Exploiting Abstraction for Specification Reuse. The Java/C# Case Study

Egon Börger and Robert F. Stärk

March 5, 2004

Abstract

From the models provided in [11] and [4] for the semantics of Java and C# programs we abstract the mathematical structure that underlies the semantics of both languages. The resulting model reveals the kernel of object-oriented programming language constructs and can be used for teaching them without being bound to a particular language. It also allows us to identify precisely some of the major differences between Java and C#.

1 Introduction

In this work the models developed in [11] and in [4] for a rigorous definition of Java and C# and their implementation on the Java Virtual Machine (JVM) resp. in the Common Language Runtime (CLR) of .NET are analyzed to extract their underlying common mathematical structure. The result is a platform-independent interpreter of high-level programming language constructs which can be instantiated to concrete interpreters of specific languages like Java, C#, C++. It is structured into components for imperative, static, object-oriented, exception handling, concurrency, pointer related and other special language features (like delegates in C#) and thus can be used in teaching to introduce step by step the basic concepts of modern programming languages and to explain the differences in their major current implementations.

The task is supported by the fact that the models in [11, 4] have been defined in terms of stepwise refined Abstract State Machines (ASMs), which

- separate the static and the dynamic parts of the semantics,
- capture the dynamics by ASM rules, one rule set for each cluster of language constructs, describing their run-time effect on the abstract program state, guided by a walk through the underlying attributed abstract syntax tree.

The stepwise refined definitions unfold in particular the following layered modules of orthogonal language features, which are also related to the historical development of programming concepts from say FORTRAN, via PASCAL and MODULA, SMALLTALK and EIFFEL, to JAVA and C#:

- imperative constructs, related to sequential control by while programs, built from statements and expressions over simple types,
- classes with so-called static class features, namely procedural abstraction with class initialization and global (module) variables,
- object-orientation with class instances, instance creation, instance methods, inheritance,
- exception handling,
- concurrency (threads),
- special features like delegates, events, etc.
- so-called unsafe features like pointers with pointer arithmetic.

This leads us to consider a sequence of sublanguages $\mathrm{L}_{\mathcal{I}} \subset \mathrm{L}_{\mathcal{C}} \subset \mathrm{L}_{\mathcal{O}} \subset \mathrm{L}_{\mathcal{E}} \subset \mathrm{L}_{\mathcal{T}} \subset \mathrm{L}_{\mathcal{D}} \subset \mathrm{L}_{\mathcal{U}}$ of a general language L, which can be instantiated to the corresponding sublanguages of Java and C# defined in [11, 4]. The interpreter $\textsc{ExecL}_{\mathcal{S}}$ of each language $\mathrm{L}_{\mathcal{S}}$ in the sequence is a conservative (purely incremental) extension of its predecessor. We show how it can be instantiated to an interpreter of $\mathrm{Java}_{\mathcal{S}}$ or $\mathrm{C\#}_{\mathcal{S}}$ by variations of well-identified state or machine components. The interpreter ExecL of the entire language L is the parallel composition of those submachines:

ExecL ≡
  ExecL$_I$
  ExecL$_C$
  ExecL$_O$
  ExecL$_E$
  ExecL$_T$
  ExecL$_D$
  ExecL$_U$

Delegates and unsafe code are peculiar features of C# and not included at all into Java, therefore we refer for the two corresponding submachines to [4]. Since the thread models of Java and C# have been analysed and compared extensivley in [11, 1, 10], we skip here to reformulate the interpreter ExecL$_T$. The static semantics of most programming languages can be captured appropriately by mainly declarative descriptions of the relevant syntactical and compile-time checked language features, e.g. typing rules, control-flow analysis, name resolution, etc.; as a consequence we concentrate our attention here on the more involved language dynamics for whose description the run-time oriented ASM framework turns out to be helpful. So we deal with the static language features in the form of conditions on the attributed abstract syntax tree, resulting from parsing and elaboration and taken as starting point of the language interpreter ExecL.

This paper does not start from scratch. We tailor the exposition for a reader who has some basic knowledge of (object-oriented) programmming. A detailed definition of ASMs and their semantics is skipped here, because ASMs can be correctly understood as pseudo-code operating over abstract (domains of) data. A textbook-style definition is available in Chapter 2 of the AsmBook [5].

2 The imperative core L$_\mathcal{I}$

In this section we define the sequential imperative core L$_\mathcal{I}$ of our general language L together with a model for its semantics. The model takes the form of an interpreter ExecL$_I$, which defines the basic machinery for the execution of the single language constructs. L$_\mathcal{I}$ provides structured while-programs consisting of to be executed statements (appearing in method bodies), which are built from to be evaluated expressions, which in turn are constructed using predefined operators over simple types. The computations of our interpreter are supposed to start with an arbitrary but fixed L-program. As explained above, syntax and compile-time matters are separated from run-time issues by assuming that the program is given as an attributed abstract syntax tree, resulting from parsing and elaboration.

2.1 Static semantics of L$_\mathcal{I}$

Expressions and statements of the sublanguage L$_\mathcal{I}$ are defined as usual by a grammar, say the one given in Fig. 1. We view this figure as defining also the corresponding ASM domains, e.g. the set *Exp* of expressions built from *Lit*erals and variable expressions using the provided operators (unary, binary, conditional) and including besides some possibly language-specific expressions the set *Sexp* of statement expressions, i.e. expressions than can be used on the top-level like an assignment to a variable expression using '`=`' (or an assignment operator from a set *Aop* or one of the typical prefix/postfix operators '`++`' or '`--`'). In this model the set *Vexp* of variable expressions (lvalues) consists of the local variables only and will be refined below.

The auxiliary sets, like *Uop* of unary operators, which one may think of as including also operators to construct type cast expressions of form '`(`' *Type* '`)`' *Exp*, vary from language to language. For example *SpecificExp*(L) may include expressions that are specific for the language L, like '`checked`' '`(`' *Exp* '`)`' and '`unchecked`' '`(`' *Exp* '`)`' in the model C#$_\mathcal{I}$ in [4, Fig.1]. In the model Java$_\mathcal{I}$ in [11, Fig.3.1] the set *SpecificExp*(L) is empty. Similarly, the set *JumpStm* of jump statements may vary from language to language; in Java$_\mathcal{I}$ it consists of '`break`' *Lab* '`;`' and '`continue`' *Lab* '`;`', in C#$_\mathcal{I}$ of '`break`' '`;`' | '`continue`' '`;`' | '`goto`' *Lab* '`;`'. *SpecificStm*(L) may contain statements that are specific to the language L, e.g. '`checked`' *Block* | '`unchecked`' *Block* for the language C#$_\mathcal{I}$. In the Java$_\mathcal{I}$ model it is empty. *Bstm* may also contain block statements for the declaration of constant expressions whose value is known at compile time, like '`const`' *Type Loc* '`=`' *Cexp* '`;`' in C#$_\mathcal{I}$.

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{Lit} \mid \mathit{Vexp} \mid \mathit{Uop}\ \mathit{Exp} \mid \mathit{Exp}\ \mathit{Bop}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{`?'}\ \mathit{Exp}\ \texttt{`:'}\ \mathit{Exp} \\
& \mid & \mathit{Sexp} \mid \mathit{SpecificExp}(L) \\
\mathit{Vexp} & ::= & \mathit{Loc} \\
\mathit{Sexp} & ::= & \mathit{Vexp}\ \texttt{`='}\ \mathit{Exp} \mid \mathit{Vexp}\ \mathit{Aop}\ \mathit{Exp} \mid \mathit{Vexp}\ \texttt{`++'} \mid \mathit{Vexp}\ \texttt{`--'} \\
\mathit{Stm} & ::= & \texttt{`;'} \mid \mathit{Sexp}\ \texttt{`;'} \mid \mathit{Lab}\ \texttt{`:'}\ \mathit{Stm} \mid \mathit{JumpStm} \\
& \mid & \texttt{`if'}\ \texttt{`('}\ \mathit{Exp}\ \texttt{`)'}\ \mathit{Stm}\ \texttt{`else'}\ \mathit{Stm} \mid \texttt{`while'}\ \texttt{`('}\ \mathit{Exp}\ \texttt{`)'}\ \mathit{Stm} \\
& \mid & \mathit{SpecificStm}(L) \mid \mathit{Block} \\
\mathit{Block} & ::= & \texttt{`\{'}\ \{\mathit{Bstm}\}\ \texttt{`\}'} \\
\mathit{Bstm} & ::= & \mathit{Type}\ \mathit{Loc}\ \texttt{`;'} \mid \mathit{Stm}
\end{array}
$$

Figure 1: Grammar of expressions and statements in $\mathrm{L}_{\mathcal{I}}$.

Not to burden the exposition with repetitions of similar arguments, we do not list here statements like `do`, `for`, `switch`, `goto case`, `goto default`, etc., which do appear in real-life languages and are treated analogously to the cases we discuss here. When referring to the set of sequences of elements from a set *Item* we write *Items*. We usually write lower case letters e to denote elements of a set E, e.g. *lit* for elements of *Lit*.

