Stm} \\
& & \mid \texttt{`while'}\ \text{`('}\ \mathit{Exp}\ \text{`)'}\ \mathit{Stm} \mid \texttt{`do'}\ \mathit{Stm}\ \texttt{`while'}\ \text{`('}\ \mathit{Exp}\ \text{`)'} \\
& & \mid \texttt{`for'}\ \text{`('}\ [\mathit{Sexps}]\ \text{`;'}\ [\mathit{Exp}]\ \text{`;'}\ [\mathit{Sexps}]\ \text{`)'}\ \mathit{Stm} \\
& & \mid \texttt{`switch'}\ \text{`('}\ \mathit{Exp}\ \text{`)'}\ \text{`\{'}\ \{\mathit{Case}\ \{\mathit{Case}\}\ \mathit{Bstm}\ \{\mathit{Bstm}\}\}\ \text{`\}'} \\
& & \mid \texttt{`goto'}\ \texttt{`case'}\ \mathit{Cexp}\ \text{`;'} \mid \texttt{`goto'}\ \texttt{`default'}\ \text{`;'} \\
& & \mid \texttt{`checked'}\ \mathit{Block} \mid \texttt{`unchecked'}\ \mathit{Block} \mid \mathit{Block} \\
\mathit{Sexps} & ::= & \mathit{Sexp}\ \{\text{`,'}\ \mathit{Sexp}\} \\
\mathit{Case} & ::= & \texttt{`case'}\ \mathit{Cexp}\ \text{`:'} \mid \texttt{`default'}\ \text{`:'} \\
\mathit{Block} & ::= & \text{`\{'}\ \{\mathit{Bstm}\}\ \text{`\}'} \\
\mathit{Bstm} & ::= & \mathit{Type}\ \mathit{Loc}\ \text{`;'} \mid \texttt{`const'}\ \mathit{Type}\ \mathit{Loc}\ \texttt{`='}\ \mathit{Cexp}\ \text{`;'} \mid \mathit{Lab}\ \text{`:'}\ \mathit{Stm} \mid \mathit{Stm}
\end{array}
$$

Fig. 1. Grammar of expressions and statements in C♯$_I$.

2 The imperative core C♯$_I$

In this section we define the model for C♯$_I$, which defines the basic machinery of the ASM model for C♯. It describes the semantics of the sequential imperative core of C♯ with to be executed statements (appearing in method bodies) and to be evaluated expressions (appearing in statements) built using predefined operators over simple types. The computations of this interpreter are supposed to start with an arbitrary but fixed C♯ program. We separate syntax and compile-time matters from run-time issues by assuming that the program is given as an attributed syntax tree (i.e. annotated abstract syntax tree resulting from parsing and elaboration), trying to achieve model simplicity also by assuming some useful syntactical simplifications which will be mentioned as we build the model. Before defining the transition rules for the dynamic semantics of C♯$_I$, we formulate what has to be said about the static semantics.

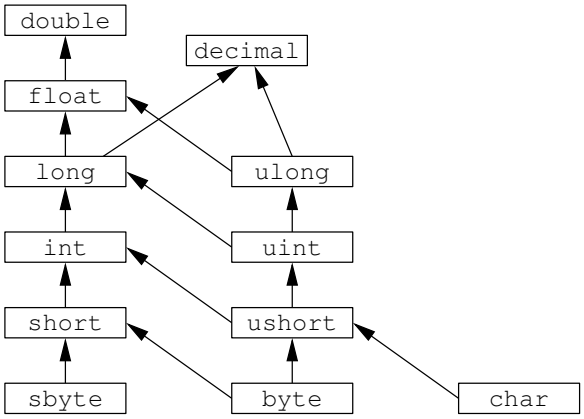


Fig. 2. The simple types of C♯$_\mathcal{I}$.

2.1 Static semantics of C♯$_\mathcal{I}$

We view the grammar in Fig. 1, which defines expressions and statements of the sublanguage C♯$_\mathcal{I}$, as defining also the corresponding ASM domains *Exp* and *Stm*. To avoid lengthy repetitions we include here already the distinctions between checked and unchecked expressions and blocks, though they are semantically irrelevant in the submodel C♯$_\mathcal{I}$ and start to play a role only with C♯$_\mathcal{E}$. The set *Vexp* of variable expressions (lvalues) consists in this model of the local variables only and will be refined below. *Sexp* denotes the set of statement expressions than can be used either as (result yielding) expressions or as (result discarding) statements, such as an assignment to a variable expression using '=' or an assignment operator from the set *Aop* or '++' or '--'. *Lit* denotes the set of literals, similarly for *Type*, *Lab* and the set *Cexp* of constant expressions whose value is known at compile time. When referring to the set of sequences of elements from a set *Item* we write *Items*, e.g. *Sexps* for the set of sequences of statement expressions. We usually write lower case letters e to denote elements of a set E, e.g. lit for elements of Lit.

The descriptions of implicit numeric conversions in [2, §13.1] and of binary numeric promotions in [2, §14.2.6] can be succinctly formulated as follows, using the type graph in Fig. 2 for the simple types of C♯, which are the types of C♯$_\mathcal{I}$ (for a classification of the types of C♯ see Fig. 4).

Definition 1 (Implicit conversion [2, §13.1]) *We say that there exists an* implicit numeric conversion *from type A to B (written $A \prec B$) iff there exists a finite, non-empty path of arrows from A to B in the type graph in Fig. 2. We write $A \preceq B$ for $A \prec B$ or $A = B$. A type C is called an* upper bound *of A and B iff $A \preceq C$ and $B \preceq C$. A type C is the* least upper bound *of A and B iff*

- *C is an upper bound of A and B and*

- $C \preceq D$ *for each upper bound* D *of* A *and* B.

We write $\sup(A, B)$ *for the least upper bound of* A *and* B *if it exists.*

We assume all the type constraints (on the operand and result values) and precedence conventions listed in [2] for the predefined (arithmetical, relational, bit and boolean logical) operators and the expression types. As usual each expression node *exp* in the attributed syntax tree has as attribute its compile-time type *type*(*exp*).

About type conversions at compile-time we assume that type casts are inserted in the syntax tree if necessary. For example, if a binary numeric operator *bop* is applied to arguments in e_1 *bop* e_2, then the least upper bound T of the types of e_1 and e_2 must exist and the expression is transformed into `(`T`)`e_1 *bop* `(`T`)`e_2.

Definition 2 (Binary numeric promotion [2, §14.2.6])
The binary numeric promotion consists of applying the following rules:

- *If the least upper bound of* A *and* B *exists, then*
  - *if* $\sup(A, B) \preceq$ `int`*, then* A *and* B *are converted to* `int`*,*
  - *otherwise,* A *and* B *are converted to* $\sup(A, B)$.
- *If the least upper bound of* A *and* B *does not exist, then a compile-time error occurs.*

We also assume the syntactical constraints for statements listed in [2], e.g. the following ones for blocks (where the *scope of a local variable* (*local constant*) is defined as the block in which it is declared, the *scope of a label* is the block in which the label is declared, and a local variable is identified by its name *and* the position of its declaration, so that in particular local variables with the same name in disjoint blocks are considered as different):

- It is not allowed to refer to a local variable (local constant) in a textual position that precedes its declaration.
- It is not allowed to declare another local variable or local constant with the same name in the scope of a local variable (local constant).
- It is not allowed for two labels with the same name to have overlapping scopes.
- A `goto` *Lab* must be in the scope of a label with name *Lab*.
- Expressions in *constant declarations* are evaluated at compile-time.

To simplify the exposition of our model we assume some standard syntactical reductions as indicated in Table 1. The correctness of these replacements with respect to [2] can easily be checked on the basis of our semantics model for C♯.

Control-flow analysis. During the static program analysis where the compiler has to verify that the given program is well-typed, predicates *reachable*

Table 1
Standard syntactical reductions.

exp_1 `&&` exp_2	exp_1 `?` exp_2 `:` `false`
exp_1 `||` exp_2	exp_1 `?` `true` `:` exp_2
`if (`exp`)` stm	`if (`exp`)` stm `else ;`
`++`$vexp$	$vexp$ `+= 1`
`--`$vexp$	$vexp$ `-= 1`
`int` x `= 1,` y`,` z `=` x `* 2;`	`int` x`;` x `= 1;` `int` y`;` `int` z`;` z `=` x `* 2;`
`for (`t loc `=` exp`;` tst`;` $step$`)` stm	`{` t loc`;` `for (`loc `=` exp`;` tst`;` $step$`)` stm `}`

and *normal* with the following intended meaning are computed for statements, using the type information contained in the attributed syntax tree as the result of parsing and elaboration:

$$\begin{aligned} \mathit{reachable}(\mathit{stm}) &\iff \mathit{stm} \text{ can be reached} \\ \mathit{normal}(\mathit{stm}) &\iff \mathit{stm} \text{ can terminate normally} \\ &\iff \text{the end point of } \mathit{stm} \text{ can be reached} \end{aligned}$$

One of the language design goals was to guarantee the following two properties for programs to be accepted by the compiler:

- during the program execution, only *reachable* positions are reached,
- normal termination happens only in *normal* positions.

These two properties are obtained by checking two sufficient conditions via so-called reachability rules, which can be inductively defined for $C\sharp_{\mathcal{I}}$ in Table 2 (similarly for `do`, `for`, `switch`).[3] For constant boolean expressions in conditional and while statements we assume that they are replaced in the abstract syntax tree by `true` or `false`.

Unreachable statements indicate programming errors and therefore generate compile-time warnings. Function bodies that can terminate normally generated compile-time errors, since at run-time execution could fall off the bottom of the code array.

Another language design goal was to achieve the type safety of well-typed C♯ programs, i.e. that a) variables at run-time contain values that are *compatible* with the declared types, and b) expressions are evaluated at run-time to values that are *compatible* with their compile-time types. Among the desired consequences of the type safety of a program one has that at run-time

[3] We include these rules here to place the corresponding natural language specification in [2] on a firm ground for a mathematical proof of the above two properties as part of a type safety proof for C♯.

Table 2
Reachability rules for C♯$_{\mathcal{I}}$.

s is a function body	$\Longrightarrow$	$reachable(s)$
$reachable($`;`$)$	$\Longrightarrow$	$normal($`;`$)$
$reachable(e$`;`$)$	$\Longrightarrow$	$normal(e$`;`$)$
$reachable($`{}`$)$	$\Longrightarrow$	$normal($`{}`$)$
$reachable($`{`$s \ldots$`}`$)$	$\Longrightarrow$	$reachable(s)$
$normal(s_i)$ in `{`$\ldots s_i\ s_{i+1} \ldots$`}`	$\Longrightarrow$	$reachable(s_{i+1})$
$reachable($`goto`$\ l$`;`$)$ in `{`$\ldots l$`:`$s \ldots$`}`	$\Longrightarrow$	$reachable(l$`:`$s)$
$normal(s)$	$\Longrightarrow$	$normal($`{`$\ldots s$`}`$)$
$reachable($`if (`e`)`$\ s_1$ `else`$\ s_2) \wedge e \neq$ `false`	$\Longrightarrow$	$reachable(s_1)$
$reachable($`if (`e`)`$\ s_1$ `else`$\ s_2) \wedge e \neq$ `true`	$\Longrightarrow$	$reachable(s_2)$
$normal(s_1) \vee normal(s_2)$	$\Longrightarrow$	$normal($`if (`e`)`$\ s_1$ `else`$\ s_2)$
$reachable($`while (`e`)`$\ s) \wedge e \neq$ `false`	$\Longrightarrow$	$reachable(s)$
$reachable($`while (`e`)`$\ s) \wedge e \neq$ `true`	$\Longrightarrow$	$normal($`while (`e`)`$\ s)$
$reachable($`break;`$)$ in s	$\Longrightarrow$	$normal($`while (`e`)`$\ s)$

its variables will never contain *undefined* values, that there are no *dangling* references, that the program cannot *corrupt* the memory, and that the dynamic *method lookup* always succeeds. Using the notation explained in the next section such invariants can be made precise and be proven to hold under appropriate assumptions. [4]

To guarantee the type safety the compiler checks a sufficient condition computing predicates *before*, *after* (for occurrences of statements and expressions in a function body) and *true*, *false* (for the two possible evaluation results of boolean expressions), which implement the so-called definite assignment rules to assure that a variable is *definitely assigned* before its value is used. The situation is illustrated in Fig. 3. Unfortunately the picture does not reflect reality. Microsoft has decided that in verified IL (intermediate language) code local variables are initialized by the run-time system with zero values. [5] Hence, also

[4] For example the following invariants can be proved to hold at run-time: a) $before(pos) \subseteq Defined$ where $Defined = \{x \in Loc \mid mem(locals(x)) \neq Undef\}$, b) $after(pos) \subseteq Defined$ if $values(pos) = Norm$ or $values(pos) \in Value$. Specifically for boolean expressions holds $true(pos) \subseteq Defined$ if $values(pos) = True$, the same for *false*. Such proofs can be carried out on the basis of the model developed in this paper, using the pattern developed in [1, Ch.8] for proving that Java is type safe. For a different approach see [20].

[5] Maybe to simplify the job of the JIT verifiers, as one of our referees suggested.

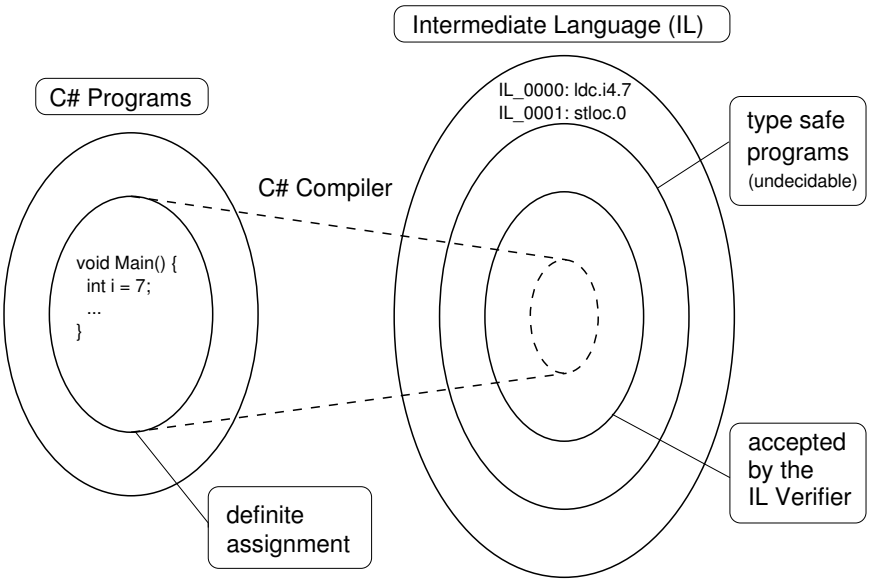


Fig. 3. Definite assignment and IL verification.

source code programs that do not fulfill the definite assignment constraints are accepted by the IL verifier.

A variable occurring in a position is called definitely assigned there, if on every execution path leading to that position (in the abstract syntax tree) a value is assigned to the variable. Thus the intended meaning of the above predicates is as follows, where by "elaboration" of an *item* we mean "execution", if *item* is a statement, and "evaluation" if it is an expression.

$x \in \mathit{before}(\mathit{item})$: x is definitely assigned *before* the elaboration of *item*

$x \in \mathit{after}(\mathit{item})$: x is definitely assigned *after* normal elaboration of *item*

$x \in \mathit{true}(\mathit{exp})$: x is definitely assigned *after exp* evaluates to *true*

$x \in \mathit{false}(\mathit{exp})$: x is definitely assigned *after exp* evaluates to *false*

To provide a basis for a mathematical analysis, we turn the verbally stated definite assignment rules of [2, §12.3.3] into a precise set of equational constraints, where $\mathit{vars}(\mathit{stm}) = \{x \mid \mathit{stm} \text{ is in the scope of } x\}$.

Table 3 contains the constraints for the statements. Table 4 contains the equations for specific boolean expressions, which are imposed for the eager (short-circuit) evaluation of boolean expressions. Note that there is no equation in Table 4 for *after* sets since by definition $\mathit{after}(\mathit{exp}) = \mathit{true}(\mathit{exp}) \cap \mathit{false}(\mathit{exp})$. If *exp* is a boolean expression which is not an instance of one of the expressions in Table 4, then the following are constraints for *exp*: $\mathit{true}(\mathit{exp}) = \mathit{after}(\mathit{exp})$ and $\mathit{false}(\mathit{exp}) = \mathit{after}(\mathit{exp})$.

Table 3
Definite assignment for statements.

s is a function body	$before(s) = \emptyset$
`;`	$after(\texttt{;}) = before(\texttt{;})$
exp`;`	$before(exp) = before(exp\,\texttt{;})$, $after(exp\,\texttt{;}) = after(exp)$
`break;`	$after(\texttt{break;}) = vars(\texttt{break;})$
`continue;`	$after(\texttt{continue;}) = vars(\texttt{continue;})$
`goto` l`;`	$after(\texttt{goto}\ l\,\texttt{;}) = vars(\texttt{goto}\ l\,\texttt{;})$
$stm =$ `{`$s_1 \ldots s_n$`}`	$before(s_1) = before(stm)$, $after(stm) = after(s_n)$, $before(s_{i+1}) = after(s_i) \cap goto(s_{i+1})$ where $goto(l\!:\!s) =$ $\quad\bigcap\{before(\texttt{goto}\,l\,\texttt{;}) \mid \texttt{goto}\,l\,\texttt{;}$ reachable in $stm\}$ and $goto(s) = vars(s)$ if s is not a labeled statement
$stm =$ `if (`e`)` s_1 `else` s_2	$before(e) = before(stm)$, $before(s_1) = true(e)$ $before(s_2) = false(e)$, $after(stm) = after(s_1) \cap after(s_2)$
$stm =$ `while (`e`)` s	$before(e) = before(stm)$, $before(s) = true(e)$, $after(stm) = false(e) \cap break(s)$ where $break(s) = \bigcap\{before(\texttt{break;}) \mid \texttt{break;}$ reachable in $s\}$

Table 4
Definite assignment for boolean expressions.

