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

Egon Börger, Georgian Fruja, Vincenzo Gervasi, Robert F. Stärk

October 10, 2003

Abstract

We propose a structured mathematical definition of the semantics of C# programs to provide a platform-independent view of the language for the C# programmer, which can also be used for a precise analysis of the ECMA [7] standard of the language. 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 [15] and to provide for implementors the possibility to check their basic design decisions against an accurate high-level model. In particular we will highlight some of the major differences between the ECMA standard and the implementation of the language in .NET.

1 Introduction

In this work the method developed in [15] for a rigorous definition and analysis of Java and its implementation on the Java Virtual Machine (JVM) is applied to formalize the semantics of C#. We provide a 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 [7] and in [10] and clarifies a certain number of semantically relevant issues which are not handled by that standard. We also consulted the Microsoft Press books [1, 11, 12]. A series of bugs and gaps in the ECMA standard for C# and in .NET and incoherences between the two were detected during our attempt to build a consistent and complete yet abstract model for the language. To support the experimentation with the model a project has been started to refine the model developed here to .NET-executable AsmL code [8], similarly to the AsmGofer refinement developed by Joachim Schmid [13, 14] for the Java and JVM models in [15].

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 State Machine (ASM) whose notion of run provides a transparent way to reflect those run-time-related features, which are encountered upon executing a given C# program. The use of ASMs[1] also allows us to specify the static and the dynamic parts of the semantics separately. 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 annotated 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 [15] 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 class initialization and global (module) variables), object-orientation (with class instances, instance methods, inheritance), exception handling, delegates together with events (including for convenience here also properties, indexers, attributes), concurrency, unsafe code with

[1] more precisely the classification of abstract states into a static and a dynamic part

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

To keep the definition of the models succinct, we avoid tedious and routine repetitions concerning language constructs which can be reduced in well-known ways to the core constructs in our models. Since the handling of truly concurrent threads, not limited to interleaving or similar simple scheduling techniques, is closely related to the underlying memory model whose description goes much beyond this paper, the submodel C$\#_{\mathcal{T}}$ and its further analysis is postponed to a forthcoming separate paper [6].

Since by and large one can correctly understand an ASM as pseudo-code operating over abstract data, we skip a detailed definition of ASMs which is available in textbook form in Chapter 2 of the AsmBook [5]. The basic framework of the model is introduced together with the model for the imperative kernel C$\#_{\mathcal{I}}$ of the language which is then refined by successive extensions to the full model. Since our paper is not a tutorial or manual on C#, we restrict our explanations here to features a reader will appreciate who is already knowledgeable about the basic concepts of object-oriented programming. In a technical report [3] also the remaining details which are skipped in this paper are spelt out completely, together with further explanations and examples.

2 The imperative core C$\#_{\mathcal{I}}$

In this section we define the model for C$\#_{\mathcal{I}}$, which defines the basic machinery of the ASM model for C# and 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 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$\#_{\mathcal{I}}$, we formulate what has to be said about the static semantics.

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 on the top-level like 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 [7, §13.1] and of binary numeric promotions in [7, §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 2.2 (Implicit conversion [7, §13.1]) There is 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 simple 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.

Exp ::= *Lit* | *Vexp* | *Uop Exp* | *Exp Bop Exp* | *Exp* ‘`?`’ *Exp* ‘`:`’ *Exp* | ‘`(`’ *Type* ‘`)`’ *Exp*
  | *Sexp* | ‘`(`’ *Exp* ‘`)`’ | ‘`checked`’ ‘`(`’ *Exp* ‘`)`’ | ‘`unchecked`’ ‘`(`’ *Exp* ‘`)`’

Vexp ::= *Loc*

Sexp ::= *Vexp* ‘`=`’ *Exp* | *Vexp Aop Exp* | *Vexp* ‘`++`’ | *Vexp* ‘`--`’

Uop ::= ‘`+`’ | ‘`-`’ | ‘`!`’ | ‘`~`’

Bop ::= ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`+`’ | ‘`-`’ | ‘`<<`’ | ‘`>>`’ | ‘`<`’ | ‘`>`’ | ‘`<=`’ | ‘`>=`’ | ‘`==`’ | ‘`!=`’ | ‘`&`’ | ‘`^`’ | ‘`|`’

Aop ::= ‘`*=`’ | ‘`/=`’ | ‘`%=`’ | ‘`+=`’ | ‘`-=`’ | ‘`<<=`’ | ‘`>>=`’ | ‘`&=`’ | ‘`^=`’ | ‘`|=`’