Different languages usually exhibit not only differences in syntax, but above all different notions of types with their conversion and promotion rules (subtype or compatibility definition), different type constraints on the operand and result values for the predefined operators, different syntactical constraints for expressions and statements like scoping rules, definite assignment and reachability rules, etc. As a consequence the static analysis differs, e.g. to establish the correctness of the definite assignment conditions or more generally of the type safety of well-typed programs (for Java see the type safety proof in [11, Ch.8], for C# see the proof of definite assignment correctness in [7]). Since this paper is focussed on modeling the dynamic semantics of a language, we omit here any general discussion of standard static semantics issues and come back to them only where needed to explain how the interpreter uses the attributed abstract syntax tree of a well-typed program. E.g. we will use that each expression node *exp* in the attributed syntax tree is annotated with its compile-time type $type(exp)$, that type casts are inserted in the syntax tree if necessary (reflecting type conversions at compile-time), etc.

2.2 Dynamic semantics of $\mathrm{L}_{\mathcal{I}}$

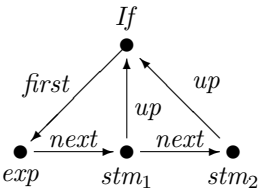
The dynamic semantics for $\mathrm{L}_{\mathcal{I}}$ describes the effect of statement execution and of expression evaluation upon the program execution state, so that the transition rule for the $\mathrm{L}_{\mathcal{I}}$ interpreter (the same for its extensions) has the form

$\textsc{ExecL}_I \equiv$
  $\textsc{ExecLExp}_I$
  $\textsc{ExecLStm}_I$

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

Syntax tree walk. To facilitate further model refinements by purely incremental extensions, the definition proceeds by walking through the abstract syntax tree, starting at pos = root-position, to compute 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. Pos is the set of positions in the abstract syntax tree. A function $label: Pos \to Label$ decorates nodes with the information which identifies the grammar rule associated to the node. For the sake of notational succinctness we use some concrete syntax from Java or C# to describe the labels, thus hiding the explicit introduction of auxiliary non-terminals. In the example the *label* of the root node is the auxiliary non-terminal *If*, identifying the grammar rule which produces the construct `if (`*exp*`)` stm_1 `else` stm_2—the 'occurrence' of which here constitutes what we are really

interested in when considering the tree. As explained below, this construct determines what we will call the *context* of the root node or of its children nodes.



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

Local variable values. The next question is what are the values computed for expressions and how they are stored as current values of local variables, namely upon executing an assignment statement or as side effect of an expression evaluation. The answer to the second question depends upon whether such values are stored directly, as for example in Java, or indirectly via an addressing mechanism, as for example in C#. To capture both possibilities we introduce two domains, namely of *Value*s and of *Adr*esses, and use the following two dynamic functions

$locals\colon Loc \to Adr$, $mem\colon Adr \to SimpleValue \cup \{Undef\}$

which can be used to assign to local variables memory addresses and to store the values there. To easen the formulation of how to instantiate our interpreter to one for Java or C# and to prepare the way for later refinements, we use a macro WriteMem(adr, t, val) to denote writing a *val*ue of given *t*ype to a given *adr*ess. For the sublanguage $L_{\mathcal{I}}$ (as for Java) the macro is only an abbreviation for $mem(adr) := val$, which will be refined in the model for $L_{\mathcal{O}}$.

One possible instance of this scheme, namely for Java, is to identify *Loc* and *Adr* so that *locals* becomes *mem*. It goes without saying and will not be mentioned any more in the sequel that a similar simplification applies to all other functions, predicates and macros introduced below in connection with handling the values stored at addresses.

Since the values we consider in $L_{\mathcal{I}}$ are of simple types, in this model the equation

$Value = SimpleValue \cup Adr$

holds, which will be refined for $L_{\mathcal{O}}$ to include references (and structs, which appear in C#). The fact that local variables have to be uniquely identified can be modeled by stipulating $Loc = Identifier \times Pos$. For the initialization of the interpreter it is natural to require that an address has been assigned to each local variable, but that the value stored there is still undefined.

- $locals(x) \in Adr$ for every variable x
- $mem(i) = Undef$ for every $i \in Adr$

Recording intermediate values. During the walk through the tree, also intermediate results of the elaboration of syntactical constructs appear, which have to be recorded somewhere, namely values of evaluated (sub-) expressions, but also possible results of the execution of statements. Statements may terminate normally, but also abruptly due to jumps (in $L_{\mathcal{I}}$) or returns from method calls (in $L_{\mathcal{C}}$) or to the occurrence of exceptions (in $L_{\mathcal{E}}$). There are many ways to keep track of such temporary items, e.g. using a stack (as do many virtual machines, see for example the Java Virtual Machine operand stack *opd* in [11, pg.140]) or replacing directly the elaborated syntactical constructs by their intermediate result (as do SOS-based formalisms, see for example the *restbody* concept in [11, pg.38]) or via some dynamic functions defined on the static syntax tree. We choose here to use a partial function to associate to nodes the *values* computed for the syntactic construct labeling each node.

$values\colon Pos \to Result$.

For $L_{\mathcal{I}}$, the range *Result* of this function contains a) *Undef*, to signal that no value is defined yet, b) simple values, resulting from expression evaluation, c) *Norm*, for normal termination of statement execution, and d) reasons for abruption of statement execution. The set *Abr* of abruptions derives here from the jump statements (see below) and will be refined in successive models to also contain statement returns and exceptions.

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

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 $L_{\mathcal{I}}$ 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 $L_{\mathcal{O}}$). 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 for YIELDINDIRECT(adr) and YIELDUPINDIRECT(adr).

A context where an address and not a value is required characterizes the context of first children of parent nodes labeled with an assignment or prefix/postfix operator. It can thus be defined as follows:

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

Notational conventions. To further reduce any notational overhead not needed by the human reader, in spelling out the ASM rules below we identify positions with the occurrences of the syntactical constructs nodes encode via their labels and those of their children. 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.[1]

For a succinct formulation of the interpreter rules we use a macro $context(pos)$ to describe the context of the currently to be handled expression or statement in the syntax tree. $context(pos)$ has to be matched against the cases appearing in the ASM rules below, choosing for the next computation step the first possible match following the textual order of the rules. 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 $uop \blacktriangleright val$ to describe the $context$ of pos, where pos is marked with the cursor ($\blacktriangleright$), 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$(Apply(uop, val))$.

Expression evaluation rules. We are now ready to define the machine EXECLEXP$_I$ for expression evaluation. We do this in a compositional way, namely proceeding expression-wise: for each group of structurally similar expressions, defined by an appropriate parameterization described in Fig. 1,[2] there is a set of rules covering each intermediate phase of their evaluation. (*SpecificExp*ressions of L are separately discussed below.) 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 (see below for the definition of *UopException* and *BopException*). When an operator has to be applied, we use a static function *Apply* to determine the value the operator provides for the given arguments. This function can be separately described, as is usually done in the language manual. Similarly for the static function defining the *ValueOfLiteral*s.

EXECLEXP$_I \equiv$ **match** $context(pos)$
  $lit \rightarrow$ YIELD$(ValueOfLiteral(lit))$
  $loc \rightarrow$ YIELDINDIRECT$(locals(loc))$

  $uop\ exp \rightarrow pos := exp$
  $uop \blacktriangleright val \rightarrow$ **if** $\neg UopException(uop, val)$ **then** YIELDUP$(Apply(uop, val))$

  $exp_1\ bop\ exp_2 \rightarrow pos := exp_1$
  $\blacktriangleright val\ bop\ exp \rightarrow pos := exp$
  $val_1\ bop \blacktriangleright val_2 \rightarrow$ **if** $\neg BopException(bop, val_1, val_2)$ **then** YIELDUP$(Apply(bop, val_1, val_2))$

  exp_0 ? exp_1 : $exp_2 \rightarrow pos := exp_0$
  $\blacktriangleright val$? exp_1 : $exp_2 \rightarrow$ **if** val **then** $pos := exp_1$ **else** $pos := exp_2$
  $True$? $\blacktriangleright val$: $exp \rightarrow$ YIELDUP(val)
  $False$? exp : $\blacktriangleright val \rightarrow$ YIELDUP(val)

[1] An identification of this kind, which is common in mathematics, has clearly to be resolved in an executable version of the model.