`true`	$true(\texttt{true}) = before(\texttt{true})$, $false(\texttt{true}) = vars(\texttt{true})$
`false`	$true(\texttt{false}) = vars(\texttt{false})$, $false(\texttt{false}) = before(\texttt{false})$
$exp =$ `!`e	$before(e) = before(exp)$, $true(exp) = false(e)$ $false(exp) = true(e)$
$exp = (e_1$ `&&` $e_2)$	$before(e_1) = before(exp)$, $before(e_2) = true(e_1)$, $true(exp) = true(e_2)$, $false(exp) = false(e_1) \cap false(e_2)$
$exp = (e_1$ `||` $e_2)$	$before(e_1) = before(exp)$, $before(e_2) = false(e_1)$, $true(exp) = true(e_1) \cap true(e_2)$, $false(exp) = false(e_2)$
$exp = (e_0$ `?` e_1 `:` $e_2)$	$before(e_0) = before(exp)$, $before(e_1) = true(e_0)$ $before(e_2) = false(e_0)$, $true(exp) = true(e_1) \cap true(e_2)$ $false(exp) = false(e_1) \cap false(e_2)$

Table 5 contains the equations for non-boolean expressions. In all other cases, if exp is an expression which has the *direct subexpressions* $e_1, e_2, \ldots, e_n$, then the left-to-right evaluation scheme yields

(1) $before(e_1) = before(exp)$
(2) $before(e_{i+1}) = after(e_i)$ for $i \in [1\,..\,n-1]$
(3) $after(exp) = after(e_n)$

Table 5
Definite assignment for abitrary expressions.

loc	$loc \in before(loc)$, $after(loc) = before(loc)$
lit	$after(lit) = before(lit)$
$exp = (loc \texttt{ = } e)$	$before(e) = before(exp)$, $after(exp) = after(e) \cup \{loc\}$
$exp = (loc\ op\texttt{=}\ e)$	$loc \in before(exp)$, $before(e) = before(exp)$ $after(exp) = after(e)$
$exp = (e_0 \texttt{ ? } e_1 \texttt{ : } e_2)$	$before(e_0) = before(exp)$, $before(e_1) = true(e_0)$ $before(e_2) = false(e_0)$, $after(exp) = after(e_1) \cap after(e_2)$

Due to the goto statement the above constraints do not specify in a unique way the sets of variables that have to be considered as definitely assigned. Consider the following block (from [21]):

```
{ int i = 1 ; L : goto L ; }
```

Then the constraints of the definite assignment analysis are satisfied for both $before($`L: goto L;`$) = \emptyset$ and $before($`L: goto L;`$) = \{i\}$. Hence during the analysis the greatest sets of variables that satisfy the constraints for *before* and *after* have to be computed (cf. [21]). For blocks without goto statements, however, it can be proved from the above axioms that the *before* set determines the *after* set in a unique way.

2.2 Dynamic semantics for $C\sharp_{\mathcal{I}}$

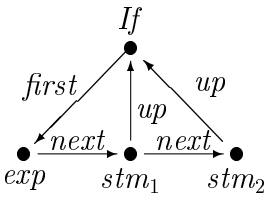
The dynamic semantics for $C\sharp_{\mathcal{I}}$ describes the effect of statement execution and expression evaluation upon the program state, so that the transition rule for $C\sharp_{\mathcal{I}}$ (the same for its extensions) has the form

ExecCsharp$_I \equiv$
  ExecCsharpExp$_I$
  ExecCsharpStm$_I$

The first subrule defines one execution step in the evaluation of expressions; the second subrule defines one step in the execution of statements.

To make the further model refinements possible via purely incremental extensions, our definition proceeds by walking through the attributed syntax tree and computing at each node the effect of the program construct attached to the node. We formalize the walk by a cursor ▶, whose position in the tree – represented by a dynamic function pos: Pos – is updated using static tree functions, leading from a node in the tree down to its *first* child, from there to the *next* brother or *up* to the parent node (if any), as illustrated by the following

self-explanatory example. The moves of *pos* contain implicitly the control-flow graph of C♯. A function $label\colon Pos \to Label$ decorates nodes with the information which identifies the grammar rule associated to the node. For the sake of notational succinctness we use concrete syntax of C♯ to describe the labels, thus avoiding the explicit introduction of auxiliary non-terminals the reader probably does not want to see. In the following example the four possible cursor positions are reachable from the root by following the tree functions *first*, *next* and *up*. The *label* of the root node is the auxiliary non-terminal *If*, identifying the grammar rule which produces in one step `if (`*exp*`)` stm_1 `else` stm_2.



`if (`*exp*`)` stm_1 `else` stm_2

For updating the values of local variables in the memory we use two dynamic functions $locals\colon Loc \to Adr$ and $mem\colon Adr \to SimpleValue \cup \{Undef\}$, which assign to local variables memory addresses and store the values there. Since in $C\sharp_{\mathcal{I}}$ the values are of simple types, the equation $Value = SimpleValue \cup Adr$ holds, which will be refined in the extended models to include also references and structs. The uniquely identified local variables are modeled by stipulating $Loc = Identifier \times Pos$, where Pos is the set of positions in the abstract syntax tree.

The indirection through memory addresses is not really needed in $C\sharp_{\mathcal{I}}$. In $C\sharp_{\mathcal{I}}$ one could assign values directly to local variables without storing them in an abstract memory. The addresses, however, are needed later for call-by-reference with `ref` and `out` parameters (one of the major differences between C♯ and Java from the modelling point of view).

Statements can terminate normally or abruptly, where in $C\sharp_{\mathcal{I}}$ the reasons of abruption are from the set $Abr = Break \mid Continue \mid Goto(Lab)$, to be refined for the extended models. We use a dynamic function $values\colon Pos \to Result$ to store intermediate evaluation results from the set

$$Result = Value \cup Abr \cup \{Undef, Norm\}.$$

For the initial state we assume

- $mem(i) = Undef$ for every $i \in Adr$
- pos = root position of the attributed syntax tree
- $locals(x) \in Adr$ for every variable x [6]

[6] This amounts to assuming that the compiler chooses an address for each variable.

As intermediate *values* at a position p the cursor is at or is passing to, the computation may yield directly a simple value; at *AddressPos*itions as defined below it may yield an address; but it may also yield a *memValue* which has to be retrieved indirectly via the given address (where for $\mathrm{C}\sharp_{\mathcal{I}}$ the memory value of a given type t at a given address adr is defined by $memValue(adr, t) = mem(adr)$; the parameter t will become relevant only in the refinement of *memValue* in $\mathrm{C}\sharp_{\mathcal{O}}$ and $\mathrm{C}\sharp_{\mathcal{U}}$). This is described by the following two macros:

YIELD$(val, p) \equiv$
  $values(p) := val$
  $pos := p$

YIELDINDIRECT$(adr, p) \equiv$
  if $AddressPos(p)$ **then** YIELD(adr, p)
  else YIELD$(memValue(adr, type(p)), p)$

We will use the macros in the two forms YIELD$(val) \equiv$ YIELD(val, pos) and YIELDUP$(val) \equiv$ YIELD$(val, up(pos))$. Similarly we have two forms also for the second macro: YIELDINDIRECT(adr) and YIELDUPINDIRECT(adr).

Being in a context where an address and not a value is required can be defined as follows:

$$AddressPos(\alpha) \iff FirstChild(\alpha) \wedge (label(up(\alpha)) \in \{\texttt{++}, \texttt{--}\} \vee label(up(\alpha)) \in Aop)$$
where $FirstChild(\alpha) \iff first(up(\alpha)) = \alpha$

To further reduce any notational overhead not needed by the human reader, in spelling out the rules below we identify positions with the occurrences of the syntactical constructs nodes are decorated with. This explains updates like $pos := exp$ or $pos := stm$, which are used as shorthand for updating pos to the node labeled with the corresponding occurrence of exp respectively stm.[7] Furthermore, for a succinct formulation we use a macro $context(pos)$ to describe 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 rule) to select the appropriate computation step. If the elaboration of the subtree at the position pos has not yet started, then $context(pos)$ is the construct encoded by the labels of pos and of its children. Otherwise, if pos carries already its result in *values*, $context(pos)$ is the pseudo-construct encoded by the labels of the parent node of pos and of its children after replacing the already evaluated constructs by their *values* in the corresponding node. This explains notations like $uop \blacktriangleright val$

[7] An identification of this kind, which is common in mathematics, has clearly to be resolved in an executable version of the model. See for example the formulation of the ASM model for Java in [1].

to describe the *context* of *pos*, where *pos* is marked with the cursor (▶), resulting from the successful evaluation of the argument *exp* of the construct *uop exp* (encoded by *up*(*pos*) and its child *pos*), just before *uop* is applied to *val* to YieldUp(*Apply*(*uop*, *val*)).

It thus remains to define the two submachines for expression evaluation and statement execution. This is done in a modular fashion, grouping behaviorally similar instructions into one parameterized instruction. [8]

Expression evaluation rules. We are now ready to define the machine ExecCsharpExp$_I$ in a compositional way, namely proceeding expression-wise: for each syntactical form of expressions there is a set of rules covering each intermediate phase of their evaluation. The machine passes control from unevaluated expressions to the appropriate subexpressions until an atom (a literal or a local variable) is reached. It can continue its computation only as long as no operator exception occurs, as a consequence it does not distinguish between checked and unchecked expression evaluation – the extension by rules to handle exceptions is defined in the model extension C♯$_\mathcal{E}$. The expressions for numeric casts will be refined in C♯$_\mathcal{O}$ and in C♯$_\mathcal{E}$. The macro WriteMem(*adr*, *t*, *val*) denotes here *mem*(*adr*) := *val*; it will be refined in the model for C♯$_\mathcal{O}$.

ExecCsharpExp$_I$ ≡ **match** *context*(*pos*)
  lit → Yield(*ValueOfLiteral*(*lit*))
  loc → YieldIndirect(*locals*(*loc*))

  uop exp → *pos* := *exp*
  uop ▶*val* → **if** ¬*UopException*(*uop*, *val*) **then** YieldUp(*Apply*(*uop*, *val*))

  exp$_1$ *bop exp*$_2$ → *pos* := *exp*$_1$
  ▶*val bop exp* → *pos* := *exp*
  val$_1$ *bop* ▶*val*$_2$ → **if** ¬*BopException*(*bop*, *val*$_1$, *val*$_2$) **then**
    YieldUp(*Apply*(*bop*, *val*$_1$, *val*$_2$))

  exp$_0$? *exp*$_1$: *exp*$_2$ → *pos* := *exp*$_0$
  ▶*val* ? *exp*$_1$: *exp*$_2$ → **if** *val* **then** *pos* := *exp*$_1$ **else** *pos* := *exp*$_2$
  True ? ▶*val* : *exp* → YieldUp(*val*)
  False ? *exp* : ▶*val* → YieldUp(*val*)

  loc = *exp* → *pos* := *exp*
  loc = ▶*val* → {WriteMem(*locals*(*loc*), *type*(*loc*), *val*), YieldUp(*val*)}

  (*type*) *exp* → *pos* := *exp*
  (*type*) ▶*val* → **if** *type*(*pos*) ∈ *NumericType* ∧ *type* ∈ *NumericType* **then**
    if ¬*UopException*(*type*, *val*) **then**
      YieldUp(*Convert*(*type*, *val*))

[8] The specializations can be regained instruction-wise by mere parameter expansion, a form of refinement that is easily proved to be correct.

$$
\begin{array}{l}
vexp\ \texttt{op=}\ exp \ \rightarrow pos := vexp \\
\blacktriangleright adr\ \texttt{op=}\ exp \rightarrow pos := exp \\
adr\ \texttt{op=}\ \blacktriangleright val \rightarrow \textbf{let}\ t = type(up(pos))\ \textbf{and}\ v = memValue(adr, t)\ \textbf{in} \\
\qquad\qquad\qquad \textbf{if}\ \neg BopException(op, v, val)\ \textbf{then} \\
\qquad\qquad\qquad\quad \textbf{let}\ w = Apply(op, v, val)\ \textbf{and}\ r = Convert(t, w)\ \textbf{in} \\
\qquad\qquad\qquad\qquad \{\textsc{WriteMem}(adr, t, r),\ \textsc{YieldUp}(r)\}
\end{array}
$$

$$
\begin{array}{l}
vexp\ op \ \rightarrow pos := vexp // \text{ for postfix operators } op \in \{\texttt{++}, \texttt{--}\} \\
\blacktriangleright adr\ op \rightarrow \textbf{let}\ old = memValue(adr, type(pos))\ \textbf{in} \\
\qquad\qquad \textbf{if}\ \neg UopException(op, old)\ \textbf{then} \\
\qquad\qquad\quad \textsc{WriteMem}(adr, type(up(pos)) \\
\qquad\qquad\quad Apply(op, old)) \\
\qquad\qquad\quad \textsc{YieldUp}(old)
\end{array}
$$

$$
\begin{array}{ll}
\texttt{checked}(exp) & \rightarrow pos := exp \\
\texttt{checked}(\blacktriangleright val) & \rightarrow \textsc{YieldUp}(val) \\
\texttt{unchecked}(exp) & \rightarrow pos := exp \\
\texttt{unchecked}(\blacktriangleright val) & \rightarrow \textsc{YieldUp}(val)
\end{array}
$$

Being in a checked context is used to define whether operators throw an overflow exception (in which case a rule will be added in the model for $\mathrm{C}\sharp_{\mathcal{E}}$). The general rule is that operators for the type `decimal` always throw overflow exceptions whereas operators for integral types only throw overflow exceptions in a checked context except for the division by zero. By default every position is unchecked, unless explicitly declared otherwise.

$$
\begin{array}{l}
Checked(\alpha) \iff label(\alpha) = Checked \vee \\
\quad (label(\alpha) \neq Unchecked \wedge up(\alpha) \neq Undef \wedge Checked(up(\alpha))) \\
UopException(uop, val) \iff Checked(pos) \wedge Overflow(uop, val) \\
BopException(bop, val_1, val_2) \iff \\
\quad DivisonByZero(bop, val_2) \vee DecimalOverflow(bop, val_1, val_2) \vee \\
\quad (Checked(pos) \wedge Overflow(bop, val_1, val_2))
\end{array}
$$

Statement execution rules. The machine $\textsc{ExecCsharpStm}_I$ is defined below statement-wise. It transfers control from structured statements to the appropriate substatements, until the current statement has been computed normally or abrupts the computation. Abruptions trigger the control to propagate through all the enclosing statements up to the target labeled statement. The concept of propagation is defined for $\mathrm{C}\sharp_{\mathcal{I}}$ in such a way that in the refined models it is easily extended to abruptions due to return from procedures or to exceptions.[9] In case of a new execution of the body of a while statement, the previously computed intermediate results have to be cleared.[10] For

[9] For $\mathrm{C}\sharp_{\mathcal{I}}$ alone it would be simpler to transfer control directly by updating *pos* to the value of a corresponding static function.

[10] ClearValues is needed in the present rule formulation, which is close to an

the sake of brevity we skip the analogous transition rules for statements `do`, `for`, `switch`, `goto case`, `goto default`. Since we formulate the model for the human reader, we also use the ...-notation, for example in the rules for abruption or for sequences of block statements. This avoids having to fuss with an explicit formulation of the context, typically determined by a walk through a list. This simplification, which is tailored for the human reader, can easily be resolved for an executable model version without increasing the number of rules.