Stm ::= ‘`;`’ | *Sexp* ‘`;`’ | ‘`break`’ ‘`;`’ | ‘`continue`’ ‘`;`’ | ‘`goto`’ *Lab* ‘`;`’
  | ‘`if`’ ‘`(`’ *Exp* ‘`)`’ *Stm* ‘`else`’ *Stm*
  | ‘`while`’ ‘`(`’ *Exp* ‘`)`’ *Stm* | ‘`do`’ *Stm* ‘`while`’ ‘`(`’ *Exp* ‘`)`’
  | ‘`for`’ ‘`(`’ [*Sexps*] ‘`;`’ [*Exp*] ‘`;`’ [*Sexps*] ‘`)`’ *Stm*
  | ‘`switch`’ ‘`(`’ *Exp* ‘`)`’ ‘`{`’ {*Case* {*Case*} *Bstm* {*Bstm*}} ‘`}`’
  | ‘`goto`’ ‘`case`’ *Cexp* ‘`;`’ | ‘`goto`’ ‘`default`’ ‘`;`’
  | ‘`checked`’ *Block* | ‘`unchecked`’ *Block* | *Block*

Sexps ::= *Sexp* {‘`,`’ *Sexp*}

Case ::= ‘`case`’ *Cexp* ‘`:`’ | ‘`default`’ ‘`:`’

Block ::= ‘`{`’ {*Bstm*} ‘`}`’

Bstm ::= *Type Loc* ‘`;`’ | ‘`const`’ *Type Loc* ‘`=`’ *Cexp* ‘`;`’ | *Lab* ‘`:`’ *Stm* | *Stm*

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

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 [7] for the predefined (arithmetical, relational, bit and boolean logical) operators and the expression types. We assume that each expression node *exp* in the abstract syntax tree is annotated with 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 2.3 (Binary numeric promotion [7, §14.2.6]) 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 [7], 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.

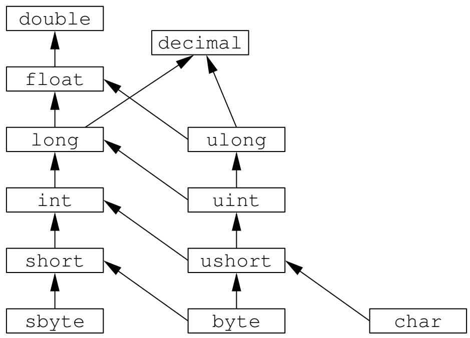


Figure 2: The simple types of C$\#_{\mathcal{I}}$.

To simplify the exposition of our model we assume some standard syntactical reductions as indicated in the following table:

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 (`$type$ loc `=` exp`;` $test$`;` $step$`)` stm	`{` $type$ loc`; for (`loc `=` exp`;` $test$`;` $step$`)` stm `}`

During the static program analysis where the compiler has to verify that the given program is well-typed, predicates *reachable* and *normal* with the following intended meaning are computed for statements, using the type information contained in the annotated syntax tree as result of parsing and elaboration:

$$\begin{array}{lll} reachable(stm) & \Longleftrightarrow & stm \text{ can be reached} \\ normal(stm) & \Longleftrightarrow & stm \text{ can terminate normally } \Longleftrightarrow \text{ the end point of } stm \text{ can be reached} \end{array}$$

For the correctness of our model we have to guarantee the following two program properties: a) during the program execution, only *reachable* positions are reached, b) normal termination happens only in *normal* positions. These two properties, which are undecidable, are guaranteed by checking two sufficient conditions via so-called reachability rules, which can be inductively defined for C$\#_{\mathcal{I}}$ as follows (similarly for `do`, `for`, `switch`). For constant boolean expressions in conditional and while statements we assume that they are replaced in the abstract syntax tree by `true` or `false`.

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` $s_2)$ and $e \neq$ `false`	$\Longrightarrow$	$reachable(s_1)$
$reachable($`if (`e`)` s_1 `else` $s_2)$ and $e \neq$ `true`	$\Longrightarrow$	$reachable(s_2)$
$normal(s_1)$ or $normal(s_2)$	$\Longrightarrow$	$normal($`if (`e`)` s_1 `else` $s_2)$
$reachable($`while (`e`)` $s)$ and $e \neq$ `false`	$\Longrightarrow$	$reachable(s)$
$reachable($`while (`e`)` $s)$ and $e \neq$ `true`	$\Longrightarrow$	$normal($`while (`e`)` $s)$
$reachable($`break;`$)$ in s	$\Longrightarrow$	$normal($`while (`e`)` $s)$

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 of the bottom of the code array.