[2] The desired specializations can be obtained expression-wise by mere parameter expansion, a form of refinement that is easily proved to be correct.

loc `=` exp $\rightarrow pos := exp$
loc `=` `▶` val $\rightarrow \{\text{WriteMem}(locals(loc), type(loc), val),\ \text{YieldUp}(val)\}$

$vexp\ op$`=` exp $\rightarrow pos := vexp$
`▶` $adr\ op$`=` exp $\rightarrow pos := exp$
$adr\ op$`=` `▶` val $\rightarrow$ **let** $t = type(up(pos))$ **and** $v = memValue(adr, t)$ **in**
if $\neg BopException(op, v, val)$ **then**
let $w = Apply(op, v, val)$ **and** $result = Convert(t, w)$ **in**
$\{\text{WriteMem}(adr, t, result),\ \text{YieldUp}(result)\}$

$vexp\ op \rightarrow pos := vexp$ // for postfix operators $op \in \{$`++`, `--`$\}$
`▶` $adr\ op \rightarrow$ **let** $old = memValue(adr, type(pos))$ **in**
if $\neg UopException(op, old)$ **then**
$\{\text{WriteMem}(adr, type(up(pos)), Apply(op, old)),\ \text{YieldUp}(old)\}$
SpecificExp_I

Language-specific expressions. In Java$_\mathcal{I}$ the set of *SpecificExp*ressions and therefore the submachine SpecificExp_I is empty, whereas the set contains `checked` and `unchecked` expressions '`checked`' '`(`' *Exp* '`)`' | '`unchecked`' '`(`' *Exp* '`)`' in the model for C#$_\mathcal{I}$ defined in [4], because the notion of *Checked* positions serves to define when an operator exception occurs due to aritmetical *Overflow* (for which case a rule will be added in the model for L$_\mathcal{E}$). The principle is that operators for integral types only throw overflow exceptions in a checked context except for the division by zero; operators for the type `decimal` always throw overflow exceptions. By default every position is unchecked, unless explicitly declared otherwise. This is formally expressed as follows.

$$UopException(uop, val) \iff Checked(pos) \wedge Overflow(uop, val)$$
$$BopException(bop, val_1, val_2) \iff DivisonByZero(bop, val_2) \vee DecimalOverflow(bop, val_1, val_2) \vee (Checked(pos) \wedge Overflow(bop, val_1, val_2))$$

$$Checked(\alpha) \iff label(\alpha) = Checked \vee (label(\alpha) \neq Unchecked \wedge up(\alpha) \neq Undef \wedge Checked(up(\alpha)))$$

As a consequence of these definitions and of the fact that the extension by rules to handle exceptions is defined in the model extension ExecL_E, the following SpecificExp_I rules of $\text{ExecC}\#_I$ do not distinguish between checked and unchecked expression evaluation.

match $context(pos)$
`checked`(exp) $\rightarrow pos := exp$
`checked`$($`▶` $val)$ $\rightarrow \text{YieldUp}(val)$
`unchecked`(exp) $\rightarrow pos := exp$
`unchecked`$($`▶` $val)$ $\rightarrow \text{YieldUp}(val)$

Statement execution rules. The machine ExecLStm_I is defined 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 L$_\mathcal{I}$ in such a way that in the refined models it is easily extended to abruptions due to return from procedures or to exceptions. In case of a new execution of the body of a while statement, the previously computed intermediate results have to be cleared.[3] Since we formulate the model for the human reader, we 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.

[3] ClearValues is needed in the present rule formulation due to our decision to have a static function *label* and a dynamic function to record the intermediate *values* associated to nodes. In a more syntax-oriented SOS-style, as used for the Java model in [11] where a function *restbody* combines the two functions *label* and *values* into one, ClearValues results automatically from re-installing the body of the while statement as new rest program.

ExecLStm$_I$ ≡ **match** *context*(*pos*)
  `;` → Yield(*Norm*)
  exp `;` → *pos* := *exp*
  ▶*val* `;` → YieldUp(*Norm*)
  JumpStm
  `if (`*exp*`)` *stm*$_1$ `else` *stm*$_2$ → *pos* := *exp*
  `if (`▶*val*`)` *stm*$_1$ `else` *stm*$_2$ → **if** *val* **then** *pos* := *stm*$_1$ **else** *pos* := *stm*$_2$
  `if (`*True*`)` ▶*Norm* `else` *stm* → YieldUp(*Norm*)
  `if (`*False*`)` *stm* `else` ▶*Norm* → YieldUp(*Norm*)
  `while (`*exp*`)` *stm* → *pos* := *exp*
  `while (`▶*val*`)` *stm* → **if** *val* **then** *pos* := *stm* **else** YieldUp(*Norm*)
  `while (`*True*`)` ▶*Norm* → {*pos* := *up*(*pos*), ClearValues(*up*(*pos*))}
  `while (`*True*`)` ▶*Break* → YieldUp(*Norm*)
  `while (`*True*`)` ▶*Continue* → {*pos* := *up*(*pos*), ClearValues(*up*(*pos*))}
  `while (`*True*`)` ▶*abr* → YieldUp(*abr*)
  type loc `;` → Yield(*Norm*)
  lab `:` *stm* → *pos* := *stm*
  lab `:` ▶*Norm* → YieldUp(*Norm*)
  SpecificStm$_I$
  … ▶*abr* … → **if** *up*(*pos*) ≠ *Undef* ∧ *PropagatesAbr*(*up*(*pos*)) **then** YieldUp(*abr*)
  `{ }` → Yield(*Norm*)
  `{`*stm* …`}` → *pos* := *stm*
  `{` … ▶*Norm*`}` → YieldUp(*Norm*)
  `{` … ▶*Norm stm* …`}` → *pos* := *stm*
  JumpOutOfBlock
  `{` … ▶*abr* …`}` → YieldUp(*abr*)

In Java$_\mathcal{I}$ the set *JumpStm* consists of jump statements of form `break` *lab*`;` and `continue` *lab*`;`, so that the set of abruptions is defined as *Abr* = *Break*(*lab*) | *Continue*(*lab*). In C#$_\mathcal{I}$ the set *JumpStm* contains `break;` | `continue;` | `goto` *lab*`;`, so that *Abr* = *Break* | *Continue* | *Goto*(*Lab*). The differences in the scoping rules for `break;` and `continue;` statements in the two languages are reflected by differences in the corresponding interpreter rules.

JumpStm(*Java*) ≡ **match** *context*(*pos*)
  `break` *lab*`;` → Yield(*Break*(*lab*))
  `continue` *lab*`;` → Yield(*Continue*(*lab*))

JumpStm(*C#*) ≡ **match** *context*(*pos*)
  `break;` → Yield(*Break*)
  `continue;` → Yield(*Continue*)
  `goto` *lab*`;` → Yield(*Goto*(*lab*))

Due to the differences in using syntactical *lab*els to denote jump statement scopes, the definitions of how abruptions are propagated upwards differ slightly for Java$_\mathcal{I}$ and for C#$_\mathcal{I}$, though the conceptual content is the same, namely to prevent propagation at statements which are relevant for determining the abruption target. For Java$_\mathcal{I}$ we have the simple definition *PropagatesAbr*(*α*) ⟺ *label*(*α*) ≠ *lab* : *s*,[4] whereas for C#$_\mathcal{I}$ we have the following definition:

PropagatesAbr(*α*) ⟺ *label*(*α*) ∉ {*Block*, *While*, *Do*, *For*, *Switch*}

Since Java has no `goto` *lab*`;` statements, it has an empty rule JumpOutOfBlock, whereas ExecC#Stm$_I$ contains the rule

JumpOutOfBlock ≡ **match** *context*(*pos*)
  `{` … ▶*Goto*(*l*) …`}` → **let** *α* = *GotoTarget*(*first*(*up*(*pos*)), *l*)
    if *α* ≠ *Undef* **then** {*pos* := *α*, ClearValues(*up*(*pos*))}
    else YieldUp(*Goto*(*l*))

[4] We disregard here the minor difference in the formulation of *PropagatesAbr* in [11], where the arguments are not positions, but syntactical constructs or intermediate values.

where an auxiliary function is needed to compute the target of a label in a list of block statements, recursively defined as follows:

GotoTarget(α, l) =
  if *label*(α) = *Lab*(l) **then** α
  elseif *next*(α) = *Undef* **then** *Undef*
  else *GotoTarget*(*next*$(\alpha), l)$

Analogously to ExecC#Exp$_I$ also ExecC#Stm$_I$ has checked contexts and therefore the following submachine (which in ExecJavaStm$_I$ is empty):

SpecificStm$_I \equiv$ **match** *context*(*pos*)
  `checked` *block* $\rightarrow$ *pos* := *block*
  `checked` ▶*Norm* $\rightarrow$ YieldUp(*Norm*)
  `unchecked` *block* $\rightarrow$ *pos* := *block*
  `unchecked` ▶*Norm* $\rightarrow$ YieldUp(*Norm*)

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[5]:

ClearValues$(\alpha) \equiv$
  values(α) := *Undef*
  if *first*$(\alpha) \neq$ *Undef* **then** ClearValuesSeq(*first*(α))

ClearValuesSeq$(\alpha) \equiv$
  ClearValues(α)
  if *next*$(\alpha) \neq$ *Undef* **then** ClearValuesSeq(*next*(α))

3 Extension $L_{\mathcal{C}}$ of $L_{\mathcal{I}}$ by procedures (static classes)

In $L_{\mathcal{C}}$ the concept of procedures (also called subroutines or functions) is added to the purely imperative instructions of $L_{\mathcal{I}}$. We introduce the basic mechanism of procedures first for so-called static methods, which belong to classes playing the role of modules. Different languages have different mechanisms to pass the parameters to a procedure call. In Java parameters are passed by-value, whereas in C# also call-by-reference is possible. Classes[6] come with variables which play the role of global module variables, called class or static variables or fields to distinguish them from instance fields provided in $L_{\mathcal{O}}$. Usually classes come with some special methods, so-called static initializers, which are used to 'initialize' the class. The initialization concepts of different languages usually differ, in particular through different policies of when a class is initialized. In the extension ExecL$_C$ of ExecL$_I$ we illustrate these differences for Java and C#. Normally classes are also put into a hierarchy, which is used to inherit methods among classes to reduce the labor of rewriting similar code fragments. As is to be expected, different languages come with different inheritance mechanisms related to their type concepts. Since the definition of the inheritance mechanism belongs mainly to the static semantics of the language, we mention it only where it directly influences the description of the dynamics.

We present the extension as a conservative (purely incremental) refinement of the ASM ExecL$_I$, which is helpful for proving properties of the extended machine on the basis of properties of the basic machine. Conservative refinement means that we perform the following three tasks (see [2] for a general description of the ASM refinement method).