$\textsc{ExecCsharpStm}_I \equiv$ **match** $context(pos)$

; $\rightarrow \textsc{Yield}(Norm)$

exp ; $\rightarrow pos := exp$
$\blacktriangleright val$; $\rightarrow \textsc{YieldUp}(Norm)$

`break;` $\rightarrow \textsc{Yield}(Break)$
`continue;` $\rightarrow \textsc{Yield}(Continue)$
`goto` lab ; $\rightarrow \textsc{Yield}(Goto(lab))$

`if (`exp`)` stm_1 `else` stm_2 $\rightarrow pos := exp$
`if (`$\blacktriangleright val$`)` stm_1 `else` stm_2 $\rightarrow$ **if** val **then** $pos := stm_1$ **else** $pos := stm_2$
`if (`$True$`)` $\blacktriangleright Norm$ `else` stm $\rightarrow \textsc{YieldUp}(Norm)$
`if (`$False$`)` stm `else` $\blacktriangleright Norm$ $\rightarrow \textsc{YieldUp}(Norm)$

`while (`exp`)` stm $\rightarrow pos := exp$
`while (`$\blacktriangleright val$`)` stm $\rightarrow$ **if** val **then** $pos := stm$
else $\textsc{YieldUp}(Norm)$
`while (`$True$`)` $\blacktriangleright Norm$ $\rightarrow \{pos := up(pos), \textsc{ClearValues}(up(pos))\}$
`while (`$True$`)` $\blacktriangleright Break$ $\rightarrow \textsc{YieldUp}(Norm)$
`while (`$True$`)` $\blacktriangleright Continue$ $\rightarrow \{pos := up(pos), \textsc{ClearValues}(up(pos))\}$
`while (`$True$`)` $\blacktriangleright abr$ $\rightarrow \textsc{YieldUp}(abr)$

$type$ loc ; $\rightarrow \textsc{Yield}(Norm)$

lab : stm $\rightarrow pos := stm$
lab : $\blacktriangleright Norm$ $\rightarrow \textsc{YieldUp}(Norm)$

`checked` $block$ $\rightarrow pos := block$
`checked` $\blacktriangleright Norm$ $\rightarrow \textsc{YieldUp}(Norm)$
`unchecked` $block$ $\rightarrow pos := block$
`unchecked` $\blacktriangleright Norm$ $\rightarrow \textsc{YieldUp}(Norm)$

... $\blacktriangleright abr$ ... $\rightarrow$ **if** $up(pos) \neq Undef \wedge PropagatesAbr(up(pos))$ **then**
$\textsc{YieldUp}(abr)$

---

executable format. In a more abstract SOS-style, as used for the Java model in [1], it wouldn't be necessary because there the intermediate *values* can be written into a dynamic function for the still to be executed rest program.

```
{ }                      → YIELD(Norm)
{stm ...}                → pos := stm
{... ▶Norm}              → YIELDUP(Norm)
{... ▶Norm stm ...}      → pos := stm
{... ▶Goto(l) ...}       → let α = GotoTarget(first(up(pos)), l)
                             if α ≠ Undef then
                               {pos := α, CLEARVALUES(up(pos))}
                             else YIELDUP(Goto(l))
{... ▶abr ...}           → YIELDUP(abr)
```

In $C\sharp_{\mathcal{I}}$ abruptions are propagated upwards except at the following statements:

$$PropagatesAbr(\alpha) \iff label(\alpha) \notin \{Block, While, Do, For, Switch\}$$

To compute the target of a label in a list of block statements we define:

```
GotoTarget(α, l) =
  if label(α) = Lab(l) then α
  elseif next(α) = Undef then Undef
  else GotoTarget(next(α), l)
```

The auxiliary macro CLEARVALUES(α) to clear all values in the subtree at position α can be defined by recursion as follows, proceeding from top to bottom and from left to right:[11]

```
CLEARVALUES(α) ≡
  values(α) := Undef
  if first(α) ≠ Undef then CLEARVALUESSEQ(first(α))

CLEARVALUESSEQ(α) ≡
  CLEARVALUES(α)
  if next(α) ≠ Undef then CLEARVALUESSEQ(next(α))
```

3 $C\sharp_{\mathcal{C}}$: refining $C\sharp_{\mathcal{I}}$ by static class features

In this section we refine the imperative core $C\sharp_{\mathcal{I}}$ to $C\sharp_{\mathcal{C}}$ by adding classes (modules) concentrating upon their static features (static fields, methods, constructors), including their initialization and the parameter mechanism that provides value, `ref` and `out` parameters. For such a refinement we a) extend the ASM

[11] Intuitively it should be clear that the execution of this recursive ASM provides simultaneously – in one step – the set of all updates of all its recursive calls, as is needed here for the clearing purpose; see [22] for a precise definition.

universes and functions, or introduce new ones, to reflect the grammar extensions for expressions and statements, b) add the appropriate constraints needed for the static analysis of the new items (type constraints, definite assignment rules), c) extend some of the macros, e.g. $PropagatesAbr(\alpha)$, to make them work also for the newly occurring cases, d) add rules which define the semantics of the new instructions that operate over the new domains.

3.1 Static semantics of $C\sharp_C$

In $C\sharp_C$ a program is a set of compilation units, each coming with "using directives" and declarations of names spaces (including a global namespace) and types (for classes and interfaces[12]) in the global namespace. For simplicity of exposition we disregard "using" directives and nested namespaces by assuming everywhere the adoption of (equivalent) fully qualified names. The precise syntax of classes and their static members, the rules for the accessibility of types and members via the access modifiers (public, internal, protected, private) and illustrating examples are spelt out in [19]. We define here the extension of the grammars for *Vexp*, *Sexp*, *Stm* and thereby of the corresponding ASM domains, which reflects the introduction of sets of *Class*es with static *Field*s and static *Meth*ods in $C\sharp_C$. The new set *Arg* of arguments appearing here reflects that besides value parameters also `ref` and `out` parameters can be used.

$$
\begin{aligned}
\mathit{Vexp} &::= \ldots \mid \mathit{Field} \mid \mathit{Class}\ \text{`.'}\ \mathit{Field} \\
\mathit{Sexp} &::= \ldots \mid \mathit{Meth}\ \texttt{(}\ [\mathit{Args}]\ \texttt{)} \mid \mathit{Class}\ \text{`.'}\ \mathit{Meth}\ \texttt{(}\ [\mathit{Args}]\ \texttt{)} \\
\mathit{Arg} &::= \mathit{Exp} \mid \text{`}\texttt{ref}\text{'}\ \mathit{Vexp} \mid \text{`}\texttt{out}\text{'}\ \mathit{Vexp} \\
\mathit{Args} &::= \mathit{Arg}\ \{\text{`,'}\ \mathit{Arg}\} \\
\mathit{Stm} &::= \ldots \mid \text{`}\texttt{return}\text{'}\ \mathit{Exp}\ \text{`;'} \mid \text{`}\texttt{return}\text{'}\ \text{`;'}
\end{aligned}
$$

The type constraints for the new expressions and the return statement are spelt out in [19]. The difference between `ref` and `out` parameters at function calls and in function bodies is reflected by including as *AddressPos*itions all nodes whose parent node is labeled by `ref` or `out` and by adding the following definite assignment constraints:

- `ref` arguments must be definitely assigned *before* the function call.
- `out` arguments are definitely assigned *after* the function call.
- `ref` parameters and value parameters of a function are definitely assigned at the beginning of the function body.
- `out` parameters must be definitely assigned when the function returns.

[12] Note that struct and enum types and delegates are introduced by further refinement steps below.

Therefore the definite assignment constraints for expressions are extended by the following constraints for general argument expressions in function calls and for `ref` and `out` argument expressions:

- For $exp = M\texttt{(}args\texttt{)}$:
  - $before(args) = before(exp)$
  - $RefParams(args) \subseteq after(args)$
  - $after(exp) = after(args) \cup OutParams(args)$
- For $exp = (\texttt{ref}\ e)$ or $exp = (\texttt{out}\ e)$:
  - $before(e) = before(exp)$
  - $after(exp) = after(e)$

The definite assignment constraints for statements are extended for function bodies and return statements as follows:

- If s is the body of M, then
  - $before(s) = ValueParams(M) \cup RefParams(M)$
- If $stm = \texttt{return;}$ is in M, then
  - $OutParams(M) \subseteq before(stm)$
  - $after(stm) = vars(stm)$
- If $stm = \texttt{return}\ e\texttt{;}$ is in M, then
  - $before(e) = before(stm)$
  - $OutParams(M) \subseteq after(e)$
  - $after(stm) = vars(stm)$

The presence of to-be-initialized classes and of method calls is reflected by the introduction of new universes to denote methods, the initialization status of a type (which will be refined below by exceptions) and the sequence of still active method calls (frame stack):

$$Meth = Type \times Msig$$

$$TypeState = Linked \mid InProgress \mid Initialized$$

$$Frame = Meth \times Pos \times Locals \times Values$$

where $Values = (Pos \to Result)$ and $Locals = (Loc \to Adr)$

A method signature *Msig* consists of the name of a method plus the sequence of types of the arguments of the method. A method is uniquely determined by the type in which it is declared and its signature. The reasons for abruptions are extended by method return:

$$Abr = \ldots \mid Return \mid Return(Value)$$

To dynamically handle the addresses of static fields (global or class variables), the initialization state of types, the current method, the execution stack and the (initially) to be initialized type we use the following new dynamic functions:

$$\begin{array}{ll} \mathit{globals}\colon \mathit{Type} \times \mathit{Field} \to \mathit{Adr} & \mathit{frames}\colon \mathit{List}(\mathit{Frame}) \\ \mathit{typeState}\colon \mathit{Type} \to \mathit{TypeState} & \mathit{meth}\colon \mathit{Meth} \end{array}$$

We extend the stipulations for the initial state as follows:

- $\mathit{typeState}(c) = \mathit{Linked}$ for each class c
- $\mathit{meth} = \mathit{EntryPoint}$::`Main()` [*EntryPoint* is the main class]
- $\mathit{pos} = \mathit{body}(\mathit{meth})$ [The root position of the body]
- $\mathit{locals} = \mathit{values} = \emptyset$ and $\mathit{frames} = []$

The submachine ExecCsharp$_C$ extends the machine ExecCsharp$_I$ for C♯$_\mathcal{I}$ by additional rules for the evaluation of the new expressions and for the execution of return statements. In the same way the further refinements in the sections below consist in the parallel addition of appropriate submachines.

ExecCsharp$_C \equiv$
  ExecCsharp$_I$
  ExecCsharpExp$_C$
  ExecCsharpStm$_C$

Expression evaluation rules. The rules for class field evaluation in the submachine ExecCsharpExp$_C$ are analogous to those for the evaluation of local variables in ExecCsharpExp$_I$, except for using *globals* instead of *locals* and for the additional clause for class initialization. The rules for method calls use the macro InvokeStatic defined below and reflect that the arguments are evaluated from left to right.[13]

ExecCsharpExp$_C \equiv$ **match** $\mathit{context}(\mathit{pos})$
  $c.f \to$ **if** $\mathit{Initialized}(c)$ **then** YieldIndirect($\mathit{globals}(c::f)$)
    else Initialize(c)

[13] These are the rules to be modified in case one wants to specify another evaluation order for expressions, involving the use of the ASM **choose** construct if some non-deterministic choice has to be formulated. For a discussion of such model variations we refer to [23] where an ASM model is developed which can be instantiated to capture the different expression evaluation strategies in Ada95, C, C++, Java, C♯ and Fortran.

$c.f$ `=` $exp \;\rightarrow pos := exp$
$c.f$ `=` $\blacktriangleright val \rightarrow$ **if** $Initialized(c)$ **then**
WriteMem($globals(c::f), type(c::f), val$)
YieldUp(val)
else Initialize(c)
$c.m$`(`$args$`)` $\rightarrow pos := $`(`$args$`)`
$c.m\blacktriangleright$`(`$vals$`)` $\rightarrow$ InvokeStatic($c::m, vals$)

`ref` $vexp \rightarrow pos := vexp$
`ref` $\blacktriangleright adr \rightarrow$ YieldUp(adr)

`out` $vexp \rightarrow pos := vexp$
`out` $\blacktriangleright adr \rightarrow$ YieldUp(adr)

`()` $\rightarrow$ Yield([])
`(`arg`,` ... `)` $\rightarrow pos := arg$
`(`val_1`,` ... `,`$\blacktriangleright val_n$`)` $\rightarrow$ YieldUp($[val_1, \ldots, val_n]$)
`(` ... $\blacktriangleright val$`,`arg ... `)` $\rightarrow pos := arg$

The macro InvokeStatic invokes the method if the initialization of its class is not triggered, otherwise it initializes the class. The initialization of a class (or struct, see Sect. 4) is not triggered if the class is already initialized.[14] For methods which are not declared external, InvokeMethod updates the frame stack and the current frame in the expected way, allocating via InitLocals for every local variable or value parameter a new address and copying every value argument there. Since we will also have to deal with external methods – whose declaration includes an `extern` modifier and which may be implemented using a language other than C♯ – we provide here for their invocation a submachine InvokeExtern, to be defined separately depending on the class of external (e.g. library) methods.[15] The predicate *StaticCtor* recognizes static class constructors; their implicit call interrupts the member access at *pos*, to later return to the evaluation of *pos* instead of $up(pos)$. We separate the current frame—consisting of *meth*, *pos*, *locals* and *values*—from the stack of such frames to notationally smoothen the transition from $\mathrm{C}\sharp_{\mathcal{I}}$ to $\mathrm{C}\sharp_{\mathcal{C}}$.

InvokeStatic($c::m, vals$) ≡
if not $TriggerInit(c)$ **then** InvokeMethod($c::m, vals$)
else Initialize(c)
where $TriggerInit(c) = \neg Initialized(c)$

[14] As analyzed in [13], it is also not triggered if the class is marked with the implementation flag *beforefieldinit*, indicating that the reference of the static method does not trigger the class (or struct) initialization. If one wants to model this flag the definition has to be refined to $TriggerInit(c) = \neg Initialized(c) \wedge \neg beforefieldinit(c)$ and furthermore in Sect. 5.)
[15] For an illustration of this use of external methods see below the model for delegates.

```
InvokeMethod(c::m, vals) ≡
  if extern ∈ modifiers(c::m) then InvokeExtern(c::m, vals)
  else let p = if StaticCtor(c::m) then pos else up(pos) in
    frames := push(frames, (meth, p, locals, values))
    meth   := c::m
    pos    := body(c::m)
    values := ∅
    InitLocals(c::m, vals)
```

The following definition for the initialization of local variables reflects that C♯ permits to pass function call parameters by value, as Java does, but also by `ref`erence. Also `out` parameters are allowed, treated as `ref` parameters except that they need not be definitely assigned until the function call returns. [16]

In the following definition, all (also simultaneous) applications of the external function *new* during the computation of the ASM are supposed to provide pairwise different fresh elements from the underlying domain *Adr*. See [24] and [18, 2.4.4] for a justification of this assumption. See also the end of Sect. 4 where we provide an abstract specification of the needed memory allocation to addresses of references and objects of struct type and to their instance fields. $paramIndex(c::m, x)$ yields the index of the formal parameter x in the signature of $c::m$.

```
InitLocals(c::m, vals) ≡
  forall x ∈ LocalVars(c::m) do                  // addresses for local variables
    locals(x) := new(Adr, type(x))
  forall x ∈ ValueParams(c::m) do                // copy value arguments
    let adr = new(Adr, type(x)) in
      locals(x) := adr
      WriteMem(adr, type(x), vals(paramIndex(c::m, x)))
  forall x ∈ RefParams(c::m) ∪ OutParams(c::m) do  // ref, out arguments
    locals(x) := vals(paramIndex(c::m, x))
```

Statement execution rules. The rules for method return in the submachine ExecCsharpStm_C trigger an abruption upon returning from a method, resulting (via the propagation of this abruption to the beginning of the method body where it occurred) in the execution of the machine ExitMethod. The rule to YieldUp(*Norm*) does not capture falling off the method body, but yields up the result of the normal execution of the invocation of a method with void return type in an expression statement.

$\text{ExecCsharpStm}_C \equiv$ **match** $context(pos)$

[16] To reflect different parameter passing mechanisms as encountered in other programming languages, due to the modular character of our model essentially only the above submachine InitLocals would have to be appropriately modified.

$\texttt{return}\ exp\,;\ \ \rightarrow pos := exp$
$\texttt{return}\ \blacktriangleright val\,;\ \rightarrow \textsc{YieldUp}(Return(val))$
$\texttt{return};\ \ \rightarrow \textsc{Yield}(Return)$

$Return \rightarrow \textbf{if}\ pos = body(meth) \wedge \neg Empty(frames)\ \textbf{then}$
$\quad \textsc{ExitMethod}(Norm)$
$Return(val) \rightarrow \textbf{if}\ pos = body(meth) \wedge \neg Empty(frames)\ \textbf{then}$
$\quad \textsc{ExitMethod}(val)$
$\blacktriangleright Norm\,; \rightarrow \textsc{YieldUp}(Norm)$

The machine ExitMethod restores the frame of the invoker and passes the result value (if any). Upon normal return from a static constructor it also updates the *typeState* of the relevant class as *Initialized*. We also add a rule FreeLocals to free the memory used for local variables and value parameters, using an abstract notion FreeMemory of how addresses of local variables and value parameters are actually de-allocated. [17]

$\textsc{ExitMethod}(result) \equiv$
  let $(oldMeth, oldPos, oldLocals, oldValues) = top(frames)$ **in**
    $meth := oldMeth$
    $pos := oldPos$
    $locals := oldLocals$
    $frames := pop(frames)$
    if $StaticCtor(meth) \wedge result = Norm$ **then**
      $typeState(type(meth)) := Initialized$
      $values := oldValues$
    else
      $values := oldValues \oplus \{oldPos \mapsto result\}$
    FreeLocals

FreeLocals $\equiv$
  forall $x \in LocalVars(meth) \cup ValueParams(meth)$ **do**
    $\textsc{FreeMemory}(locals(x), type(x))$

Following [2, §17.11,17.4.5.1,10.11,10.4.5.1] a type c is considered as initialized if its static constructor has been invoked (see the update of $typeState(c)$ to *InProgress* in Initialize below) or has terminated normally (see the update of $typeState(c)$ to *Initialized* in ExitMethod above). We therefore define:

$$Initialized(c) \iff typeState(c) = Initialized \vee typeState(c) = InProgress$$

To initialize a class its static constructor is invoked (`.cctor` = class constructor). This macro will be further refined in $C\sharp_{\mathcal{E}}$ to account for exceptions during

[17] Under the assumption of a potentially infinite supply of addresses, which is often made when describing the semantics of a programming language, one can dispense with FreeLocals.

an initialization.

```
INITIALIZE(c) ≡
  if typeState(c) = Linked then
    typeState(c) := InProgress
    forall f ∈ staticFields(c) do
      let t = type(c::f) in WRITEMEM(globals(c::f), t, defaultValue(t))
    INVOKEMETHOD(c::.cctor, [])
```

Note that in C♯ the initialization of a class does not trigger the initialization of its direct base class, differing on this point from Java where the rule for calling static constructors (see [1, Fig.4.5]) triggers the initialization of the superclass in case the superclass is not yet initialized.