- Extension of the ASM universes and functions, or introducion of new ones, for example to reflect the grammar extensions for expressions and statements. This goes together with the addition of the appropriate constraints needed for the static analysis of the new items (like type constraints, definite assignment rules, etc.).
- Extension of some of the definitions or macros, here for example *PropagatesAbr*(α), to make them work also for the newly occurring cases.
- Addition of new ASM rules, in the present case to define the semantics of the new expressions and statements.

[5] 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 [3] for a precise definition.

[6] We disregard here the slight variations to be made for interfaces.

3.1 Static semantics of L_C

In L_C a program is a set (usually declared as 'package') of compilation units (classes and interfaces), each coming with declarations of names spaces (including a global namespace), types in the global namespace, using directives, conditions on class extension, accessability, visibility, etc. Since in this paper the focus is on dynamic semantics, we assume nested namespaces to be realized by the adoption of fully qualified names. We also do not discuss here the rules for class extensions (inheritance), for overriding of methods, for the accessibility of types and members via access modifiers like public, private, etc. This allows us to make use, for example, of a static function $body(m)$ which associates to a method its code.

The extension of the grammars for *Vexp*, *Sexp*, *Stm* and thereby of the corresponding ASM domains reflects the introduction of *Class*es with static *Field*s and static *Meth*ods, which can be called with various arguments and upon '`return`'ing may pass a computed value to the calling method. The new set *Arg* of arguments appearing here forsees that different parameters may be used. For example, Java provides value parameters (so that $Arg ::= Exp$), whereas C# allows also `ref` and `out` parameters (in which case $Arg ::= Exp \mid$ '`ref`' *Vexp* | '`out`' *Vexp*). We do not discuss here the different static constraints (on types, definite assignment, reachability, etc.) which are imposed on the new expressions and statements in different languages.[7] For the instantiation of L to Java, the set *Stm* of statements in Java_C also contains the special form `static` *stm* for static initialization blocks.

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

The presence of method calls and of to-be-initialized classes makes it necessary to introduce new universes to denote multiple methods (pairs of type and signature), the initialization status of a type (which may have additional elements in specific languages, e.g. *Unusable* for the description of class initialization errors in Java, see below) and the sequence of still active method calls (so-called frame stack of environments of method executions). One also has to extend the set *Abr* of reasons for abruption by returns from a method, with or without a computed value which has to be passed to the caller.

$$
\begin{array}{l}
\mathit{Meth} = \mathit{Type} \times \mathit{Msig} \\
\mathit{TypeState} = \mathit{Linked} \mid \mathit{InProgress} \mid \mathit{Initialized} \\
\mathit{Frame} = \mathit{Meth} \times \mathit{Pos} \times \mathit{Locals} \times \mathit{Values} \\
\quad \text{where } \mathit{Values} = (\mathit{Pos} \to \mathit{Result}) \text{ and } \mathit{Locals} = (\mathit{Loc} \to \mathit{Adr})
\end{array}
$$

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:

$$\mathit{Abr} = \ldots \mid \mathit{Return} \mid \mathit{Return}(\mathit{Value})$$

3.2 Dynamic semantics of L_C

To dynamically handle the (addresses of) static fields, the initialization state of types, the current method and the execution stack we use the following new dynamic functions:

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

To allow us to reuse without any rewriting the ExecL_I rules as part of the ExecL_C rules, we provide a separate notation $(\mathit{meth}, \mathit{pos}, \mathit{locals}, \mathit{values})$ for the current frame, instead of having it on top of the frame stack. We extend the stipulations for the initial state as follows:

- $\mathit{typeState}(c) = \mathit{Linked}$ for each class c

[7] See for example [7] for a detailed analysis of the extension of the definite assignment rules needed when allowing besides by-value parameter passing (as does Java) also call-by-reference (as does C#).

- *meth* = *EntryPoint*::`Main()` [*EntryPoint* is the main class]
- *pos* = *body*(*meth*) [The root position of the body]
- *locals* = *values* = ∅ and *frames* = []

The submachine ExecL$_C$ extends the interpreter ExecL$_I$ for L$_\mathcal{I}$ 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.

ExecL$_C$ ≡
  ExecL$_I$
  ExecLExp$_C$
  ExecLStm$_C$

Expression evaluation rules. The rules for class field evaluation in ExecLExp$_C$ are analogous to those for the evaluation of local variables in ExecLExp$_I$, except for using *globals* instead of *locals* and for the additional clause for class initialization. The rules for method calls use the macro InvokeStatic explained below, which takes care of the class initialization. The submachine ArgEval for the evaluation of sequences of arguments depends on the evaluation strategy of *L*. The definition of the submachine ParamExp for the evaluation of special parameter expressions depends on the parameter types provided by the language *L*. If *Arg* = *Exp* as in Java, this machine is empty; for the case of C#, where *Arg* ::= *Exp* | '`ref`' *Vexp* | '`out`' *Vexp*, we show below its definition.

ExecLExp$_C$ ≡ **match** *context*(*pos*)
  c.f → **if** *Initialized*(*c*) **then** YieldIndirect(*globals*(*c*::*f*)) **else** Initialize(*c*)
  c.f `=` *exp* → *pos* := *exp*
  c.f `=` ▶ *val* → **if** *Initialized*(*c*) **then**
      WriteMem(*globals*(*c*::*f*), *type*(*c*::*f*), *val*)
      YieldUp(*val*)
    else Initialize(*c*)
  c.m`(`*args*`)` → *pos* := `(`*args*`)`
  c.m▶`(`*vals*`)` → InvokeStatic(*c*::*m*, *vals*)
  ArgEval
  ParamExp

Once the arguments of a method call are computed, the macro InvokeStatic invokes the method if the initialization of its class is not triggered, otherwise it initializes the class. In both Java and C#, the initialization of a class is not triggered if the class is already initialized.[8] For methods which are not declared external or native, InvokeMethod updates the frame stack and the current frame in the expected way (the same in both Java and C#), taking care also of the initialization of local variables, which includes passing the call parameters. Consequently the definition of the macro InitLocals depends on the parameter passing mechanism of the considered language *L*, which is different for Java and for C#. Since we will also have to deal with external (native) methods – whose declaration includes an `extern` (`native`) modifier and which may be implemented using a language other than *L* – we provide here for their invocation a submachine InvokeExtern, to be defined separately depending on the class of external/native (e.g. library) methods. 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*).

InvokeStatic(*c*::*m*, *vals*) ≡
  if not *triggerInit*(*c*) **then** InvokeMethod(*c*::*m*, *vals*) **else** Initialize(*c*)
  where *triggerInit*(*c*) = ¬*Initialized*(*c*)

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**

[8] See [8] for other cases where the initialization is not triggered in C#, in particular the refinement for classes which are marked with the implementation flag *beforefieldinit* to indicate that the reference of the static method does not trigger the class initialization.

$frames := push(frames, (meth, p, locals, values))$
$meth \;\; := c::m$
$pos \;\;\;\; := body(c::m)$
$values \; := \emptyset$
INITLOCALS$(c::m, vals)$

The definition of the macro INITLOCALS for the initialization of local variables depends on the parameter passing mechanism. In Java the macro simply defines *locals* (which assumes the role of *mem* in our general model) to take as first arguments the actual values of the call parameters (the *ValueParams* for call-by-value). In C# one has to add a mechanism to pass `ref`erence parameters, including so-called `out` parameters, which can be treated as `ref` parameters except that they need not be definitely assigned until the function call returns. In the following definition of INITLOCALS for C#, 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*.[9] $paramIndex(c::m, x)$ yields the index of the formal parameter x in the signature of $c::m$.

INITLOCALS$(c::m, vals)(C\#) \equiv$
  forall $x \in LocalVars(c::m)$ **do** // addresses for local variables
    $locals(x) := new(Adr, type(x))$
  forall $x \in ValueParams(c::m)$ **do** // copy value arguments
    let $adr = new(Adr, type(x))$ **in**
      $locals(x) := adr$
      WRITEMEM$(adr, type(x), vals(paramIndex(c::m, x)))$
  forall $x \in RefParams(c::m) \cup OutParams(c::m)$ **do** // ref and out arguments
    $locals(x) := vals(paramIndex(c::m, x))$

The difference between `ref` and `out` parameters at function calls and in function bodies of C# is reflected by including as *AddressPos*itions all nodes whose parent node is labeled by `ref` or `out` and by adding corresponding definite assignment constraints (listed in [4]):

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

Therefore the following rules of PARAMEXP for C# can ignore `ref` and `out`:

PARAMEXP(C#) $\equiv$ **match** $context(pos)$
  `ref` $vexp \;\; \rightarrow pos := vexp$
  `ref` $\blacktriangleright adr \rightarrow$ YIELDUP(adr)
  `out` $vexp \;\; \rightarrow pos := vexp$
  `out` $\blacktriangleright adr \rightarrow$ YIELDUP(adr)

For the sake of illustration we provide here a definition for the submachine ARGEVAL with left-to-right evaluation strategy for sequences of arguments. The definition has 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 [12] 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.

ARGEVAL $\equiv$ **match** $context(pos)$
  `()` $\rightarrow$ YIELD$([])$
  `(`arg`,` ...`)` $\rightarrow pos := arg$
  `(`val_1`,` ...`,`$\blacktriangleright val_n$`)` $\rightarrow$ YIELDUP$([val_1, \ldots, val_n])$
  `(` ...$\blacktriangleright val$`,`arg ...`)` $\rightarrow pos := arg$

Statement execution rules. The semantics of static initialization is language dependent and is further discussed below for Java and C#. The rules for method return in EXECLSTM$_C$ trigger an abruption upon returning from a method. Via the RETURNPROPAGATION submachine defined below, an abruption *Return* or *Return*(val) due to method return is propagated to the beginning of

[9] See [9] and [5, 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.

the body of the method one is returning from. There an execution of the submachine ExitMethod is triggered, which restores the environment of the caller. This abruption propagation mechanism allows one an elegant refinement for $L_{\mathcal{E}}$, where the method exit is subject to the prior execution of so-called `finally` code which may be present in the method. The rule to YieldUp(*Norm*) does not capture falling off the method body, but yields up the result of the normal execution of the invokation of a method with void return type in an expression statement.

ExecCsharpStm$_C$ ≡ **match** *context*(*pos*)
  StaticInitializer(*L*)
  `return` *exp*`;` → *pos* := *exp*
  `return` ▶*val*`;` → YieldUp(*Return*(*val*))
  `return;` → Yield(*Return*)
  ReturnPropagation(*L*)
  ▶*Norm*`;` → YieldUp(*Norm*)

The return propagation machine for C# is simpler than (in fact part of) that for Java due to static differences (including the different use of labels) in the two languages. As mentioned above, both machines, instead of transfering the control from a return statement directly to the invoker, propagate the return abruption up to the starting point of the current method body, from where the method ist exited.

ReturnPropagation(*C#*) ≡ **match** *context*(*pos*)
  Return → **if** *pos* = *body*(*meth*) ∧ ¬*Empty*(*frames*) **then** ExitMethod(*Norm*)
  Return(*val*) → **if** *pos* = *body*(*meth*) ∧ ¬*Empty*(*frames*) **then** ExitMethod(*val*)

ReturnPropagation(*Java*) ≡ **match** *context*(*pos*)
  lab : ▶*Return* → YieldUp(*Return*)
  lab : ▶*Return*(*val*) → YieldUp(*Return*(*val*))
  ReturnPropagation(*C#*)

To complete the return propagation in Java one still has to treat the special case of a return from a class initialization method. In [11, Fig.4.5] this has been formulated as part of the StaticInitializer machine, which also realizes the condition for the semantics of Java that before initializing a class, all its superclasses have to be initialized. To stop the return propagation at the point of return from a class initialization, in the case of Java the predicate *PropagatesAbr* has to be refined to *PropagatesAbr*(α) ⟺ *label*(α) ≠ *lab* : *s* ∧ *label*(α) ≠ `static` *s*. In C# the initialization of a class does not trigger the initialization of its direct base class, so that StaticInitializer(C#) is empty.

StaticInitializer(*Java*) ≡ **match** *context*(*pos*)
  `static` *stm* →**let** *c* = *classNm*(*meth*)
    if *c* = `Object` ∨ *Initialied*(*super*(*c*)) **then** *pos* := *stm*
    else Initialize(*super*(*c*))
  `static` *Return* → YieldUp(*Return*)

The machine ExitMethod, which is the same for Java and for C# (modulo the submachine FreeLocals), restores the frame of the invoker and passes the result value (if any). Upon normal return from a static constructor (in Java called `clinit` method) 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.[10]

ExitMethod(*result*) ≡
  let (*oldMeth*, *oldPos*, *oldLocals*, *oldValues*) = *top*(*frames*) **in**
    meth := *oldMeth*
    pos := *oldPos*
    locals := *oldLocals*
    frames := *pop*(*frames*)

[10] 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.

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**
FreeMemory($locals(x), type(x)$)

For both Java and C#, a type c is considered as initialized if its static constructor has terminated normally, as is expressed by the update of $typeState(c)$ to *Initialized* in ExitMethod above. In addition, c is considered as initialized already if its static constructor has been invoked, to guarantee that during the execution of the static constructor accesses to the fields of c or invokations of methods of c do not trigger a new initialization of c. This explains the update of $typeState(c)$ to *InProgress* in the definition of Initialize and the following definition of *Initialized*:

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

To initialize a class its static constructor is invoked (denoted `<clinit>` in Java and `.cctor` in C#). All static fields of the class are initialized with their default value. The typeState of the class is updated to prevent further invokations of Initialize(c) during the execution of the static constructor of c. The macro will be further refined in $L_{\mathcal{E}}$ to account for exceptions during an initialization.

Initialize(c) $\equiv$
if $typeState(c) = Linked$ **then**
$typeState(c) := InProgress$
forall $f \in staticFields(c)$ **do**
let $t = type(c::f)$ **in** WriteMem($globals(c::f), t, defaultValue(t)$)
InvokeMethod(c::`.cctor`, [])

With respect to the execution of initializers of static class fields the ECMA standard [6, §17.4.5.1] for C# 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 definitions above for C# expresse 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:[11]

if $typeToBeInitialized \neq Undef$ **then**
Initialize($typeToBeInitialized$)
else ExecC#

4 Extension $L_{\mathcal{O}}$ of $L_{\mathcal{C}}$ by object oriented features

In this section we extend $L_{\mathcal{C}}$ to an object-oriented language $L_{\mathcal{O}}$ by adding objects for class instances, formally represented as elements of a new set *Ref* of references. The extension provides new expressions, namely for *instance* fields, instance methods and constructors and for the dynamic creation of new class instances. The inheritance mechanism we consider supports overriding and

[11] This is discussed in detail in [8]. 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.

overloading of methods and dynamic type checks and type casts. We skip the (via syntactical differences partly language-specific) treatment of arrays; their description for Java and C# can be found in [11, 4]. The interpreter ExecL$_O$ is defined as a refinement of ExecL$_C$, obtained from the latter by extending its universes, functions, macros and rules to make them work also for the new expressions.

4.1 Static semantics of L$_\mathcal{O}$

The first extension concerns the sets *Exp*, *Vexp*, *Sexp* where the new reference types appear. '`null`' denotes an empty reference, '`this`' is interpreted as the current reference. A *RefExp* is an expression of a reference type. We use '`pred`' to denote a predecessor class, in Java written '`super`' and in C# '`base`'.

Exp ::= ... | '`null`' | '`this`' | '`(`' *Type* '`)`' *Exp* | *SpecificExp*(*L*)

Vexp ::= ... | *Vexp* '`.`' *Field* | *RefExp* '`.`' *Field* | '`pred`' '`.`' *Field*

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

The specific expressions of Java$_\mathcal{I}$ and C#$_\mathcal{I}$ are extended by specific object-oriented expressions of these languages as follows:

SpecificExp(*Java*) ::= ... | *Exp* '`instanceof`' '`(`' *Type* '`)`'

SpecificExp(*C#*) ::= ... | '`typeof`' '`(`' *RetType* '`)`' | *Exp* '`is`' *Type* | *Exp* '`as`' *RefType*

Type structure. To be able to explain by variations of our interpreter ExecL the major differences between Java and C#, we need to mention here some of the major differences in the type structure underlying the two languages. For efficiency reasons C# distinguishes between value types and reference types. When a compiler encounters the declaration of a variable of value type, it directly allocates the memory to that variable, whereas for declarations of variables of reference type it creates a pointer to an object on the heap. A mediation between the two types is provided, known under the name of boxing, to convert values into references, and of an inverse operation called unboxing. At the level of L$_\mathcal{O}$, besides the new type of *Ref*erences present in both languages, C# also introduces so-called *Struct* types, a value-type restricted version of classes, to circumvent the overhead associated with class instances.

Therefore, to be prepared to instantiate our *L*-interpreter to an interpreter for both Java and C#, the domain of values of L$_\mathcal{I}$ is extended to also contain not only *Ref*erences (with a special value $null \in Ref$ to denote a null reference), as would suffice for interpreting Java$_\mathcal{O}$, but also struct values. For the case of C# we assume furthermore references to be different from addresses, i.e. $Ref \cap Adr = \emptyset$.

$Value = SimpleValue \cup Adr \cup Ref \cup Struct$.

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. $structField(val, T, f)$. 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 is assumed to satisfy the following properties, where 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 all pred(ecessor) classes of t:

- 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 $runTimeType(ref) = c$ and occupies the following addresses in *mem*:
  $\{fieldAdr(ref, f) \mid f \in instanceFields(c)\}$

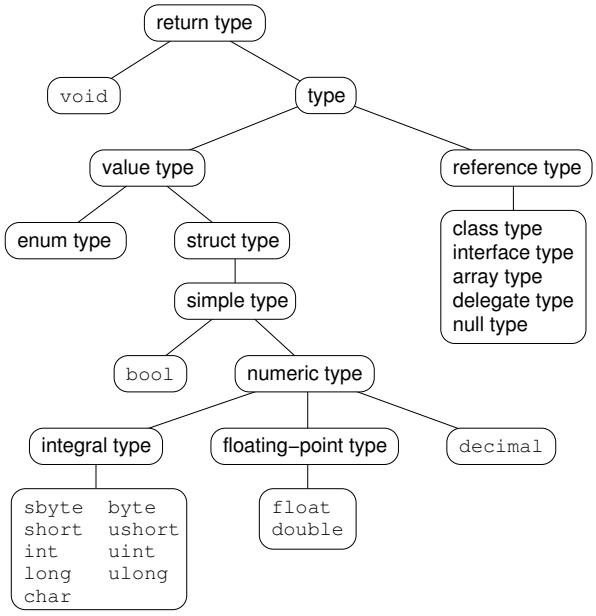


Figure 2: The classification of types of C#.