With respect to the execution of initializers of static class fields the ECMA standard [2, §17.4.5.1] says that the static field initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an *implementation-dependent* time prior to the first use of a static field of that class. Our model expresses the decision taken by Microsoft's current C♯ compiler, which in the second case creates a static constructor. If one wants to reflect also the non-determinism suggested by the ECMA formulation, one can formalize the implementation-dependent external control by a monitored function *typeToBeInitialized* (which by the way can also be used for the classes and structs classified by an implementation flag as *beforefieldinit* type). The C♯ interpreter then takes the following form: [18]

```
if typeToBeInitialized ≠ Undef then
  INITIALIZE(typeToBeInitialized)
else EXECCSHARP
```

4 Refinement $C\sharp_{\mathcal{O}}$ of $C\sharp_{\mathcal{C}}$ by object related features

In this section we refine the static class features of $C\sharp_{\mathcal{C}}$ by adding objects (for class instances, comprising arrays and structs) together with *instance* fields, methods and constructors [19] as well as inheritance (including overriding

[18] This is discussed in detail in [13]. The reader finds there also a detection of further class initialization features that are missing in the ECMA specification, related to the definition of when a static class constructor has to be executed and to the initialization of structs.

[19] Destructors or finalizers which relate to garbage collection are not modeled here.

and overloading of methods). Accordingly we extend the ASM universes and functions of $C\sharp_{\mathcal{C}}$ to reflect the new expressions and statements together with the appropriate constraints and new rules, using appropriate refinements of some of the macros to define the semantics of the new instructions of $C\sharp_{\mathcal{O}}$. For the detailed definition of the syntax of (members of) classes, interfaces, structs, etc., and of the constraints for the new modifiers (`abstract`, `sealed`, `readonly`, `volatile`, `virtual`, `override`) together with illustrating examples, we refer the reader to [19].

4.1 Static semantics of $C\sharp_{\mathcal{O}}$

The first extension concerns the sets *Exp*, *Vexp*, *Sexp* where the new reference and array types appear. *Rank* serves to denote the dimension of array types; *NonArrayType* stands for value types, classes and interfaces and will be extended in $C\sharp_{\mathcal{D}}$ to comprise also delegates. Value types represent a feature that distinguishes C♯ from Java. In $C\sharp_{\mathcal{I}}$ we have cast expressions `(`*t*`)` *exp* where the type *t* and the type of *exp* are numeric types. Here, we extend the grammar to `(`*t*`)` *exp* where *t* and the type of *exp* can be any type. A *RefExp* is an expression of a reference type and an *ArrayExp* is an expression of an array type.

Exp ::= ... | '`null`' | '`this`' | '`typeof`' '`(`' *RetType* '`)`' | *Exp* '`is`' *Type*
| *Exp* '`as`' *RefType* | '`(`' *Type* '`)`' *Exp* | *Exp* '`.`' *Field*
| '`new`' *NonArrayType* '`[`' *Exps* '`]`' {*Rank*} [*ArrayInitializer*]

Vexp ::= ... | *Vexp* '`.`' *Field* | *RefExp* '`.`' *Field* | '`base`' '`.`' *Field*
| *ArrayExp* '`[`' *Exps* '`]`'

Sexp ::= ... | '`new`' *Type* `(` [*Args*] `)` | *Exp* '`.`' *Meth* `(` [*Args*] `)`
| '`base`' '`.`' *Meth* `(` [*Args*] `)`

Exps ::= *Exp* {'`,`' *Exp*}

Rank ::= '`[`' { '`,`' } '`]`'

A `this` in an instance constructor or instance method of a struct is considered to be a *Vexp*. When a `this` occurs in a class it is not a *Vexp*.

The extended type classification where simple types become aliases for struct types is re-assumed by Fig. 4. We refer the reader to [19] for the detailed list of new type constraints. Also the constraints for overriding and overloading of methods and the resolution of overloaded methods at compile-time are spelt out there.

The subtype relation (i.e. the standard implicit conversion) is based on the inheritance relation – defined as a finite tree with root `object` – together with the "implements" relation between classes and interfaces. It is defined as

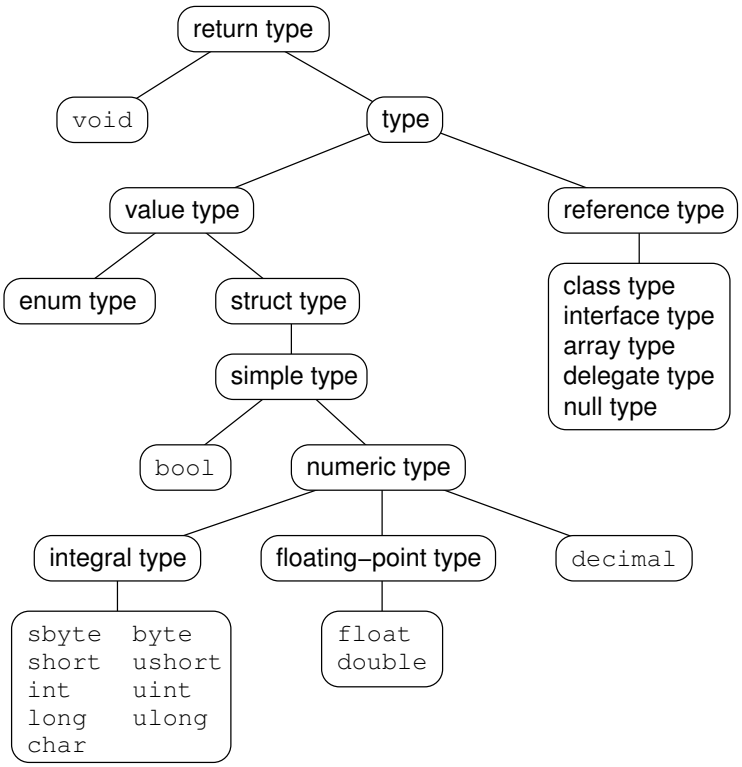


Fig. 4. The classification of types of C♯.

follows (and should not be confused with the classification of types in Fig. 4):

- T any type $\Longrightarrow T \preceq$ `object` and $T \preceq T$
- class S derived from $T \Longrightarrow S \preceq T$
- class, interface or struct S implements interface $T \Longrightarrow S \preceq T$
- T array type $\Longrightarrow T \preceq$ `System.Array`
- T delegate type $\Longrightarrow T \preceq$ `System.Delegate`
- T value type $\Longrightarrow T \preceq$ `System.ValueType`
- T array or delegate type $\Longrightarrow T \preceq$ `System.ICloneable`
- T reference type $\Longrightarrow \Lambda \preceq T$ [Λ is the null type]
- S and T reference types, $S \preceq T \Longrightarrow S\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]} \preceq T\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]}$

Note that types of one category in Fig. 4 can be subtypes of another (disjoint) category. For example, if a struct type S implements the interface I, then (the value type) S is considered to be a subtype of (the reference type) I.

We list here the additional definite assignment rules for local variables of struct type:

- If p is a local variable of a struct type S, then $p.f$ is considered as a local variable for each instance field f of S.
- A local variable p of struct type S is definitely assigned $\Longleftrightarrow$ $p.f$ is definitely assigned for each instance field f of S.

We assume that as a result of field and method resolution the attributed syntax tree has exact information. Each field access has the form T::f where f is a field declared in the type T. Each method call has the form T::m`(`*args*`)` where m is the signature of a method declared in type T. Moreover, at compile-time certain expressions are reduced to basic expressions as follows.

For the base access of fields and methods we have:

- `base.`f in class C is replaced by `this.`B::f, where B is the first base class of C where the field f is declared.
- `base.`m`(`*args*`)` in class C is replaced by `this.`B::M`(`*args*`)`, where B::M is the method signature of the method selected by the compiler (the set of applicable methods is constructed starting in the *base class* of C). This selection algorithm is described in [19], formalizing the conditions stated in [2, §14.4.2/3].

For instance field access and class instance creation we have:

- If f is a field, then f is replaced by `this.`T::f, where f is declared in T.
- Let T be a class type. Then the instance creation `new` T::M`(`*args*`)` is replaced by `new` T`.`T::M`(`*args*`)`.

Hence we split an instance creation expression into a creation part and an invocation of an instance constructor. To make the splitting correctly reflect the intended meaning of `new` T::M`(`*args*`)`, we assume in our model that class instance constructors return the value of `this`.

For instance constructors of structs one has to reflect that in addition they need an address for `this`. Also for constructors of structs we assume that they return the value of `this`. Let S be a struct type. Then the following transformations are applied:

- An assignment *vexp* `=` `new` S::M`(`*args*`)` is replaced by *vexp*`.`S::M`(`*args*`)`. This reflects that such a `new` triggers no object creation or memory allocation since structs get their memory allocated at declaration time.
- Other occurrences of `new` S::M`(`*args*`)` are replaced by x`.`S::M`(`*args*`)`, where x is a new temporary local variable of type S.

For automatic boxing we have:

- An assignment *vexp* `=` *exp* is replaced by *vexp* `=` `(`T`)`*exp* if *type*(*exp*) is a value type, T = *type*(*vexp*) and T is a reference type. In this case we must have *type*(*exp*) $\preceq T$.
- An argument *arg* is replaced by `(`T`)`*arg* if *type*(*arg*) is a value type, the selected method expects an argument of type T and T is a reference type. In this case we must have *type*(*arg*) $\preceq T$.

We are now ready to describe the extension of the dynamic state for the model of C♯$_\mathcal{O}$. The domain of values is extended to contain also references (assuming $Ref \cap Adr = \emptyset$ and $null \in Ref$) and struct values: $Value = SimpleValue \cup Adr \cup Ref \cup Struct$. The set $Struct$ of struct values can be defined as the set of mappings from $StructType::Field$ to $Value$. The value of an instance field of a value of struct type T can then be extracted by applying the map to the field name, i.e. $structField(val, T, f)$.

Two dynamic functions keep track of the $runTimeType\colon Ref \to Type$ of references and of the type object $typeObj\colon RetType \to Ref$ of a given type, where $RetType ::= Type \mid$ '`void`'. The memory function is extended to store also references: $mem\colon Adr \to SimpleValue \cup Ref \cup \{Undef\}$. For boxing we need a dynamic function $valueAdr\colon Ref \to Adr$ to record the address of a value in a box. If $runTimeType(ref)$ is a *value type* t, then $valueAdr(ref)$ is the address of the struct value of type t stored in the box. The static function $instanceFields\colon Type \to Powerset(Type::Field)$ yields the set of instance fields of any given type t; if t is a class type, it includes the fields declared in base classes of t. We use the common programming notation $Type::Field$ instead of the set-theoretic product set notation. We abstract from the implementation-dependent layout of structs and objects and use a function $fieldAdr\colon (Adr \cup Ref) \times Type::Field \to Adr$ to record addresses of fields. This function satisfies the following properties:

- If t is a *struct type*, then $fieldAdr(adr, t::f)$ is the address of field f of a value of type t stored in mem at address adr.
- A value of struct type t at address adr occupies the following addresses in mem:

  $$\{fieldAdr(adr, f) \mid f \in instanceFields(t)\}$$

- If $runTimeType(ref)$ is a *class type*, then $fieldAdr(ref, t::f)$ is the address of field $t::f$ of the object referenced by ref.
- An object of class c is represented by a reference ref with the property $runTimeType(ref) = c$ and occupies the following addresses in mem:

  $$\{fieldAdr(ref, f) \mid f \in instanceFields(c)\}$$

A function $elemAdr\colon Ref \times \mathbb{N}^* \to Adr$ records addresses of array elements. The `this` reference is treated as first parameter and is passed by value. Therefore $paramIndex(c::m, \texttt{this}) = 0$ and `this` is an element of $ValueParams(c::m)$.

For the refinement of the ExecCsharp$_C$ transition rules it suffices to add the new rule ExecCsharpExp$_O$ for evaluating the new expressions, since C♯$_\mathcal{O}$

introduces no new statements.

```
ExecCsharp_O ≡
  ExecCsharp_C
  ExecCsharpExp_O
```

For better readability we organize the numerous ExecCsharpExp$_O$ rules for each of the new expressions into parallel submachines each of which collects the rules for expressions which belong to the same category (for type testing and casting, for fields, for arrays). As analyzed in [13], the invocation of an instance constructor of a class may trigger the class initialization.

```
ExecCsharpExp_O ≡ match context(pos)
  null → Yield(null)
  this → YieldIndirect(locals(this))
  TestCastExp_O
  FieldExp_O
  new c → let ref = new(Ref, c) in
            runTimeType(ref) := c
            forall f ∈ instanceFields(c) do
              let adr = fieldAdr(ref, f) and t = type(f) in
                WriteMem(adr, t, defaultValue(t))
            Yield(ref)
  exp.T::M(args)   → pos := exp
  ▶val.T::M(args) → if StructValueInvocation(up(pos)) then
                      // create home for struct value
                      let adr = new(Adr, type(pos)) in
                        WriteMem(adr, type(pos), val)
                        values(pos) := adr
                    pos := (args)
  val.T::M▶(vals) → if InstanceCtor(M) ∧ TriggerInit(T) then
                      Initialize(T)
                    elseif val ≠ null then
                      InvokeInstance(T::M, val, vals)
  ArrayExp_O
```

A struct value invocation is a method invocation on a struct value which is not stored in a variable. For such struct values the above rule creates a temporary storage area (called "home") to be passed in the invocation as value of `this`.

```
StructValueInvocation(exp.T::M(args)) ⟺
  type(exp) ∈ StructType ∧ exp ∉ Vexp
```

The rules for casting in TestCastExp$_O$ use the new macro YieldUpBox defined below. Note that in expressions '*exp* `is` *t*' and `(`*t*`)`*exp* the type *t* can

be any type, whereas in '*exp* `as` t' the type t must be a reference type. The type of '*exp* `is` t' is `bool`, while the type of `(`t`)`*exp* and '*exp* `as` t' is t.

TestCastExp$_O$ ≡
  `typeof(`t`)` → Yield(*typeObj*(t))
  exp `is` t → $pos := exp$
  ▸*val* `is` t → **if** $type(pos) \in ValueType$ **then**
      YieldUp($type(pos) \preceq t$) // compile-time property
    else
      YieldUp($val \neq null \wedge runTimeType(val) \preceq t$)
  exp `as` t → $pos := exp$
  ▸*val* `as` t → **if** $type(pos) \in ValueType$ **then**
      YieldUpBox($type(pos), val$) // box a copy of the value
    elseif ($val \neq null \wedge runTimeType(val) \preceq t$) **then**
      YieldUp(val) // pass reference through
    else YieldUp($null$) // convert to null reference
  `(`t`)`*exp* → $pos := exp$
  `(`t`)`▸*val* → **if** $type(pos) \in ValueType$ **then**
      // compile-time identity
      if $t = type(pos)$ **then** YieldUp(val)
      // box value
      if $t \in RefType$ **then** YieldUpBox($type(pos), val$)
    if $type(pos) \in RefType$ **then**
      if $t \in RefType \wedge (val = null \vee runTimeType(val) \preceq t)$ **then**
        YieldUp(val) // pass reference through
      if $t \in ValueType \wedge val \neq null \wedge t = runTimeType(val)$ **then**
        // un-box a copy of the value
        YieldUp($memValue(valueAdr(val), t)$)

The rules for instance field access and assignment in FieldExp$_O$ are analogous to those for class variables, adding the evaluation of the instance, using *fieldAdr* instead of *globals*, and instead of WriteMem the macro SetField defined below. The second rule for instance field access has to distinguish two cases, depending upon the statically known instance type. Since this type information is already known at the time of static analysis, it could be resolved by introducing two separate constructs for field access, as one of our referees observed pointing also out that in fact the CLI has a single, overloaded instruction for field access with overloading to be resolved by the JIT. However from the modeling point of view, having two separate constructs for field access would lead to essentially the same two rules we have formulated here, except for having as rule guard a matching condition for the two constructs instead of the type test. We use $type(exp\,.\,t{::}f) = type(t{::}f)$.

FieldExp$_O$ ≡

$exp\,.\,t{::}f \;\to pos := exp$
$\blacktriangleright val\,.\,t{::}f \to$ **if** $type(pos) \in ValueType \wedge val \notin Adr$ **then**
$\textsc{YieldUp}(structField(val, type(pos), t{::}f))$
elseif $val \neq null$ **then**
$\textsc{YieldUpIndirect}(fieldAdr(val, t{::}f))$
$exp_1\,.\,t{::}f$ `=` $exp_2 \;\to pos := exp_1$
$\blacktriangleright val\,.\,t{::}f$ `=` $exp \;\to pos := exp$
$val_1\,.\,t{::}f$ `=` $\blacktriangleright val_2 \to$ **if** $val_1 \neq null$ **then**
$\textsc{SetField}(val_1, t{::}f, val_2)$
$\textsc{YieldUp}(val_2)$

C♯$_{\mathcal{O}}$ supports single dimensional as well as multi-dimensional arrays. Array types are read from right to left. For example, `int[][,]` is the type of single-dimensional arrays of two-dimensional arrays with elements of type `int`. By $dim(n)$ we denote a sequence of $n-1$ commas, hence T`[`$dim(n)$`]` is the type of n-dimensional arrays with elements of type T. The length of the ith dimension of an n-dimensional array represented by a reference ref is stored as the value of $dimLength(ref, i)$. Note that the rules for using array indexing expressions as rvalue respectively as lvalue appear together as subgroups of $\textsc{ArrayExp}_O$, separated by pattern matching.

$\textsc{ArrayExp}_O \equiv$
`new` T`[`exp_1`,` $\ldots$ `,` exp_n`]`$[R_1]\cdots[R_k] \to pos := exp_1$
`new` T`[`l_1`,` $\ldots$ `,`$\blacktriangleright l_i$`,` exp_{i+1}`,` $\ldots$ `,` exp_n`]`$[R_1]\cdots[R_k] \to pos := exp_{i+1}$
`new` T`[`l_1`,` $\ldots$ `,`$\blacktriangleright l_n$`]`$[R_1]\cdots[R_k] \to$
if $\forall i \in [1\,..\,n]\,(0 \leq l_i)$ **then**
let $S = T$`[`R_1`]`$\cdots$`[`R_k`]` **in**
let $ref = new(Ref, [l_1, \ldots, l_n], S)$ **in**
$runTimeType(ref) := T$`[`$dim(n)$`]`$[R_1]\cdots[R_k]$
forall $i \in [1\,..\,n]$ **do** $dimLength(ref, i-1) := l_i$
forall $\alpha \in [0\,..\,l_1-1] \times \cdots \times [0\,..\,l_n-1]$ **do**
$\textsc{WriteMem}(elemAdr(ref, \alpha), S, defaultValue(S))$
$\textsc{YieldUp}(ref)$
exp_0`[`exp_1`,` $\ldots$ `,` exp_n`]` $\to pos := exp_0$
$\blacktriangleright ref$`[`exp_1`,` $\ldots$ `,` exp_n`]` $\to pos := exp_1$
ref`[`i_1`,` $\ldots$ `,`$\blacktriangleright i_k$`,` exp_{k+1}`,` $\ldots$ `,` exp_n`]` $\to pos := exp_{k+1}$
ref`[`i_1`,` $\ldots$ `,`$\blacktriangleright i_n$`]` $\to$
if $ref \neq null \wedge \forall k \in [1\,..\,n]\,(0 \leq i_k < dimLength(ref, k-1)) \wedge$
$(RefOrOutArg(up(pos)) \wedge type(up(pos)) \in RefType \to$
$elementType(runTimeType(ref)) = type(up(pos)))$
then
$\textsc{YieldUpIndirect}(elemAdr(ref, (i_1, \ldots, i_n)))$
exp_0`[`exp_1`,` $\ldots$ `,` exp_n`]` `=` $exp \;\to pos := exp_0$
$\blacktriangleright ref$`[`exp_1`,` $\ldots$ `,` exp_n`]` `=` $exp \to pos := exp_1$

$ref\,\texttt{[}\,i_1\,,\,\ldots\,,\,\texttt{▶}\,i_k\,,\,exp_{k+1}\,,\,\ldots\,,\,exp_n\,\texttt{]}\,\texttt{=}\,exp \rightarrow pos := exp_{k+1}$
$ref\,\texttt{[}\,i_1\,,\,\ldots\,,\,\texttt{▶}\,i_n\,\texttt{]}\,\texttt{=}\,exp \rightarrow pos := exp$
$ref\,\texttt{[}\,i_1\,,\,\ldots\,,\,i_n\,\texttt{]}\,\texttt{=}\,\texttt{▶}\,val \rightarrow$
  let $T = elementType(runTimeType(ref))$ **in**
    if $ref \neq null \wedge \forall k \in [1\,..\,n]\,(0 \leq i_k < dimLength(ref, k-1)) \wedge$
      $(type(pos) \in RefType \rightarrow runTimeType(val) \preceq T)$
    then
      WriteMem($elemAdr(ref, (i_1, \ldots, i_n)), T, val$)
      YieldUp(val)

Refinement of macros. Invocation of instance methods splits into virtual and non-virtual calls. The function *lookup* yields the class where the given method specification is defined in the class hierarchy, depending on the run-time type of the given reference.

InvokeInstance($T::M, val, vals$) $\equiv$
  if $callKind(T::M) = Virtual$ **then** // indirect call, $val \in Ref$
    let $S = lookup(runTimeType(val), T::M)$ **in**
    let $this =$ **if** $S \in StructType$ **then** $valueAdr(val)$ **else** val **in**
      InvokeMethod($S::M, [this] \cdot vals$)
  if $callKind(T::M) = NonVirtual$ **then** // direct call, $val \in Adr \cup Ref$
    InvokeMethod($T::M, [val] \cdot vals$)

In $C\sharp_{\mathcal{O}}$ the notion of reading from the memory is refined by extending the simple equation $memValue(adr, t) = mem(adr)$ of $C\sharp_{\mathcal{I}}$ to fit also reference and struct types. This is done by the following simultaneous recursive definition of *memValue* and *getField* along the given struct type.

$memValue(adr, t) =$
  if $t \in SimpleType \cup RefType$ **then** $mem(adr)$
  elseif $t \in StructType$ **then** $\{f \mapsto getField(adr, f) \mid f \in instanceFields(t)\}$

$getField(adr, t::f) = memValue(fieldAdr(adr, t::f), type(t::f))$

Writing to memory is refined recursively together with SetField along the given struct type:

WriteMem(adr, t, val) $\equiv$
  if $t \in SimpleType \cup RefType$ **then** $mem(adr) := val$
  elseif $t \in StructType$ **then**
    forall $f \in instanceFields(t)$ **do** SetField($adr, f, val(f)$)

SetField($adr, t::f, val$) $\equiv$ WriteMem($fieldAdr(adr, t::f), type(t::f), val$)

The notion of *AddressPos* from $C\sharp_{\mathcal{C}}$ is refined to include also lvalue nodes of *StructType*, so that address positions are of the following form: `ref` □, `out` □,

$\square$`++`, $\square$`--`, $\square$ *op*`=` *exp*, $\square$`.`f, $\square$`.`m`(`*args*`)`.

$$\begin{array}{l} \mathit{AddressPos}(\alpha) \iff \mathit{FirstChild}(\alpha) \wedge \\ \quad \mathit{label}(\mathit{up}(\alpha)) \in \{\texttt{ref}, \texttt{out}, \texttt{++}, \texttt{--}\} \vee \mathit{label}(\mathit{up}(\alpha)) \in \mathit{Aop} \vee \\ \quad (\mathit{label}(\mathit{up}(\alpha)) = \text{'.'} \wedge \alpha \in \mathit{Vexp} \wedge \mathit{type}(\alpha) \in \mathit{StructType}) \end{array}$$

YieldUpBox creates a box for a given value of a given type and returns its reference. The run-time type of a reference to a boxed value of struct type t is defined to be (the value type) t of the value. There is no need to introduce special reference types for boxed values. If $\mathit{type}(\mathit{exp})$ is a value type that implements the interface I, then $\mathit{type}(\mathit{exp}) \preceq I$ and the value can be boxed using '`(`I`)` *exp*' or '*exp* `as` i'.

$$\begin{array}{l} \textsc{YieldUpBox}(t, \mathit{val}) \equiv \textbf{let } \mathit{ref} = \mathit{new}(\mathit{Ref}) \textbf{ and } \mathit{adr} = \mathit{new}(\mathit{Adr}, t) \textbf{ in} \\ \quad \mathit{runTimeType}(\mathit{ref}) := t \\ \quad \mathit{valueAdr}(\mathit{ref}) := \mathit{adr} \\ \quad \textsc{WriteMem}(\mathit{adr}, t, \mathit{val}) \\ \quad \textsc{YieldUp}(\mathit{ref}) \end{array}$$

The struct value is copied in both cases, when it is boxed and when it is un-boxed.

ASM function new. We now justify in the context of the basic parallel execution mechanism of ASM rules the sequentiality which is used in the following macros:

$$\begin{array}{l} \textbf{let } \mathit{adr} = \mathit{new}(\mathit{Adr}, T) \textbf{ in } P \\ \textbf{let } \mathit{ref} = \mathit{new}(\mathit{Ref}, T) \textbf{ in } P \\ \textbf{let } \mathit{ref} = \mathit{new}(\mathit{Ref}, [l_1, \ldots, l_n], T) \textbf{ in } P \end{array}$$

In the context of the machine ExecCsharp this comes up to specify an abstract memory management. In fact **let** $\mathit{adr} = \mathit{new}(\mathit{Adr}, T)$ **in** P stands for the sequential execution of a new address allocation (which uses the ASM construct **import** to provide a completely fresh element) followed by P:

$$\textbf{let } \mathit{adr} = \mathit{new}(\mathit{Adr}, T) \textbf{ in } P \equiv (\textbf{import } \mathit{adr} \textbf{ do } \textsc{AllocAdr}(\mathit{adr}, T)) \textbf{ seq } P$$

where the operator **seq** for sequential execution of two ASMs M, N is to be understood as defined for turbo ASMs in [25] (alternatively see [18, Ch.4.1]), namely as binding into one overall ASM step the two steps of first executing M in the given state and then N in the resulting state. Similarly **let** $\mathit{ref} = \mathit{new}(\mathit{Ref}, T)$ **in** P stands for the sequential execution of address allocation for all instance fields of a given type followed by P:

$$\begin{array}{l} \textbf{let } \mathit{ref} = \mathit{new}(\mathit{Ref}, T) \textbf{ in } P \equiv \\ \quad \textbf{import } \mathit{ref} \textbf{ do} \end{array}$$

```
    Ref(ref) := True
    AllocFields(ref, instanceFields(T))
  seq P
```

Similarly we define the address allocation for elements of an n-dimensional array:

```
let ref = new(Ref, [l_1, ..., l_n], T) in P ≡
  import ref do
    Ref(ref) := True
    forall α ∈ [0 .. l_1 − 1] × ··· × [0 .. l_n − 1] do
      import adr do
        elemAdr(ref, α) := adr
        AllocAdr(adr, T)
  seq P
```

The two macros for allocation of addresses and fields can be recursively defined as follows, relying again upon the definition of recursive turbo ASMs in [22] (or see alternatively [18, Ch.4.1.2]):

```
AllocAdr(adr, T) ≡
  Adr(adr) := True
  if T ∈ StructType then AllocFields(adr, instanceFields(T))

AllocFields(x, fs) ≡
  forall f ∈ fs import adr do
    fieldAdr(x, f) := adr
    AllocAdr(adr, type(f))
```

5 Refinement C♯$_\mathcal{E}$ of C♯$_\mathcal{O}$ by exception handling

In this section we extend C♯$_\mathcal{O}$ with the exception handling mechanism of C♯, which separates normal program code from exception handling code. To this purpose exceptions are represented as objects of predefined system exception classes or of user-defined application exception classes. Once created (thrown), these objects trigger an abruption of the normal program execution to catch the exception – in case it is compatible with one of the exception classes appearing in the program in an enclosing try-catch-finally statement. Optional finally statements are guaranteed to be executed independently of whether the try statement completes normally or is abrupted.

For the refinement of ExecCsharp$_O$ by exceptions, as in the previous section it suffices to add the new rules for exception handling and to extend the static semantics. The set of statements is extended by throw and try-catch-finally statements satisfying the following constraints:

$$\begin{aligned} \mathit{Stm} \quad &::= \ldots \mid \text{`}\texttt{throw}\text{'}\ \mathit{Exp}\ \text{`}\texttt{;}\text{'} \mid \text{`}\texttt{throw}\text{'}\ \text{`}\texttt{;}\text{'} \\ &\quad \mid \text{`}\texttt{try}\text{'}\ \mathit{Block}\ \{\mathit{Catch}\}\ [\text{`}\texttt{catch}\text{'}\ \mathit{Block}]\ [\text{`}\texttt{finally}\text{'}\ \mathit{Block}] \\ \mathit{Catch} \ &::= \text{`}\texttt{catch}\text{'}\ \text{`}\texttt{(}\text{'}\ \mathit{ClassType}\ [\mathit{Loc}]\ \text{`}\texttt{)}\text{'}\ \mathit{Block} \end{aligned}$$

- every try-catch-finally statement contains at least one *catch clause*, *general catch clause* (i.e. of form `catch` *block*), or *finally block*
- no `return` statements are allowed in finally blocks
- a `break`, `continue`, or `goto` statement is not allowed to jump out of a finally block
- a `throw` statement without expression is only allowed in catch blocks
- the exception classes in a *Catch* clause appear there in a non-decreasing type order, more precisely, for every try-catch statement of the form

  `try` *block* `catch (`E_1 x_1`)` $block_1 \ldots$ `catch (`E_n x_n`)` $block_n$

  the following holds $i < j \Longrightarrow E_j \not\preceq E_i$ (and $E_i \preceq$ `System.Exception`).

In our model the sets of abruptions and type states have to be extended by exceptions. Due to the presence of `throw` statements without expression, a stack of references is needed to record exceptions which are to be re-thrown.

$$\mathit{Abr} = \ldots \mid \mathit{Exc}(\mathit{Ref}),\ \mathit{TypeState} = \ldots \mid \mathit{Exc}(\mathit{Ref}),\ \mathit{excStack}\colon \mathit{List}(\mathit{Ref})$$

To simplify the exposition we assume that general catch clauses '`catch` *block*' are replaced at compile-time by '`catch (Object` x `)` *block*' with a new variable x. We also reduce try-catch-finally statements to try-catch and try-finally statements as follows. Both reductions can easily be shown to correctly express the ECMA specification.

```
try TryBlock
catch (E1 x1) CatchBlock1
  ⋮                                  ⟹
catch (En xn) CatchBlockn
finally FinallyBlock
```

```
try {
  try TryBlock
  catch (E1 x1) CatchBlock1
    ⋮
  catch (En xn) CatchBlockn
} finally FinallyBlock
```

If a static constructor throws an exception, and no catch clauses exists to catch it, then this exception is wrapped into a `TypeInitializationException` by translating `static` T`()` `{` *BlockStatements* `}` into

```
static T() {
  try { BlockStatements }
  catch (Exception e) {
    throw new TypeInitializationException(T,e);
  }
}
```

The reachability rules and the definite assignment constraints for a try-catch $stm \equiv$ `try` *tryBlock* `catch (`$E_1\ x_1$`)` $catchBlock_1 \ldots$ `catch (`$E_n\ x_n$`)` $catchBlock_n$ are:

- If $reachable(stm)$, then $reachable(tryBlock)$ and $reachable(catchBlock_i)$ for every $i \in [1\,..\,n]$.
- If $normal(tryBlock)$ or $normal(catchBlock)$ for at least one $i \in [1\,..\,n]$, then $normal(stm)$.

- $before(tryBlock) = before(stm)$
- $before(catchBlock_i) = before(stm) \cup \{x_i\}$ for every $i \in [1\,..\,n]$
- $after(stm) = after(tryBlock) \cap \bigcap_{i=1}^{n} after(catchBlock_i)$

For a statement *stm* of the form `try` *tryBlock* `finally` *finallyBlock* the rules and constraints are:

- If $reachable(stm)$, then $reachable(tryBlock)$ and $reachable(finallyBlock)$.
- If $normal(tryBlock)$ and $normal(finallyBlock)$, then $normal(stm)$.

- $before(tryBlock) = before(stm)$
- $before(finallyBlock) = before(stm)$
- $after(stm) = after(tryBlock) \cup after(finallyBlock)$

5.2 Dynamic semantics for $\mathrm{C}\sharp_{\mathcal{E}}$

The transition rules for ExecCsharp$_E$ are defined by adding two submachines to ExecCsharp$_O$. The first one provides the rules for handling the exceptions which may occur during the evaluation of expressions, the second one describes the meaning of the new throw and try-catch-finally statements.

ExecCsharp$_E \equiv$
  ExecCsharp$_O$
  ExecCsharpExp$_E$

ExecCsharpStm$_E$

Expression evaluation rules. ExecCsharpExp$_E$ contains rules for each of the numerous forms of run-time exceptions defined in the subclasses of `System.Exception`. We give here seven characteristic examples and group them for the ease of presentation into parallel submachines by the form of expression they are related to, namely for arithmetical exceptions and for those related to cast expressions, reference expressions or array expressions. The notion of FailUp we use is supposed to execute the code `throw new` E`()` at the parent position, so that we define the macro by invoking an internal method `Throw`E with that effect for each of the exception classes E used as parameter of FailUp.