For our language L we do not specify types further. For the sake of illustration see Fig. 2 with the extended type classification of C#, where the simple types of $\mathrm{L}_{\mathcal{I}}$ became aliases for struct types.

Syntax-tree information. According to our assumption that the attributed syntax tree has exact information, for the formulation of our model we assume as result of field and method resolution that each field access has the form $T{::}f$, where f is a field declared in the type T. Similarly, each method call has the form $T{::}m\texttt{(}\mathit{args}\texttt{)}$, where m is the signature of a method declared in type T. Moreover, for the access of fields and methods via the current instance or the predecessor class we know the following:

- `pred.`f in class C has been replaced by `this.`$B{::}f$, where B is the first predecessor class of C where the field f is declared.
- `pred.`$m\texttt{(}\mathit{args}\texttt{)}$ in class C has been replaced by `this.`$B{::}M\texttt{(}\mathit{args}\texttt{)}$, where $B{::}M$ is the method signature of the method selected by the compiler (the set of applicable methods is constructed starting in the *pred class* of C).
- If f is a field, then f has been replaced by `this.`$T{::}f$, where f is declared in T.

Instance creation expressions are treated like ordinary method invocations, splitting 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` $T{::}M\texttt{(}\mathit{args}\texttt{)}$, we assume in our model without loss of generality that class instance constructors return the value of `this`.[12]

- Let T be a class type. Then `new` $T{::}M\texttt{(}\mathit{args}\texttt{)}$ is replaced by `new` $T\texttt{.}T{::}M\texttt{(}\mathit{args}\texttt{)}$.

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

- vexp `=` `new` $S{::}M\texttt{(}\mathit{args}\texttt{)}$ has been replaced by $\mathit{vexp}\texttt{.}S{::}M\texttt{(}\mathit{args}\texttt{)}$. 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` $S{::}M\texttt{(}\mathit{args}\texttt{)}$ have been replaced by $x\texttt{.}S{::}M\texttt{(}\mathit{args}\texttt{)}$, where x is a new temporary local variable of type S.

For automatic boxing we have:

[12] The result of a constructor invocation with `new` is the newly created object, which is stored in the local environment as value for `this`. Therefore one can equivalently refine the macro ExitMethod for constructors to pass the value of `this` upon returning from a constructor, see [11, pg.82].

- *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*) $\preceq T$, where $\preceq$ denotes the here not furthermore specified subtype relation (standard implicit conversion) resulting from the inheritance and the 'implements' relation between classes and interfaces.
- *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*) $\preceq T$.

4.2 Dynamic semantics of $L_{\mathcal{O}}$

Two new dynamic functions are needed to keep track of the *runTimeType*: *Ref* $\rightarrow$ *Type* of references and of the type object *typeObj*: *RetType* $\rightarrow$ *Ref* of a given type, where *RetType* ::= *Type* | '`void`'. The memory function is extended to store also references:

mem: *Adr* $\rightarrow$ *SimpleValue* $\cup$ *Ref* $\cup$ {*Undef*}.

For boxing we need a dynamic function *valueAdr*: *Ref* $\rightarrow$ *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 `this` reference is treated as first parameter and is passed by value. Therefore `this` is an element of *ValueParams*(c::m) and *paramIndex*(c::m, `this`) = 0.

For the refinement of the ExecL$_C$ transition rules it suffices to add the new machine ExecLExp$_O$ for evaluating the new expressions, since $L_{\mathcal{O}}$ introduces no new statements.

ExecL$_O$ ≡
  ExecL$_C$
  ExecLExp$_O$

In ExecLExp$_O$ the type cast rule contains three clauses concerning value types, which are needed for C# but are not present for Java. In fact for Java the first *RefType* subrule suffices, the one where both the declared type and the target type are compatible reference types and the reference is passed through. FieldExp$_O$ contains the rules for field access and assignment as needed for C#, where for Java the additional access rule for value types is not needed (and the macros for getting and setting field values are simplified correspondingly). New$_O$ differ for Java and C#, reflecting the different scheduling for the initialization, as specified below. The rules for instance method invokation are the same for Java and C# modulo different definitions for the macro Invoke and except that for C# an additional clause is needed for *StructValueInvokation*s. A struct value invocation is a method invocation on a struct value which is not stored in a variable. For such struct values a temporary storage area (called 'home') has to be created, to be passed in the invocation as value of `this`. The submachine SpecificExp$_O$ is specified below for Java and C#.

ExecExp$_O$ ≡ **match** *context*(*pos*)
  `null` → Yield(*null*)
  `this` → YieldIndirect(*locals*(`this`))
  `(`t`)` *exp* → *pos* := *exp*
  `(`t`)`$^{\blacktriangleright}$*val* → **if** *type*(*pos*) ∈ *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**
        YieldUp(*memValue*(*valueAdr*(*val*), t)) // un-box a copy of the value
    if *type*(*pos*) ∈ *ValueType* **then**
      if t = *type*(*pos*) **then** YieldUp(*val*) // compile-time identity
      if $t \in$ *RefType* **then** YieldUpBox(*type*(*pos*), *val*) // box value
  FieldExp$_O$
  New$_O$
  exp `.` T::M`(`*args*`)` → *pos* := *exp*
  $^{\blacktriangleright}$*val* `.` T::M`(`*args*`)` → *pos* := `(`*args*`)`
      if *StructValueInvocation*(*up*(*pos*)) **then**
        let *adr* = *new*(*Adr*, *type*(*pos*)) **in** // create home for struct value
          WriteMem(*adr*, *type*(*pos*), *val*)
          values(*pos*) := *adr*
  val `.` T::$M^{\blacktriangleright}$`(`*vals*`)` → Invoke(*val*, T, M, *vals*)

SpecificExp$_O$

The following definition formalizes that a struct value invocation is a method invocation on a struct value which is not stored in a variable.

$$StructValueInvocation(exp\,.\,T::M\,(args)) \iff type(exp) \in StructType \wedge exp \notin Vexp$$

The rules for instance field access and assignment in FieldExp$_O$ are equivalent for Java and C# modulo two differences. The first difference comes through the different definitions for the macro SetField explained below. The second difference consists in the fact that C# needs the struct type clause formulated below (in the second rule for field access), which is not needed for Java.[13] We use $type(exp\,.\,t::f) = type(t::f)$.

FieldExp$_O \equiv$ **match** $context(pos)$
  $exp\,.\,t::f \;\rightarrow pos := exp$
  $\blacktriangleright val\,.\,t::f \rightarrow$ **if** $type(pos) \in ValueType \wedge val \notin Adr$ **then**
    YieldUp$(structField(val, type(pos), t::f))$
  elseif $val \neq null$ **then**
    YieldUpIndirect$(fieldAdr(val, t::f))$
  $exp_1\,.\,t::f$ = $exp_2 \;\rightarrow pos := exp_1$
  $\blacktriangleright val\,.\,t::f$ = $exp \;\rightarrow pos := exp$
  $val_1\,.\,t::f$ = $\blacktriangleright val_2 \rightarrow$ **if** $val_1 \neq null$ **then**
    SetField$(val_1, t::f, val_2)$
    YieldUp(val_2)

The different schedules for the initialization of classes in Java and C# appear in the different definitions for their submachines New$_O$ and Invoke. When creating a new class instance, Java checks whether the class is initialized. If not, it initializes the class. Otherwise it does what also the machine New$_O(C\#)$ does, namely it creates a new class instance on the heap, initializing all instance fields with their default values. See below for the detailed definition of HeapInit.

New$_O(Java) \equiv$ **match** $context(pos)$
  `new` $c \rightarrow$ **if** $Initialized(c)$ **then**
    let $ref = new(Ref, c)$ **in**
      HeapInit(ref, c)
      Yield(ref)
    else Initialize(c)

New$_O(C\#) \equiv$ **match** $context(pos)$
  `new` $c \rightarrow$ **let** $ref = new(Ref, c)$ **in**
    $runTimeType(ref) := c$
    forall $f \in instanceFields(c)$ **do**
      let $adr = fieldAdr(ref, f)$ **and** $t = type(f)$ **in**
        WriteMem$(adr, t, defaultValue(t))$
    Yield(ref)

The Invoke rule for Java is paraphrased from [11, Fig.5.2]. The compile-time computable static 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.

Invoke$(val, c, m, vals)(Java) \equiv$
  let $c' =$ **case** $callKind(up(pos))$ **of**
    $Virtual \rightarrow lookup(runTimeType(val), c/m)$
    $Super \rightarrow lookup(super(classNm(meth)), c/m)$
    $Special \rightarrow c$
  InvokeMethod$(c'/m, [val]vals)$

[13] As in most parts of this paper, we disregard merely notational differences between the two models, here the fact that due to the presence of both memory addresses and values, the C#$_\mathcal{O}$ model uses the machine YieldUpIndirect$(fieldAdr(val, t::f)$ where the Java$_\mathcal{O}$ model has the simpler update YieldUp$(getField(val, t::f))$.

C# performs the initialization test only in the very moment of performing INVOKE, after the evaluation of the constructor arguments. Thus the invocation of an instance constructor of a class may trigger the class initialization (see the detailed analysis in [8]). The split into virtual and non-virtual method calls is reflected in the submachine INVOKEINSTANCE.

INVOKE(val, T, M, $vals$)($C\#$) $\equiv$
  if $InstanceCtor(M) \wedge triggerInit(T)$ **then** INITIALIZE(T)
  elseif $val \neq null$ **then** INVOKEINSTANCE($T::M$, val, $vals$)

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($T::M$, $[val] \cdot vals$)

The machines SPECIFICEXP$_O$ define the semantics of the language-specific expressions listed above, which are all related to type checking.

SPECIFICEXP$_O$($Java$) $\equiv$ **match** $context(pos)$
  exp `instanceof` t $\rightarrow$ $pos := exp$
  $\blacktriangleright val$ `instanceof` $t \rightarrow$ YIELDUP($val \neq null \wedge runTimeType(val) \preceq t$)

SPECIFICEXP$_O$($C\#$) contains SPECIFICEXP$_O$($Java$) as a submachine (modulo notational differences), namely consisting of the first and the third rule for the `is`-instruction. In addition we have rules to yield the type of an object and for type conversion between compatible types, which needs a new macro YIELDUPBOX defined below for yielding the reference of a newly created box.

SPECIFICEXP$_O$($C\#$) $\equiv$ **match** $context(pos)$
  `typeof(`t`)` $\rightarrow$ YIELD($typeObj(t)$)
  exp `is` t $\rightarrow$ $pos := exp$
  $\blacktriangleright val$ `is` $t \rightarrow$ **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 $\rightarrow$ $pos := exp$
  $\blacktriangleright val$ `as` $t \rightarrow$ **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

Memory refinement. Due to the appearance of reference (and in C# also struct) types an extension of the memory notion is needed. To model the dynamic state of objects, storage is needed for all instance variables and to record to which class an object belongs. The model for Java$_\mathcal{O}$ in [11] provides for this reason a dynamic function $heap$: $Ref \rightarrow Heap$ to record every class instance together with the values of its fields. $Heap$ can be considered as an abstract set of elements of form $Object(t, fields)$, where $fields$ is a map associating a value to each field in $instanceFields(t)$. One can then define two simple macros SETFIELD and GETFIELD to manipulate references on this abstract heap as follows (where $\oplus$ denotes adding a new (argument,value)-pair to a function, or overwriting an existing value by a new one):

GETFIELD(ref, f)($Java$) $\equiv$ **case** $heap(ref)$ **of**
  `Object`$(t, fields) \rightarrow fields(f)$
SETFIELD(ref, f, val)($Java$) $\equiv$ **let** `Object`$(t, fields) = heap(ref)$ **in**
  $heap(ref) :=$ `Object`$(t, fields \ \oplus \ \{(f, val)\})$

For modeling C#$_{\mathcal{O}}$ a further refinement of both reading from and writing to memory is needed, due to the presence of struct types. The notion of reading from the memory is refined by extending the simple equation $memValue(adr, t) = mem(adr)$ of C#$_{\mathcal{I}}$ to fit also struct types, in addition to reference 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** $\{f \mapsto getField(adr, f) \mid f \in instanceFields(t)\}$

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

Similarly, writing to memory is refined from $\text{WriteMem}(adr, t, val) \equiv mem(adr) := val$ in C#$_{\mathcal{I}}$, recursively together with SetField along the given struct type:

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

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

The notion of *AddressPos* from C#$_{\mathcal{C}}$ is refined to include also lvalue nodes of *StructType*, so that address positions are of the following form: `ref` □, `out` □, □`++`, □`--`, □ *op*`=` *exp*, □`.`*f*, □`.`*m*`(`*args*`)`.

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

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

$\text{YieldUpBox}(t, val) \equiv$ **let** $ref = new(Ref)$ **and** $adr = new(Adr, t)$ **in**
  $runTimeType(ref) := t$
  $valueAdr(ref) := adr$
  $\text{WriteMem}(adr, t, val)$
  $\text{YieldUp}(ref)$

5 Extension L$_{\mathcal{E}}$ of L$_{\mathcal{O}}$ by exceptions

L$_{\mathcal{E}}$ extends L$_{\mathcal{O}}$ with exceptions, designed to provide support for recovering from abnormal situations, separating normal program code from exception handling code. When an *L*-program violates certain semantic constraints at run-time, the interpreter signals this as an *exception*. The control is transferred, from the point where the exception occurred, to a point that can be specified by the programmer. An exception is said to be *thrown* from the point where it occurred, and it is said to be *caught* at the point to which control is transferred. The model for L$_{\mathcal{E}}$ makes explicit how jump statements from L$_{\mathcal{I}}$, return statements from L$_{\mathcal{C}}$ and the initialization of classes from L$_{\mathcal{O}}$ interact with catching and handling exceptions.

Technically, exceptions are represented as objects of predefined system exception classes (in Java *Throwable* and in C# *System.Exception*) 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. We consider *run-time exceptions*, which correspond to invalid operations violating the semantic constraints of the language (like an attempt to divide by zero or to index an array outside its bounds) and *user-defined exceptions*. We do not treat *errors* which are failures detected by the underlying virtual machine machine (JVM or CLR).

5.1 Static semantics of $\mathrm{L}_{\mathcal{E}}$

For the refinement of $\textsc{ExecL}_O$ by exceptions, it suffices to extend the static semantics and to add the new rules for exception handling. The set of statements is extended by throw and try-catch-finally statements as defined byt he folowing grammar (where the '`throw`' '`;`' statement without expression and so-called *general catch clauses* of form `catch` *block*) are present only in C#, not in Java):

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

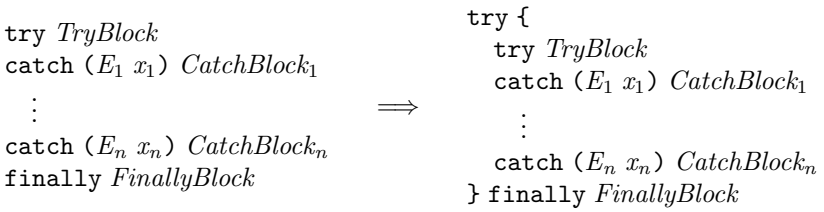
Various static constraints are imposed on try-catch-finally statements in L-programs, like the following ones we need below to explain the correctness of the transition rules below:

- every try-catch-finally statement contains at least one *catch clause*, *general catch clause*, or *finally block*
- the exception classes in a *Catch* clause appear there in a non-decreasing type order, more precisely for every try-catch statement `try` *block* `catch (`E_1 x_1`)` $\mathit{block}_1 \ldots$ `catch (`E_n x_n`)` block_n holds: $i < j \Longrightarrow E_j \not\preceq E_i$

Some static constraints on try-catch-finally statements are language-specific. We only list the following three specific constraints of C# which will be needed to justify the correctness of the transition rules below.

- 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

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 and that try-catch-finally statements have been reduced to try-catch and try-finally statements, e.g. as follows:



Since throwing an exception completes the computation of an expression or a statement abruptly, we introduce into the model a new type of reasons of abruptions and type states, namely references $\mathit{Exc}(\mathit{Ref})$ to an exception object. Due to the presence of `throw` statements without expression in C#, also a stack of references is needed to record exceptions which are to be re-thrown.

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

5.2 Dynamic semantics of $\mathrm{L}_{\mathcal{E}}$

The transition rules for $\textsc{ExecL}_E$ are defined by adding two submachines to $\textsc{ExecL}_O$. 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.

$\textsc{ExecL}_E \equiv$
  $\textsc{ExecL}_O$
  $\textsc{ExecLExp}_E$
  $\textsc{ExecLStm}_E$

Expression evaluation rules. $\textsc{ExecLExp}_E$ contains rules for each of the forms of run-time exceptions forseen by L. We give here some 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 and reference expressions. The notion of FailUp we use is supposed to execute the code `throw new E()` at the parent position, which allocates a new object for the exception and throws the exception (whereby the execution of the corresponding finally code starts, if there is some, together with the search for the appropriate exception handler. Therefore one can 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.

In the formulation of the following rules we use the exception class names from C#, which are often slightly different from those of Java. A binary expression throws an arithmetical exception, if the operator is an integer division or remainder operator and the right operand is 0. The overflow-clause for unary or binary operators is expressed using the above defined *Checked* predicate from C#.

ExecLExp$_E$ ≡ **match** *context*(*pos*)
  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`)
  uop ▶ *val* → **if** *Checked*(*pos*) ∧ *Overflow*(*uop*, *val*) **then** FailUp(`OverflowException`)
  CastExceptions
  NullRefExceptions

In Java, a reference type *cast expression* throws a `ClassCastException`, if the value of the direct subexpression is neither *null* nor *compatible* with the required type. This is the first clause in the rule below which is formulated for C#, where additional clauses appear due to value types.

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*) ∈ *SimpleType* ∧ *t* ∈ *SimpleType* ∧ *Checked*(*pos*) ∧ *Overflow*(*t*, *val*)
      then FailUp(`OverflowException`)

An instance target expression throws a `NullReferenceException`, if the operand is *null*.