ExecCsharpExp$_E$ ≡ **match** *context*(*pos*)
  uop ▸ val → **if** *Checked*(*pos*) ∧ *Overflow*(uop, val) **then**
    FailUp(`OverflowException`)
  val_1 bop ▸ val_2 → **if** *DivisionByZero*(bop, val_2) **then**
    FailUp(`DivideByZeroException`)
  elseif *DecimalOverflow*(bop, val_1, val_2)∨
    (*Checked*(*pos*) ∧ *Overflow*(bop, val_1, val_2))
  then FailUp(`OverflowException`)
  CastExceptions
  NullRefExceptions
  ArrayExceptions

FailUp(E) ≡ InvokeMethod(*ExcSupport*::`Throw`E, [ ])

CastExceptions ≡ **match** *context*(*pos*)
  (*t*)▸ val →
    if *type*(*pos*) ∈ *RefType* **then**
      if t ∈ *RefType* ∧ val ≠ *Null* ∧ *runTimeType*(val) ⋡ t **then**
        FailUp(`InvalidCastException`)
      if t ∈ *ValueType* **then** // attempt to unbox
        if val = *Null* **then** FailUp(`NullReferenceException`)
        elseif t ≠ *runTimeType*(val) **then**
          FailUp(`InvalidCastException`)
      if *type*(*pos*) ∈ *NumericType* ∧ t ∈ *NumericType* ∧ *Checked*(*pos*) ∧
        Overflow(t, val)
      then FailUp(`OverflowException`)

The semantics of assignments as defined by the ECMA standard and formalized by the rule NullRefExceptions is violated by a compiler optimization in [7] related to the timing of the *Null* check for certain expressions, see the

analysis in [13].

$\textsc{NullRefExceptions} \equiv \textbf{match}\ \mathit{context}(\mathit{pos})$
$\quad {}^{\blacktriangleright}\mathit{ref}\,.\,t{::}f \rightarrow \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \textsc{FailUp}(\texttt{NullReferenceException})$
$\quad \mathit{ref}\,.\,t{::}f\ \texttt{=}\ {}^{\blacktriangleright}\mathit{val} \rightarrow \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \textsc{FailUp}(\texttt{NullReferenceException})$
$\quad \mathit{ref}\,.\,T{::}M\,\texttt{(}{}^{\blacktriangleright}\mathit{vals}\texttt{)} \rightarrow \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \textsc{FailUp}(\texttt{NullReferenceException})$

If the address of an array element is passed as a `ref` or `out` argument to a method, then the run-time element type of the array must be *equal* to the parameter type that the method expects. If an object is assigned to an array element, then the type of the object must be a *subtype* of run-time element type of the array (array covariance problem). In both cases, if the condition is not satisfied, an `ArrayTypeMismatchException` is thrown.

$\textsc{ArrayExceptions} \equiv \textbf{match}\ \mathit{context}(\mathit{pos})$
$\quad \texttt{new}\ T\texttt{[}l_1\texttt{, \ldots ,}{}^{\blacktriangleright}l_n\texttt{]}\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]} \rightarrow$
$\qquad \textbf{if}\ \exists i \in [1\,..\,n]\,(l_i < 0)\ \textbf{then}\ \textsc{FailUp}(\texttt{OverflowException})$
$\quad \mathit{ref}\,\texttt{[}i_1\texttt{, \ldots ,}{}^{\blacktriangleright}i_n\texttt{]} \rightarrow$
$\qquad \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \textsc{FailUp}(\texttt{NullReferenceException})$
$\qquad \textbf{elseif}\ \exists k \in [1\,..\,n]\,(i_k < 0 \vee \mathit{dimLength}(\mathit{ref}, k-1) \leq i_k)\ \textbf{then}$
$\qquad\quad \textsc{FailUp}(\texttt{IndexOutOfRangeException})$
$\qquad \textbf{elseif}\ \mathit{RefOrOutArg}(\mathit{up}(\mathit{pos})) \wedge \mathit{type}(\mathit{up}(\mathit{pos})) \in \mathit{RefType} \wedge$
$\qquad\quad \mathit{elementType}(\mathit{runTimeType}(\mathit{ref})) \neq \mathit{type}(\mathit{up}(\mathit{pos}))$
$\qquad \textbf{then}\ \textsc{FailUp}(\texttt{ArrayTypeMismatchException})$
$\quad \mathit{ref}\,\texttt{[}i_1\texttt{, \ldots ,}i_n\texttt{]}\ \texttt{=}\ {}^{\blacktriangleright}\mathit{val} \rightarrow$
$\qquad \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \textsc{FailUp}(\texttt{NullReferenceException})$
$\qquad \textbf{elseif}\ \exists k \in [1\,..\,n]\,(i_k < 0 \vee \mathit{dimLength}(\mathit{ref}, k-1) \leq i_k)\ \textbf{then}$
$\qquad\quad \textsc{FailUp}(\texttt{IndexOutOfRangeException})$
$\qquad \textbf{elseif}\ \mathit{type}(\mathit{pos}) \in \mathit{RefType} \wedge$
$\qquad \mathit{runTimeType}(\mathit{val}) \not\leq \mathit{elementType}(\mathit{runTimeType}(\mathit{ref}))\ \textbf{then}$
$\qquad\quad \textsc{FailUp}(\texttt{ArrayTypeMismatchException})$

Statement execution rules. The statement execution submachine splits naturally into submachines for throw, try-catch, try-finally statements and a rule for the propagation of an exception (from the root position of a method body) to the method caller. To support a correct understanding of the exception messages that are printed to the console we include into the rule for throw statements the initialization of stack traces. The initialization of stack traces in Java and C♯ is different. In Java, the stack trace is initialized with the complete trace up to the main function once and for all when the exception object is created. In C♯ the stack trace is initialized with the empty trace each time when the exception object is thrown with `throw` and then augmented whenever the exception is propagated to a parent frame. The semantics of

the parameterless `throw;` instruction is explained in terms of the exception Stack *excStack*. When an exception is caught, it is pushed on top of the exception stack (which as explained above is needed to record exceptions which are to be re-thrown). Whenever a catch block terminates (normally or abruptly) the topmost element of the exception stack is deleted. No special rules are needed for general catch clauses '`catch` *block*' in try-catch statements, due to their compile-time transformation mentioned above. The completeness of the try-finally rules is due to the constraints listed above, which restrict the possibilities for exiting a finally block to normal completion or triggering an exception.

$$
\begin{array}{l}
\textsc{ExecCsharpStm}_E \equiv \textbf{match}\ \mathit{context}(\mathit{pos}) \\
\quad \texttt{throw}\ \mathit{exp}\,; \ \ \to \mathit{pos} := \mathit{exp} \\
\quad \texttt{throw}\ \blacktriangleright \mathit{ref}\,; \to \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \textsc{FailUp}(\texttt{NullReferenceException}) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \textsc{InitStackTrace}(\mathit{ref}, \mathit{meth}) \\
\qquad\qquad\qquad\qquad\quad \textsc{YieldUp}(\mathit{Exc}(\mathit{ref})) \\
\quad \texttt{throw;} \to \textsc{Yield}(\mathit{Exc}(\mathit{top}(\mathit{excStack}))) \\
\quad \texttt{try}\ \mathit{block}\ \texttt{catch}\ (E\ x)\ \mathit{stm}\ \ldots \quad \to \mathit{pos} := \mathit{block} \\
\quad \texttt{try}\ \blacktriangleright \mathit{Norm}\ \texttt{catch}\ (E\ x)\ \mathit{stm}\ \ldots \to \textsc{YieldUp}(\mathit{Norm}) \\
\quad \texttt{try}\ \blacktriangleright \mathit{Exc}(\mathit{ref})\ \texttt{catch}(E_1\ x_1)\ \mathit{stm}_1\ \ldots\ \texttt{catch}(E_n\ x_n)\ \mathit{stm}_n \to \\
\qquad \textbf{if}\ \exists i \in [1\,..\,n]\ \mathit{runTimeType}(\mathit{ref}) \preceq E_i\ \textbf{then} \\
\qquad\quad \textbf{let}\ j = \min\{i \in [1\,..\,n] \mid \mathit{runTimeType}(\mathit{ref}) \preceq E_i\}\ \textbf{in} \\
\qquad\qquad \mathit{pos} := \mathit{stm}_j \\
\qquad\qquad \mathit{excStack} := \mathit{push}(\mathit{ref}, \mathit{excStack}) \\
\qquad\qquad \textsc{WriteMem}(\mathit{locals}(x_j), \texttt{object}, \mathit{ref}) \\
\qquad \textbf{else}\ \textsc{YieldUp}(\mathit{Exc}(\mathit{ref})) \\
\quad \texttt{try}\ \blacktriangleright \mathit{abr}\ \texttt{catch}(E_1\ x_1)\ \mathit{stm}_1\ \ldots\ \texttt{catch}(E_n\ x_n)\ \mathit{stm}_n \ \to \textsc{YieldUp}(\mathit{abr}) \\
\quad \texttt{try}\ \mathit{Exc}(\mathit{ref})\ \ldots\ \texttt{catch}(\ldots)\ \blacktriangleright \mathit{res}\ \ldots \to \\
\qquad \{\mathit{excStack} := \mathit{pop}(\mathit{excStack}),\ \textsc{YieldUp}(\mathit{res})\} \\
\quad \texttt{try}\ \mathit{tryBlock}\ \texttt{finally}\ \mathit{finallyBlock} \to \mathit{pos} := \mathit{tryBlock} \\
\quad \texttt{try}\ \blacktriangleright \mathit{res}\ \texttt{finally}\ \mathit{finallyBlock} \quad \to \mathit{pos} := \mathit{finallyBlock} \\
\quad \texttt{try}\ \mathit{res}\ \texttt{finally}\ \blacktriangleright \mathit{Norm} \quad \to \textsc{YieldUp}(\mathit{res}) \\
\quad \texttt{try}\ \mathit{res}\ \texttt{finally}\ \blacktriangleright \mathit{Exc}(\mathit{ref}) \quad \to \textsc{YieldUp}(\mathit{Exc}(\mathit{ref})) \\
\quad \mathit{Exc}(\mathit{ref}) \to \textbf{if}\ \mathit{pos} = \mathit{body}(\mathit{meth}) \wedge \neg \mathit{Empty}(\mathit{frames})\ \textbf{then} \\
\qquad \textbf{if}\ \mathit{StaticCtor}(\mathit{meth})\ \textbf{then}\ \mathit{typeState}(\mathit{type}(\mathit{meth})) := \mathit{Exc}(\mathit{ref}) \\
\qquad \textbf{else}\ \textsc{AppendStackTrace}(\mathit{ref}, \mathit{meth}(\mathit{top}(\mathit{frames}))) \\
\qquad \textsc{ExitMethod}(\mathit{Exc}(\mathit{ref}))
\end{array}
$$

In case an exception happened in the static constructor of a type, its type state is set to that exception to prevent its re-initialization and instead to re-throw the old exception object. The refinement of the macro Initialize defined in $\mathrm{C}\sharp_{\mathcal{C}}$ re-throws the exception object of a type which had an exception in the

static constructor, thus preventing its re-initialization. [20]

INITIALIZE(c) $\equiv$
  ...
  if *typeState*(c) = *Exc*(*ref*) **then** YIELD(*Exc*(*ref*))

6 Refinement C♯$_{\mathcal{D}}$ of C♯$_{\mathcal{E}}$ by delegates

In this section we extend C♯$_{\mathcal{E}}$ by features which distinguish C♯ from other languages, e.g. Java. We start with delegates and then add further constructs whose semantics can be defined mainly by reducing them via syntactical translations to the language model developed so far: properties, indexers, overloaded operators, enumerators with the `foreach` statement, the `using` statement, events and attributes. We use the model developed so far as ground model (in the sense of [12]) for C♯, thus providing a basis to justify the correctness (with respect to the ECMA standard [2]) of the "semantics of syntactic sugar" introduced in this section to define the semantics for delegates, properties, etc.

6.1 Delegates

Delegate types in C♯ are reference types that encapsulate a static or instance method with a specific signature, with the intention of having delegates playing the role of type-safe function pointers. A delegate type D is declared as follows:

`delegate` T D`(`S_1 x_1`, ...,`S_n x_n`);`

It represents the type of methods that take n arguments of type $S_1, \ldots, S_n$ and have return type T. Delegate types appear as subtypes of `System.Delegate` and provide in particular the *callback* functionality and asynchronous event handling. More precisely, the characteristic ability of delegates is to call a list of multiple methods sequentially. This feature is realized by means of an *invocationList*: *Ref* $\to$ *Delegate** $\cup \{$*Undef*$\}$ with which each delegate instance is equipped upon its creation. Each such list is a per instance immutable, non-empty, ordered list of static methods or pairs of target objects and instance methods. Upon invocation of a delegate instance with arguments *args*, the

[20] For modeling the implementation flag *beforefieldinit* mentioned above this implies, as observed in [13], to refine also the predicate *TriggerInit*, used for invoking static or instance methods, namely to guarantee for a class in exception state its initialization even if the class is marked *beforefieldinit*: *TriggerInit*(c) = ($\neg$*Initialized*(c) $\wedge$ $\neg$*beforefieldinit*(c)) $\vee$ *typeState*(c) = *Exc*(*ref*).

methods of its invocation list are called one after the other with these arguments *args*, returning to the caller of the delegate either the *return value* of the last list element or the first *exception* a list element has thrown during its execution, preventing the remaining list elements from being invoked.

Therefore we introduce a new universe $Delegate = Meth \cup (Ref \times Meth)$. To express the creation and use of new delegate expressions the sets $Exp, Sexp$ are extended by additional grammar rules as follows, using a new set $Dexp$ of delegate expressions:

$$\begin{array}{lll} Sexp & ::= & \ldots \mid Exp\ \texttt{(}\ [Args]\ \texttt{)} \\ Exp & ::= & \ldots \mid \texttt{`new'}\ DelegateType\ \texttt{`(' }\ Dexp\ \texttt{`)'} \\ Dexp & ::= & Meth \mid Type\ \texttt{`.' }\ Meth \mid Exp\ \texttt{`.' }\ Meth \mid Exp \end{array}$$

A method T::M is called *compatible* with the delegate type D iff T::M and D have the same return type and the same number of parameters with the same parameter types (including `ref`, `out`, `params` modifiers). The type constraints on the new expressions are spelt out in [19].

We use the model ExecCsharpStm$_I$, which includes a description of the `for` statement of C♯$_{\mathcal{I}}$, to express the sequentiality of the execution of delegate invocation list elements. In fact the above delegate declaration can be translated for $T \neq$ `void` in the following class:

```
sealed class D : System.Delegate {
  public T Invoke(S1 x1, ... ,Sn xn) {
    T result;
    for (int i = 0; i < this._length() ; i++)
      result = this._invoke(i,x1, ... ,xn);
    return result;
  }
  private extern int _length();
  private extern T _invoke(int i,S1 x1, ... ,Sn xn);
}
```

A delegate invocation expression *exp*`(`*args*`)` can be syntactically translated into a normal method call *exp*`.`D`::Invoke(`*args*`)` where D is the type of *exp*.[21] It then suffices to refine the ASM rule InvokeExtern defined in the model ExecCsharpExp$_C$ to describe the meaning of the method D`::_invoke`, which is to invoke the ith element of the invocation list on the given arguments, and

[21] In [2, §10.4.7] the members of a delegate are defined to be the members inherited from the class `System.Delegate`. However neither .NET nor Rotor nor Mono do respect this stipulation since they add further methods to those inherited. One such example is the method `_invoke` we use here.

analogously of `_length`.

```
InvokeExtern(T::M, vals) ≡
  if T ∈ DelegateType then
    if name(M) = _length then
      DelegateLength(vals(0))
    if name(M) = _invoke then
      InvokeDelegate(vals(0), vals(1), drop(vals, 2))

DelegateLength(ref) ≡
  YieldUp(length(invocationList(ref)))

InvokeDelegate(ref, i, vals) ≡
  match invocationList(ref)(i)
    T::M            → InvokeStatic(T::M, vals)
    (target, T::M)  → InvokeInstance(T::M, target, vals)
```

Since there are no new statements appearing in $C\sharp_{\mathcal{D}}$, the addition of the rule ExecCsharp$_D$ to ExecCsharp$_E$ consists in the following ASM subrule ExecCsharpExp$_D$, which defines the meaning of delegate instance creation. For a detailed analysis of the discrepancy we exhibit here between the ECMA standard and the .NET implementation see [13].

```
ExecCsharpExp_D ≡ match context(pos)
  new D(T::M) →
    let d = new(Ref, D) in
      runTimeType(d) := D
      invocationList(d) := [T::M]
      Yield(d)
  new D(exp . T::M)   → pos := exp
  new D(▶ref . T::M) →
    if ref = Null then FailUp(NullReferenceException)
    else let d = new(Ref, D) in
      runTimeType(d) := D
      invocationList(d) := [(ref, T::M)]
      YieldUp(d)
  new D(exp)   → pos := exp
  new D(▶ref) →
    if ref = Null then FailUp(NullReferenceException)
    else let d = new(Ref, D) in
      runTimeType(d) := D
      invocationList(d) := invocationList(ref)   // ECMA §14.5.10.3
      // Microsoft .NET Framework:
      // invocationList(d) := [(ref, D::Invoke)]
      YieldUp(d)
```

To be complete, one should add some rules which reflect the special character of delegate invocation lists. As usual for lists, two invocation lists are *equal* (`==`) iff they have the same length and the elements of the lists are pairwise equal; they can be *combined* (concatenated with '`+`') and sublists determined by a particular prefix and suffix condition can be *removed* from them (with '`-`'). To describe this specialization of list operations in our model it suffices to refine the macro InvokeExtern by new rules for these operators `operator+`, `operator-`, `operator==`.

InvokeExtern(T::M, *vals*) ≡
  ...
  if $T \in$ *DelegateType* **then**
    if *name*(M) = `operator+` **then**
      DelegateCombine(T, *vals*(0), *vals*(1))
    if *name*(M) = `operator-` **then**
      DelegateRemove(T, *vals*(0), *vals*(1))
    if *name*(M) = `operator==` **then**
      DelegateEqual(*vals*(0), *vals*(1))

Since invocation lists are considered to be immutable, combination and removal return *new* delegate instances unless one of the arguments is `null`. The `null` reference represents a delegate instance with an empty invocation list.