NullRefExceptions ≡ **match** *context*(*pos*)
  ▶ *ref* `.` *t*`::`*f* → **if** *ref* = *Null* **then** FailUp(`NullReferenceException`)
  ref `.` *t*`::`*f* `=` ▶ *val* → **if** *ref* = *Null* **then** FailUp(`NullReferenceException`)
  ref `.` *T*`::`*M* `(` ▶ *vals* `)` → **if** *ref* = *Null* **then** FailUp(`NullReferenceException`)

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. We formulate the machine below for C# and then explain its simplification for the case of Java (essentially obtainable by deleting every exception-stack-related feature).

When the exception value *ref* of a `throw` statement has been computed, and if it turns out to be *null*, a `NullReferenceException` is reported to the enclosing phrase using FailUp, which allocates a new object for the exception and throws the exception. If the exception value *ref* of a `throw` statement is not *null*, the abruption *Exc*(*ref*) is passed up to the (position of the) `throw` statement, thereby abrupting the control flow with the computed exception as reason. The semantics of the parameterless `throw;` statement is explained as throwing the top element of the exception stack *excStack*.

Upon normal completion of a `try` statement, the machine passes the control to the parent statement, whereas upon abrupted completion the machine attempts to catch the exception by one of the `catch` clauses. The catching condition is the compatibility of the class of the exception with one of the catcher classes. If the catching fails, the exception is passed to the parent statement, as is every other abruption which was propagated up from within the `try` statement. Otherwise the

control is passed to the execution of the relevant `catch` statement, recording the current exception object in the corresponding local variable and pushing it on the exception stack (thus recording the last exception in case it has to be re-thrown). Upon completion of this `catch` statement, the machine passes the control up and pops the current exception object from the exception stack—the result of this statement execution may be normal or abrupted, in the latter case the new exception is passed up to the parent statement. No special rules are needed for general catch clauses '`catch` *block*' in try-catch statements, due to their compile-time transformation mentioned above.

For a `finally` statement, upon normal or abrupted completion of the first direct substatement, the control is passed to the execution of the second direct substatement, the `finally` statement proper. Upon normal completion of this statement, the control is passed up, together with the possible reason of abruption, the one which was present when the execution of `finally` statement proper was started, and which in this case has to be resumed after execution of the `finally` statement proper. However, should the execution of this `finally` statement proper abrupt, then this new abruption is passed to the parent statement. The constraints listed above for C# restrict the possibilities for exiting a finally block to normal completion or triggering an exception, whereas in Java also other abruptions may occur here.

In Java there is an additional rule for passing exceptions when they have been propagated to the position directly following a label, namely:

$lab:\ ▶Exc(ref) \rightarrow \textsc{YieldUp}(Exc(ref))$

If the attempt to catch a thrown exception in the current method fails, the exception is propagated to the caller using the submachine explained below.

$\textsc{ExecCsharpStm}_E \equiv$ **match** $context(pos)$
- `throw` exp`;` $\rightarrow pos := exp$
- `throw` $▶ref$`;` $\rightarrow$ **if** $ref = Null$ **then** $\textsc{FailUp}$(`NullReferenceException`) **else** $\textsc{YieldUp}(Exc(ref))$
- `throw;` $\rightarrow \textsc{Yield}(Exc(top(excStack)))$
- `try` $block$ `catch (`$E\ x$`)` $stm\ \ldots \rightarrow pos := block$
- `try` $▶Norm$ `catch (`$E\ x$`)` $stm\ \ldots \rightarrow \textsc{YieldUp}(Norm)$
- `try` $▶Exc(ref)$ `catch(`$E_1\ x_1$`)` $stm_1\ \ldots$ `catch(`$E_n\ x_n$`)` $stm_n \rightarrow$
  - **if** $\exists i \in [1\,..\,n]\ runTimeType(ref) \preceq E_i$ **then**
    - **let** $j = \min\{i \in [1\,..\,n] \mid runTimeType(ref) \preceq E_i\}$ **in**
      - $pos := stm_j$
      - $excStack := push(ref, excStack)$
      - $\textsc{WriteMem}(locals(x_j), \texttt{object}, ref)$
  - **else** $\textsc{YieldUp}(Exc(ref))$
- `try` $▶abr$ `catch(`$E_1\ x_1$`)` $stm_1\ \ldots$ `catch(`$E_n\ x_n$`)` $stm_n \rightarrow \textsc{YieldUp}(abr)$
- `try` $Exc(ref)\ \ldots$ `catch(...)` $▶res\ \ldots \rightarrow \{excStack := pop(excStack),\ \textsc{YieldUp}(res)\}$
- `try` $tryBlock$ `finally` $finallyBlock \rightarrow pos := tryBlock$
- `try` $▶res$ `finally` $finallyBlock \rightarrow pos := finallyBlock$
- `try` res `finally` $▶Norm \rightarrow \textsc{YieldUp}(res)$
- `try` res `finally` $▶Exc(ref) \rightarrow \textsc{YieldUp}(Exc(ref))$
- $\textsc{PropagateToCaller}(Exc(ref))$

If the attempt to catch a thrown exception in the current method fails, the exception is passed by $\textsc{PropagateToCaller}$(Exc(ref)) to the invoker of this method (if there is some), to continue the search for an exception handler there. In case an exception happened in the static constructor of a type, in C# its type state is set to that exception to prevent its re-initialization and instead to re-throw the old exception object, performed by an extension of $\textsc{Initialize}$(c) by the clause **if** $typeState(c) = Exc(ref)$ **then** $\textsc{Yield}(Exc(ref))$. In Java, the corresponding type becomes *Unusable*, meaning that its initialization is not possible, which is realized by the additional $\textsc{Initialize}$(c)-clause **if** $typeState(c) = Unusable$ **then** $\textsc{Fail}$(`NoClassDefFoundErr`).

$\textsc{PropagateToCaller}(Exc(ref)) \equiv$ **match** $context(pos)$
- $Exc(ref) \rightarrow$ **if** $pos = body(meth) \wedge \neg Empty(frames)$ **then**
  - **if** $StaticCtor(meth)$ **then** $typeState(type(meth)) := Exc(ref)$
  - $\textsc{ExitMethod}(Exc(ref))$

The model ExecJavaStm$_E$ in [11, Fig.6.2] has the following rule for uncaught exceptions in class initializers, which is inserted before the general rule PropagateToCaller(Exc(ref)). For this case Java specifies the following strategy. If during the execution of the body of a static initializer an exception is thrown, and if this is not an `Error` or one of its subclasses, `ExceptionInInitializerError` is thrown. If the exception is compatible with `Error`, then the exception is rethrown in the directly preceding method on the frame stack.

match *context*(*pos*)
  `static` *Exc*(*ref*) → **if** *runTimeType*(*ref*) $\preceq_h$ `Error` **then** YieldUp(*Exc*(*ref*))
    else FailUp(`ExceptionInInitializerError`)

An alternative treatment appears in the model ExecC#Stm$_E$ in [4] where unhandled exceptions in a static constructor are wrapped into a `TypeInitializationException` by translating `static` T`()` `{` *BlockStatements* `}` into

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

The interpreter for Java$_\mathcal{E}$ needs also a refinement of the definition of propagation of abruptions, to the effect that `try` statements suspend jump and return abruptions for execution of relevant `finally` code. For C# this is not needed due to the constraints cited above for finally code in C#. As explained above, after the execution of this `finally` code, that abruption will be resumed (unless during the `finally` code a new abruption did occur which cancels the original one).

PropagatesAbr(α) $\iff$
  label(α) $\neq$ *lab* : s $\wedge$ *label*(α) $\neq$ `static` s $\wedge$ *label*(α) $\neq$ `try` ... $\wedge$ *label*(α) $\neq$ s_1 `finally` s_2

6 Conclusion

We have defined hierarchically structured components of an interpreter for a general object-oriented programmming language. In doing this we have identified a certain number of static and dynamic parameters and have shown that they can be instantiated to obtain an interpreter for Java or C#. As a by-product this pinpoints in a precise and explicit way the main semantical differences between the two languages. The work confirms the idea that one can use ASMs to define in an accurate way appropriate abstractions to support the development of *precise patterns for fundamental computational concepts* in the fields of hardware and software, reusable for design-for-change and useful for communicating and teaching design skills.

Acknowledgement. We gratefully acknowledge partial support of this work by a Microsoft grant within the ROTOR project during the year 2002-2003.

References

[1] V. Awhad and C. Wallace. A unified formal specification and analysis of the new Java memory models. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 166–185. Springer-Verlag, 2003.

[2] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.

[3] E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer-Verlag, 2003.

[4] E. Börger, N. G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004.

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

[6] C# Language Specification. Standard ECMA–334, 2001.
http://www.ecma-international.org/publications/standards/ECMA-334.HTM.

[7] N. G. Fruja. The correctness of the definite assignment analysis in C#. Technical report, Computer Science Department, ETH Zürich, 2004.

[8] N. G. Fruja. Specification and implementation problems for C#. In B. Thalheim and W. Zimmermann, editors, *Abstract State Machines 2004*, LNCS. Springer, 2004.

[9] N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 244–262. Springer-Verlag, 2003.

[10] R. F. Stärk and E. Börger. An ASM model for C# threads. In B. Thalheim and W. Zimmermann, editors, *Abstract State Machines 2004*, LNCS. Springer, 2004.

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

[12] W. Zimmermann and A. Dold. A framework for modeling the semantics of expression evaluation with Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 391–406. Springer-Verlag, 2003.