DelegateCombine(D, r_1, r_2) ≡
  if $r_1 = Null$ **then** YieldUp(r_2)
  elseif $r_2 = Null$ **then** YieldUp(r_1)
  else let $d = new(Ref, D)$ **in**
    runTimeType(d) := D
    invocationList(d) := *invocationList*(r_1) · *invocationList*(r_2)
    YieldUp(d)

DelegateRemove(D, r_1, r_2) ≡
  if $r_1 = Null$ **then** YieldUp(*Null*)
  elseif $r_2 = Null$ **then** YieldUp(r_1)
  else let l_1 = *invocationList*(r_1) **and** l_2 = *invocationList*(r_2) **in**
    if $l_1 = l_2$ **then** YieldUp(*Null*)
    elseif *Subword*(l_2, l_1) **then let** $d = new(Ref, D)$ **in**
      runTimeType(d) := D
      invocationList(d) := *prefix*(l_2, l_1) · *suffix*(l_2, l_1)
      YieldUp(d)
    else YieldUp(r_1)

The notions of *prefix* and *suffix* are defined here in terms of the *last* occurrence of a subword: *prefix*(u, v) is the part of v before the last occurrence of u in v

and $\mathit{suffix}(u, v)$ the part of v after the last occurrence of u in v.

$\textsc{DelegateEqual}(r_1, r_2) \equiv$
  if $r_1 = \mathit{Null} \vee r_2 = \mathit{Null}$ **then** $\textsc{YieldUp}(r_1 = r_2)$
  else let $l_1 = \mathit{invocationList}(r_1)$ **and** $l_2 = \mathit{invocationList}(r_2)$ **in**
    $\textsc{YieldUp}(\mathit{length}(l_1) = \mathit{length}(l_2) \wedge \forall i < \mathit{length}(l_1)\,(l_1(i) = l_2(i)))$

6.2 Properties, events and further features in $\mathrm{C}\sharp_{\mathcal{D}}$

In this section we add further language features of C♯ whose semantics can be easily defined in terms of the model developed so far, essentially by simple syntactical reductions which one can easily justify to formalize correctly the explanations in [2].

Properties. Collections of a read and/or a write method for attributes of a class or struct are called *properties* in C♯ and declared in the following form (we skip the modifiers):

Type Identifier '`{`' ['`get`' *Block*] ['`set`' *Block*] '`}`'

By definition a *read-write* property has a `get` and a `set` accessor, a *read-only* property has only a `get` accessor, a *write-only* property has only a `set` accessor. The identifier of a property P of type T can be used like a field identifier,[22] except that it cannot be passed as `ref` or `out` argument. Furthermore it is required that the body of a `get` accessor is the body of a method with return type T, that a `set` accessor has a value parameter named `value` of type T and that its body is the body of a `void` method. Using the signatures T `get_`P`();` and `void set_`P`(`T `value);`, which are reserved for get and set accessors, the intended semantics of properties is reduced to the semantics of methods, using the following syntactical reductions:

$$\begin{array}{l} T\ P\ \texttt{\{} \\ \quad \texttt{get \{}\ \mathit{getAccessor}\ \texttt{\}} \\ \quad \texttt{set \{}\ \mathit{setAccessor}\ \texttt{\}} \\ \texttt{\}} \end{array} \quad \Longrightarrow \quad \begin{array}{l} T\ \texttt{get\_}P\texttt{() \{}\ \mathit{getAccessor}\ \texttt{\}} \\ \texttt{void set\_}P\texttt{(}T\ \texttt{value) \{} \\ \quad \mathit{setAccessor} \\ \texttt{\}} \end{array}$$

$$\mathit{exp}\,.\,P \Longrightarrow \mathit{exp}\,.\,\texttt{get\_}P\texttt{()} \qquad \mathit{exp}_1\,.\,P\ \texttt{=}\ \mathit{exp}_2\,\texttt{;} \Longrightarrow \mathit{exp}_1\,.\,\texttt{set\_}P\texttt{(}\mathit{exp}_2\texttt{);}$$

The above translation comprises also expressions of the form $\mathit{exp}_1\,.\,P\ \mathit{op}\texttt{=}\ \mathit{exp}_2$, because we can assume that these compound assignments are compiled to

[22] Without knowing whether it is accessed directly or whether an accessor method is being called.

$\langle x$ `=` $exp_1,\ y$ `=` x`.get_P()` $op\ exp_2,\ x$`.set_P(`y`)`$,\ y\rangle$ with fresh local variables x, y, using as auxiliary operator the comma operator familiar from C/C++. This necessitates auxiliary rules for going through sequences of expressions of the following form:

$$\begin{array}{ll}\langle exp\,,\ldots\rangle & \to pos := exp\\ \langle val_1\,,\ldots,{\blacktriangleright}val_n\rangle & \to \textsc{YieldUp}(val_n)\\ \langle\ldots{\blacktriangleright}val\,,exp\ldots\rangle & \to pos := exp\end{array}$$

Indexers. Indexers can be used like array elements except that they cannot contain `ref` or `out` parameters and their elements cannot be passed as `ref` or `out` arguments. They are declared in a class or struct type as follows (we skip the modifiers):

Type '`this`' '`[`' [*Params*] '`]`' '`{`' ['`get`' *Block*] ['`set`' *Block*] '`}`'

Analogously to the constraints for properties, for an indexer of type T with parameters p, the body of a `get` accessor is the body of a method with parameters p and return type T, the body of a `set` accessor is the body of a `void` method with parameters p and an implicit value parameter named `value` of type T. A base class indexer can be accessed by `base[`*exps*`]`. Using the signatures T `get_Item(`*params*`)` and `void set_Item(`*params*`,`T `value)`, which are are reserved for get and set accessors, the intended semantics of indexers is reduced to the semantics of arrays and methods via the following compile-time translation (and corresponding operator expression translation as explained for properties):

$$\begin{array}{l}T\ \texttt{this[}params\texttt{]}\ \texttt{\{}\\ \quad\texttt{get \{}\ getAccessor\ \texttt{\}}\\ \quad\texttt{set \{}\ setAccessor\ \texttt{\}}\\ \texttt{\}}\end{array}\quad\Longrightarrow\quad\begin{array}{l}T\ \texttt{get\_Item(}params\texttt{)\ \{}\ getAccessor\ \texttt{\}}\\ \texttt{void set\_Item(}params\texttt{,}\ T\ \texttt{value) \{}\\ \quad setAccessor\\ \texttt{\}}\end{array}$$

Events. Events can be declared in C♯ like fields as follows: in the form '`event`' *DelegateType Identifier* '`;`' (we omit the modifiers), or like properties, in the form

'`event`' *DelegateType Identifier* '`{`' '`add`' *Block* '`remove`' *Block* '`}`'.

Outside the type that contains the declaration, an event X can only be used as the left-hand operand of `+=` and `-=` in expressions X `+=` *exp* and X `-=` *exp* of type `void`; within the type that contains the declaration, field-like events can be used like fields of delegate types. The accessors of property-like events have to be bodies of `void` methods with an implicit parameter `value` of *DelegateType*.

The semantics of events in C♯ follows the *Publish/Subscribe* pattern. A class publishes an event it can raise, so that any number of classes can subscribe to that event. When the event is actually raised, each subscriber is notified that the event has occurred, namely by calling a delegate whose invocation list is executed with the sender object and the event data as its arguments. This idea is realized as follows. The *event sender* class that raises an event named X has the member `event` X`_EventHandler` X`;` where the delegate type X`_EventHandler` for the event is declared as follows (with two arguments, the first one for the publisher and the second one for the event information object, which must be derived from the class `EventArgs`):

`delegate void` X`_EventHandler(object` *sender*`,` X`_EventArgs` e`);`

To consume the event, the *event receiver* declares an event-handling method `Receive_`X with the same signature as the event delegate:

`void Receive_`X`(object` *sender*`,` X`_EventArgs` e`) { ... }`

To *register* the event handler, the event receiver has to add the `Receive_`X method to the event X of the event sender object:

X `+= new` X`_EventHandler(this.Receive_`X`);`

The event sender *raises* the event by invoking the invocation list of X with the sender object and the event data, e.g.

`void On_`X`(`X`_EventArgs` e`) { if (`X `!= null)` X`(this,` e`); }`

It suffices to assign a meaning to the signatures `void add_`X`(`D `value)` and `void remove_`X`(`D `value)`, which are reserved for every event X of delegate type D. This is done by the following translation of field-like events, anticipating the `lock` statement of C♯$_{\mathcal{T}}$ which is explained in [17]. [23]

```
class C {
  private D __X;
  void add_X(D value) {
    lock (this) { __X = __X + value; }
  }
  void remove_X(D value) {
```

[23] If one prefers to be independent of the thread model C♯$_{\mathcal{T}}$, one can consider lock statements `lock (`*exp*`)` *stm* translated for single-thread execution by `{ Object` o `=` *exp*`;` *stm* `}` (with a fresh variable o), which is then refined in C♯$_{\mathcal{T}}$ for the multiple thread execution model.

```
    lock (this) { __X = __X - value; }
  }
}
```

Further constructs. Similar syntactical reductions to those given above can be used to define the semantics of overloaded standard mathematical operators and user-defined conversions, of enumeration related statements '`foreach (`T x `in` *exp*`)` *stm*', of using statements '`using (` *resource* `)` *stm*', of parameter arrays and of attributes.

7 Refinement C♯$_\mathcal{U}$ by pointers in unsafe code

In this section we add the features C♯ offers for using pointers (coming with address-of and dereferencing operators '`&`', '`*`', '`->`' together with pointer arithmetic) to directly work on memory addresses, bypassing the type checking by the compiler – hence the name '`unsafe`' code blocks. Java has no such unsafe extension. The extension includes a mechanism called *pinning* of objects to prevent the runtime during the execution of a '`fixed`' statement to manage via the garbage collector memory one wants to address directly. Code for which (de-) allocation is not controlled by the runtime is called *unmanaged*. As an alternative to pinning, data of unmanaged type can also be '`stackalloc`'ated, instead of using the heap.

The refinement consists, besides some new rules, mainly in a definition of the *mem*ory function in terms of byte sequences. This is a typical data refinement, using an encoding of simple types and a corresponding refinement of the function *structField*.

7.1 Signature refinement for C♯$_\mathcal{U}$

We refine *Type* by adding pointer types to value and reference types.

$$\begin{array}{lll} \mathit{Type} & ::= & \mathit{ValueType} \mid \mathit{RefType} \mid \mathit{PointerType} \\ \mathit{PointerType} & ::= & \mathit{UnmanagedType}\ \text{`}\texttt{*}\text{'} \mid \text{`}\texttt{void}\text{'}\ \text{`}\texttt{*}\text{'} \end{array}$$

where *unmanaged* types are types which are not managed and *managed* types are recursively defined as a) reference types or b) struct types that contain a field of a managed type or a pointer to a managed type. The subtype relation is extended to pointer types such that $\Lambda \preceq T\texttt{*} \preceq \texttt{void*}$. *Exp* and *Vexp* are extended by address-of and dereferencing expressions and expressions to denote the values of a new function indicating the '`sizeof`' unmanaged types. *Stm* is extended to reflect '`unsafe`' code blocks, '`fixed`' statements and

Table 6
Type constraints for unsafe expressions.

Expression	Constraints	Expression type
`sizeof(`t`)`	t unmanaged type	`int`
`*`e	*type*(e) = T`*`, $T \neq$ `void`	T
`&`v	v a fixed variable	T`*`, where T = *type*(v)
e `->` m	*type*(e) = T`*`, $T \neq$ `void`	*type*(T::m)
e`[`i`]`	*type*(e) = T`*`, $T \neq$ `void`, *type*(i) integral	T

'`stackalloc`'ation of arrays. '`unsafe`' can also appear as modifier for classes, structs, interfaces, delegates as well as for fields, methods, properties, indexers, operators, events, constructors, destructors.

Exp ::= ... | '`&`' *Vexp* | *Exp* '`->`' *Meth* `(` [*Args*] `)` | *Exp* '`->`' *Field*
| '`sizeof`' '`(`' *UnmanagedType* '`)`'

Vexp ::= ... | '`*`' *Exp*

Stm ::= ... | '`unsafe`' *Block* | '`fixed`' '`(`' *PointerType Loc* `=` *Exp* '`)`' *Stm*

Bstm ::= ... | *PointerType Loc* '`=`'
'`stackalloc`' *UnmanagedType* '`[`' *Exp* '`]`' '`;`'

In the following expressions, the basic arithmetical operators are used for pointer increment and decrement, pointer addition and subtraction, pointer comparison, and pointer conversion (where p and q are of a pointer type, i is of integer type):

- `++`p, `--`p, p`++`, p`--`, p `+` i, i `+` p, p `-` i, p `-` q, p `==` q, p `!=` q, p `<` q, p `<=` q, p `>` q, p `>=` q
- `(`T`*)`i, `(`T`*)`p, `(int)`p, `(uint)`p, `(long)`p, `(ulong)`p

On the types of the new expressions the constraints in Table 6 are imposed. We assume the dereferencing and member access operator e`->` m to be translated to `(*`e`).`m, similarly e`[`i`]` to `*(`e `+` i`)`.

For statements the following type constraints in Table 7 are assumed. A variable is called *moveable* (by the garbage collector) iff it is not a fixed variable. Fixed variables are (by recursive definition): local variables, value parameters, `*`*exp* for *exp* of pointer type, and instance field expressions v`.`f if v is a fixed variable of struct type T and f is an instance field of T.

The local variable p in the fixed statement is called a *pinned* local variable. A pinned local variable is a read-only variable. It is not allowed to assign a new value to it in the body of the fixed statement.

Table 7
Type constraints for unsafe statements.

Statement	Constraints
T`*` p `= stackalloc` T`[`exp`];`	$type(exp) =$ `int`, T unmanaged
`fixed (char*` p `=` exp`)` stm	$type(exp) =$ `string`, p read-only in stm
`fixed (`T`*` p `=` exp`)` stm	$type(exp) = T$`[`R`]`, T unmanaged, p read-only in stm
`fixed (`T`*` p `= &`$vexp$`)` stm	$type(vexp) = T$, T unmanaged, $vexp$ a moveable variable, p read-only in stm

The principal refinement in the ASM extension ExecCsharp$_U$ for C♯$_\mathcal{U}$ is that of the *mem*ory together with its operators, where the set of *SimpleValue*s is replaced by *Byte*s (8-bit strings), using non-negative integers as memory addresses ($Adr = \mathbb{N}$):

$$mem\colon Adr \to Byte \cup Ref \cup \{Undef\}$$

The partial functions to *encode* (resp. *decode*) values of a given simple type T by byte sequences, of a length (number of bytes) depending on $sizeOf(T)$, satisfy for values val the equations

$$decode(T, encode(val)) = val \quad \text{and} \quad length(encode(val)) = sizeOf(T).$$

For every pointer type T`*` holds $sizeOf(T*) = sizeOf($`void*`$)$.

A function $fieldOffset\colon UnmanagedStructType \times Field \to \mathbb{N}$ is used to describe the layout of unmanaged structs. It has to satisfy the following constraint for every unmanaged struct type T and instance field f of T (overlapping fields are allowed in C♯$_\mathcal{U}$):

$$fieldOffset(T, f) + sizeOf(type(f)) \leq sizeOf(T)$$

We assume that if adr is an address allocated using $new(Adr, T)$ for struct type T, then for every instance field f of T the equation $fieldAdr(adr, f) = adr + fieldOffset(T, f)$ holds.

To determine the layout of arrays with unmanaged element type we stipulate the following refinement of the function *elemAdr* which reflects that array elements are stored such that the indices of the right most dimension are increased first, then the next left dimension, and so on. For $runTimeType(ref) =$ T`[`$dim(n)$`]`, where T is an unmanaged type and $l_i = dimLength(ref, i - 1)$

for $i \in [1\,..\,n]$, we assume the following:

$$\begin{array}{l} \mathit{elemAdr}(\mathit{ref}, [i_1, i_2, \ldots, i_n]) = \\ \quad \mathit{elemAdr}(\mathit{ref}, [0, \ldots, 0]) + (\ldots(i_1 \cdot l_2 + i_2) \cdot l_3 + \ldots + i_n) \cdot \mathit{sizeOf}(T) \end{array}$$

7.2 Transition rule refinement for unsafe code

Besides the rules below which define the semantics of the new expressions and statements, to be added to ExecCsharp$_D$, we have to data refine the notions of reading from and writing to memory for values of unmanaged type.

$$\begin{array}{l} \mathit{memValue}(\mathit{adr}, t) = \\ \quad \textbf{if } t \in \mathit{RefType} \textbf{ then } \mathit{mem}(\mathit{adr}) \\ \quad \textbf{elseif } t \in \mathit{UnmanagedType} \textbf{ then} \\ \quad\quad [\mathit{mem}(\mathit{adr} + i) \mid i \in [0\,..\,\mathit{sizeOf}(t) - 1]\,] \\ \quad \textbf{elseif } t \in \mathit{StructType} \textbf{ then} \\ \quad\quad \{f \mapsto \mathit{getField}(\mathit{adr}, f) \mid f \in \mathit{instanceFields}(t)\} \end{array}$$

$$\mathit{getField}(\mathit{adr}, t::f) = \mathit{memValue}(\mathit{fieldAdr}(\mathit{adr}, t::f), \mathit{type}(t::f))$$

$$\begin{array}{l} \textsc{WriteMem}(\mathit{adr}, t, \mathit{val}) \equiv \\ \quad \textbf{if } t \in \mathit{RefType} \textbf{ then } \mathit{mem}(\mathit{adr}) := \mathit{val} \\ \quad \textbf{elseif } t \in \mathit{UnmanagedType} \textbf{ then} \\ \quad\quad \textbf{forall } i \in [0\,..\,\mathit{sizeOf}(t) - 1] \textbf{ do } \mathit{mem}(\mathit{adr} + i) := \mathit{val}(i) \\ \quad \textbf{elseif } t \in \mathit{StructType} \textbf{ then} \\ \quad\quad \textbf{forall } f \in \mathit{instanceFields}(t) \textbf{ do } \textsc{SetField}(\mathit{adr}, f, \mathit{val}(f)) \end{array}$$

$$\textsc{SetField}(\mathit{adr}, t::f, \mathit{val}) \equiv \textsc{WriteMem}(\mathit{fieldAdr}(\mathit{adr}, t::f), \mathit{type}(t::f), \mathit{val})$$

Values of unmanaged struct types are directly represented as sequences of bytes. Hence, the function *structField* has to be refined to extract a subsequence in case of unmanaged struct types:

$$\begin{array}{l} \mathit{structField}(\mathit{val}, T, f) = \\ \quad \textbf{if } T \in \mathit{ManagedType} \textbf{ then } \mathit{val}(f) \\ \quad \textbf{else let } n = \mathit{fieldOffset}(T, f) \textbf{ in } [\mathit{val}(i) \mid n \le i < n + \mathit{sizeOf}(\mathit{type}(f))] \end{array}$$

In the rules for ExecCsharpExp$_U$ we have `&` $\square$ as additional address position. We follow the implementation in Rotor and .NET in formulating the *Null* check to prevent writing to null addresses; the ECMA standard describes this check as optional.

$$\begin{array}{l} \textsc{ExecCsharpExp}_U \equiv \textbf{match } \mathit{context}(\mathit{pos}) \\ \quad \texttt{sizeof(}T\texttt{)} \rightarrow \textsc{Yield}(\mathit{sizeOf}(T)) \end{array}$$

$\texttt{\&}exp \quad \rightarrow pos := exp$
$\texttt{\&}^{\blacktriangleright} adr \rightarrow \textsc{YieldUp}(adr)$

$\texttt{*}exp \quad \rightarrow pos := exp$
$\texttt{*}^{\blacktriangleright} adr \rightarrow \textbf{if } adr = Null \textbf{ then} \quad //$ null pointer check optional
$\textsc{FailUp}(\texttt{NullReferenceException})$
$\textbf{else } \textsc{YieldUpIndirect}(adr)$

$\texttt{*}exp_1 \texttt{ = } exp_2 \rightarrow pos := exp_1$
$\texttt{*}^{\blacktriangleright} adr \texttt{ = } exp_2 \rightarrow pos := exp_2$
$\texttt{*}adr \texttt{ = }^{\blacktriangleright} val \rightarrow \textbf{if } adr = Null \textbf{ then} \quad //$ null pointer check optional
$\textsc{FailUp}(\texttt{NullReferenceException})$
$\textbf{else}$
$\textsc{WriteMem}(adr, type(pos), val)$
$\textsc{YieldUp}(val)$

The rules for pointer arithmetic can be summarized as follows:

$Apply(\texttt{+}(T\texttt{*}, \texttt{int}), adr, i) = adr + i \cdot sizeOf(T)$
$Apply(\texttt{+}(\texttt{int}, T\texttt{*}), i, adr) = adr + i \cdot sizeOf(T)$
$Apply(\texttt{-}(T\texttt{*}, T\texttt{*}), adr_1, adr_2) = (adr_1 - adr_2)/sizeOf(T)$
$Convert(T\texttt{*}, adr) = adr = Convert(S, adr)$
for $S \in \{\texttt{int}, \texttt{uint}, \texttt{long}, \texttt{ulong}\}$
$Convert(T\texttt{*}, i) = i$

In the execution of the stackalloc statement we assume that $new(adr, T, i)$ allocates i consecutive chunks of addresses of size $sizeOf(T)$ which are later de-allocated on method exit in FreeLocals.

$\textsc{ExecCsharpStm}_U \equiv \textbf{match } context(pos)$
$\texttt{unsafe } block \quad \rightarrow pos := block$
$\texttt{unsafe } ^{\blacktriangleright} Norm \rightarrow \textsc{YieldUp}(Norm)$

$T\texttt{*} \; loc \texttt{ = stackalloc } T\texttt{[}exp\texttt{];} \rightarrow pos := exp$
$T\texttt{*} \; loc \texttt{ = stackalloc } T\texttt{[}^{\blacktriangleright} i\texttt{];} \rightarrow \textbf{let } adr = new(Adr, T, i) \textbf{ in}$
$\textsc{WriteMem}(locals(loc), T\texttt{*}, adr)$
$\textsc{YieldUp}(Norm)$

The run-time execution of fixed statements can be explained by syntactical transformations.

Statement	Run-time execution
`fixed (char*` p `=` exp`)` stm	`{ char*` p`;` p `=` $Cstring$`(`exp`);` stm `}`
`fixed (`T`*` p `=` exp`)` stm	`{` T`*` p`;` p `= &`exp`[0];` stm `}`
`fixed (`T`*` p `= &`$vexp$`)` stm	`{` T`*` p`;` p `= &`$vexp$`;` stm `}`

In the first case, it is assumed that *Cstring*(s) is an internal function that returns the address of the first element of a C-style null-terminated character array representation of the string s. How it is related to the original representation of the string is not specified in [2].

8 Related Work and Conclusion

One of our referees would like to see a critical assessment of the ASM method we used for this work and a comparison to alternative approaches. Some justification of the kind from the perspective of semantic methods for programming languages has been given in [26, Sect.4], containing concrete illustrations of and references to the numerous and earlier competing proposals. This was at a time when ASMs were applied for the first time to (successfully) specify an industrial language standard, namely the ISO Prolog standard [27]. A decade later, a broader comparison of the then well-developed ASM method with respect to other system design and analysis frameworks has been provided in [28,29]. However, a systematic, comprehensive and authoritative evaluation of the multitude of system design and analysis proposals in the literature remains a highly desirable and challenging task to be accomplished, even if limited to the use of the major so-called formal methods for the development and investigation of programming languages and their implementations. From the perspective of practical system design and analysis some comparative studies of this kind have been published, see e.g. [30–32] (the interested reader may also consult the corresponding ASM-based work in [33–35]). For work centered around Java and the JVM the reader finds in [36] a collection of formal-method-approaches to language specification and analysis; [37] contains an excellent, detailed and at the time complete review of the huge literature on the subject (including an evaluation of the ASM-based Java/JVM investigations), with a focus on safety issues and their impact on smart cards. We cannot perform here a similar analysis for work on C♯ or other major programming languages. This explains why the references in this paper stick to C♯ documentation from ECMA and Microsoft and to some ASM work we have built upon directly.

For the work presented in this paper we set ourselves a more modest though not completely trivial major goal, namely to test whether the method developed in [1] for the definition and a proven to be correct implementation of a real-life programming language like Java scales naturally to the somewhat richer and more complex C♯. It is up to the reader to judge whether this ASM reuse case study for a real-life complex model succeeded. For the formalization of other programming languages something can also be learnt directly from the formalization of the semantics of C♯ worked out here. For example, how to "divide and conquer" the static and the dynamic semantics of a language, how to

separate the description of conceptually independent programming constructs by dividing them into sublanguages, how to unify and streamline the formalization of similar constructs by appropriate parameterizations (which means abstractions), how to model and evaluate variations of specific features (e.g. expression evaluation, parameter passing mechanism, class initialization, etc.) by varying macros, rules and/or domains together with their operations, how to extend within a single framework the model for a language core by a form of bootstrapping (including in particular syntactical translations) to a model for the entire language, etc.

There are several by-products of the work presented here. Through the ASM-model-oriented analysis of the ECMA standard for C♯ we found several bugs and gaps in the formulation of the standard and in its .NET implementation as well as some incoherences between the two, as documented in detail in [13] in terms of our ASM model for C♯. Another by-product of the high-level modular interpreter defined here is the support it provides to teachers of C♯, in particular if they want to shed light on certain subtle language features which are not clarified by the ECMA documents. In the forthcoming paper [38] we are going to work out a concrete comparison of the two models for C♯ and for Java, which will allow us to 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. As a specific part of this reuse-case-study the second author is investigating how the main new features of C♯ 2.0 can be modeled by appropriate extensions of the ASM model developed here for C♯, in particular generic types (parametric polymorphism), anonymous methods and iterators. Last but not least, with our C♯ model and its extension to threads in [17] we have laid the ground for a mathematical analysis and possibly mechanical verification of interesting properties of the language and its implementation, like type safety, [24] compiler correctness and completeness, correctness of (a mathematical model to be developed for) garbage collection, security, etc. For the correctness of the definite assignment analysis performed by a C♯ compiler, we may refer the interested reader to [21]. We hope somebody will feel challenged to use our model for precisely formulating and proving such theorems for C♯ and to build a corresponding model for Microsoft's Common Language Runtime together with a compilation scheme from C♯ to IL code, applying to our model the powerful ASM refinement technique [40] along the lines shown in the ASM-based Java/JVM study in [1].

The questions asked by our referees lead us to mention another practical and

[24] For a fragment of Microsoft's Intermediate Language, which is executed by Microsoft's Common Language Runtime, a type safety proof has been given in [20], based upon Syme's method [39] for writing functional specifications which can be subject to theorem proving in HOL.

industrially viable use that can be made of a modeling and analysis activity as the one reported in this paper, except if the extreme time pressure usually imposed on developers to produce executable code from incomplete verbal specifications (mostly formulated in natural language) prevents them from at least once trying out a more reliable option. Here is a concrete example what could have been done. On September 27, 2000, the penultimate day of his sabbatical stay with Microsoft Research in Redmond, in a seminar talk to representatives of the C♯ development team, the first author suggested to use the method, at the time formulated and presented in terms of Java/JVM for publication in what became the Jbook [1], for the following five fundamental activities in relation to the at-the-time ongoing development of what became known as the C♯ language with the underlying CLR virtual machine:

- defining an ASM model as *executable specification of critical language constructs* or layers (if not of the entire language) and of the mapping to IL code,
- *generating test cases* for the implementing code from the ASM model,
- using the ASM model as *oracle for test evaluations* and for comparing model test runs with code test runs,
- using the ASM model as *internal documentation* for future language extensions and for relating other .NET languages to C♯, in particular those which are equipped already with an ASM model of their semantics,
- using the ASM model as *basis for writing innovative handbooks* for users and for maintenance professionals, where the innovative character derives from being a) accurate yet simple and easy to understand, b) complete and detailed yet succinct.

Acknowledgements

We gratefully acknowledge partial support of this work by a Microsoft grant within the ROTOR project during the year 2002–2003. We thank Bruno Quarta for attracting us to the C♯ modeling work, even if *post festam*, as part of the ROTOR project. We also thank two anonymous referees for valuable criticism which helped us improve the exposition.

References

[1] R. F. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine—Definition, Verification, Validation, Springer-Verlag, 2001.

[2] C♯ Language Specification, Standard ECMA–334, `http://www.ecma-international.org` (2001).

[3] A. Hejlsberg, S. Wiltamuth, P. Golde, C♯ Language Specification, Addison-Wesley, 2003.

[4] T. Archer, A. Whitechapel, Inside C♯, Microsoft Press, 2002.

[5] J. Prosise, Programming Microsoft .NET, Microsoft Press, 2002.

[6] J. Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002.

[7] Visual Studio .NET 2003, `http://msdn.microsoft.com/vstudio/`.

[8] SSCLI (Rotor) web site, `http://www.sscli.net`.

[9] Mono compiler for C♯, `http://www.go-mono.com/c-sharp.html`.

[10] Common Language Infrastructure, Standard ECMA–335, `http://www.ecma-international.org` (2003).

[11] D. Stutz, T. Neward, G. Shilling, Shared Source CLI Essentials, O'Reilly, 2003.

[12] E. Börger, The ASM ground model method as a foundation of requirements engineering, in: N. Dershowitz (Ed.), Manna-Symposium, Vol. 2772 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[13] N. G. Fruja, Specification and implementation problems for C♯, in: B. Thalheim, W. Zimmermann (Eds.), Abstract State Machines 2004, Lecture Notes in Computer Science, Springer-Verlag, 2004.

[14] Foundations of Software Engineering Group, Microsoft Research, AsmL, Web pages at `http://research.microsoft.com/foundations/AsmL/` (2001).

[15] J. Schmid, Executing ASM specifications with AsmGofer, Web pages at `http://www.tydo.de/AsmGofer`.

[16] J. Schmid, Refinement and implementation techniques for Abstract State Machines, Ph.D. thesis, University of Ulm, Germany (2002).

[17] R. F. Stärk, E. Börger, An ASM specification of C♯ threads and the .NET memory model, in: B. Thalheim, W. Zimmermann (Eds.), Abstract State Machines 2004, Lecture Notes in Computer Science, Springer-Verlag, 2004.

[18] E. Börger, R. F. Stärk, Abstract State Machines. A Method for High-Level System Design and Analysis, Springer-Verlag, 2003.

[19] E. Börger, N. G. Fruja, V. Gervasi, R. Stärk, A complete formal definition of the semantics of C♯, Technical report, In preparation (2004).

[20] A. D. Gordon, D. Syme, Typing a multi-language intermediate code, in: Proc. 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, 2001, pp. 248–260.

[21] N. G. Fruja, The correctness of the definite assignment analysis in C♯, Technical report 435, Computer Science Department, ETH Zürich (2004).

[22] E. Börger, T. Bolognesi, Remarks on turbo ASMs for computing functional equations and recursion schemes, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Abstract State Machines 2003 – Advances in Theory and Applications, Vol. 2589 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 218–228.

[23] W. Zimmermann, A. Dold, A framework for modeling the semantics of expression evaluation with Abstract State Machines, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Abstract State Machines 2003–Advances in Theory and Applications, Vol. 2589 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 391–406.

[24] N. G. Fruja, R. F. Stärk, The hidden computation steps of turbo Abstract State Machines, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Abstract State Machines 2003 – Advances in Theory and Applications, Vol. 2589 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 244–262.

[25] E. Börger, J. Schmid, Composition and submachine concepts for sequential ASMs, in: P. Clote, H. Schwichtenberg (Eds.), Computer Science Logic (Proceedings of CSL 2000), Vol. 1862 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 41–60.

[26] E. Börger, A logical operational semantics for full Prolog. Part I: Selection core and control, in: E. Börger, H. Kleine Büning, M. M. Richter, W. Schönfeld (Eds.), CSL'89. 3rd Workshop on Computer Science Logic, Vol. 440 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 36–64.

[27] E. Börger, K. Dässler, Prolog: DIN papers for discussion, ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England (1990).

[28] E. Börger, High-level system design and analysis using Abstract State Machines, in: D. Hutter, W. Stephan, P. Traverso, M. Ullmann (Eds.), Current Trends in Applied Formal Methods (FM-Trends 98), Vol. 1641 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 1–43.

[29] E. Börger, Abstract State Machines: A unifying view of models of computation and of system design frameworks, Annals of Pure and Applied LogicTo appear.

[30] C. Lewerentz, T. Lindner, Formal Development of Reactive Systems. Case Study "Production Cell", Vol. 891 of Lecture Notes in Computer Science, Springer-Verlag, 1995.

[31] J.-R. Abrial, E. Börger, H. Langmaack, The steam boiler case study: Competition of formal program specification and development methods, in: J.-R. Abrial, E. Börger, H. Langmaack (Eds.), Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control, Vol. 1165 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 1–12.

[32] M. Broy, S. Merz, K. Spies, Formal Systems Specification – The RPC-Memory Specification Case Study, Vol. 1169 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[33] E. Börger, L. Mearelli, Integrating ASMs into the software development life cycle, J. Universal Computer Science 3 (5) (1997) 603–665.

[34] C. Beierle, E. Börger, I. Durdanović, U. Glässer, E. Riccobene, Refining abstract machine specifications of the steam boiler control to well documented executable code, in: J.-R. Abrial, E. Börger, H. Langmaack (Eds.), Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control, no. 1165 in Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 62–78.

[35] J. Huggins, Broy-Lamport Specification Problem: A Gurevich Abstract State Machine Solution, Technical Report CSE-TR-320-96, EECS Dept., University of Michigan (1996).

[36] J. Alves-Foss, Formal Syntax and Semantics of Java, Vol. 1523 of Lecture Notes in Computer Science, Springer-Verlag, 1998.

[37] P. Hartel, L. Moreau, Formalizing the safety of Java, the Java Virtual Machine and Java Card, ACM Computing Surveys 33 (4) (2001) 517–558.

[38] E. Börger, R. F. Stärk, Exploiting abstraction for specification reuse. The Java/C♯ case study, in: M. B. et al. (Ed.), Proc. FMCO'03, Lecture Notes in Computer Science, Springer-Verlag, 2004.

[39] D. Syme, Declarative theorem proving for operational semantics, Ph.D. thesis, University of Cambridge (1998).

[40] E. Börger, The ASM refinement method, Formal Aspects of Computing 15 (2003) 237–257.