We also have to reflect in the model 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

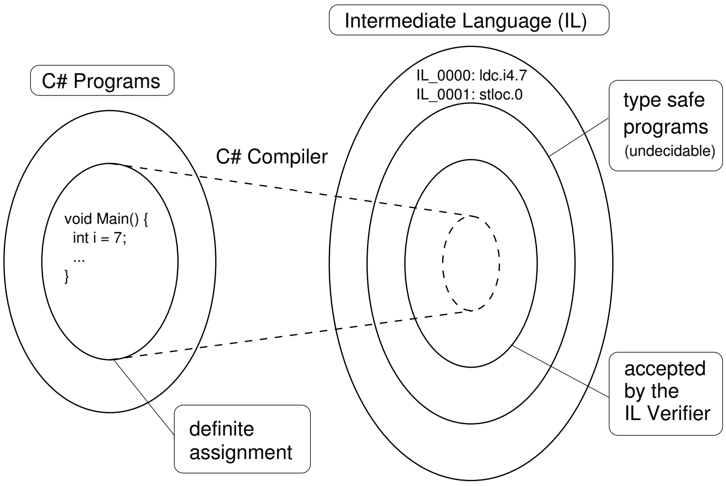


Figure 3: Definite assignment and IL verification.

are evaluated at run-time to values that are *compatible* with their compile-time types. The desired consequence of the type safety of a program is that at run-time 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[2].

To guarantee the type safety, which is an undecidable program property, the compiler checks a sufficient condition computing predicates $\mathit{before}, \mathit{after}$ (for occurrences of statements and expressions in a function body) and $\mathit{true}, \mathit{false}$ (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 the reality. For unknown reasons Microsoft has decided that in verified IL (intermediate language) code local variables are initialized by the run-time system with zero values. Hence, also 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.

$$\begin{array}{lll}
x \in \mathit{before}(\mathit{item}) & \iff & x \text{ is definitely assigned } \mathit{before} \text{ the elaboration of } \mathit{item} \\
x \in \mathit{after}(\mathit{item}) & \iff & x \text{ is definitely assigned } \mathit{after} \text{ normal elaboration of } \mathit{item} \\
x \in \mathit{true}(\mathit{exp}) & \iff & x \text{ is definitely assigned } \mathit{after}\ \mathit{exp} \text{ evaluates to } \mathit{true} \\
x \in \mathit{false}(\mathit{exp}) & \iff & x \text{ is definitely assigned } \mathit{after}\ \mathit{exp} \text{ evaluates to } \mathit{false}
\end{array}$$

We make the definite assignment rules of [7, §12.3.3] precise by the following constraints, where $\mathit{vars}(\mathit{stm}) = \{x \mid \mathit{stm} \text{ is in the scope of } x\}$.

- If s is a function body, then $\mathit{before}(s) = \emptyset$
- $\mathit{after}(\texttt{;}) = \mathit{before}(\texttt{;})$
- $\mathit{before}(\mathit{exp}) = \mathit{before}(\mathit{exp}\texttt{;})$
- $\mathit{after}(\mathit{exp}\texttt{;}) = \mathit{after}(\mathit{exp})$
- $\mathit{after}(\texttt{break;}) = \mathit{vars}(\texttt{break;})$
- $\mathit{after}(\texttt{continue;}) = \mathit{vars}(\texttt{continue;})$
- $\mathit{after}(\texttt{goto}\ l\texttt{;}) = \mathit{vars}(\texttt{goto}\ l\texttt{;})$

For blocks $\mathit{stm} = \texttt{\{}s_1 \ldots s_n\texttt{\}}$ the constraints are as follows:

[2]For example the following invariants can be proved to hold at run-time: a) $\mathit{before}(\mathit{pos}) \subseteq \mathit{Defined}$ where $\mathit{Defined} = \{x \in \mathit{Loc} \mid \mathit{mem}(\mathit{locals}(x)) \neq \mathit{Undef}\}$, b) $\mathit{after}(\mathit{pos}) \subseteq \mathit{Defined}$ if $\mathit{values}(\mathit{pos}) = \mathit{Norm}$ or $\mathit{values}(\mathit{pos}) \in \mathit{Value}$. Specifically for boolean expressions holds $\mathit{true}(\mathit{pos}) \subseteq \mathit{Defined}$ if $\mathit{values}(\mathit{pos}) = \mathit{True}$, the same for false. Such proofs can be carried out using the pattern developed in [15, Ch.8] for proving that Java is type safe.

- $before(s_1) = before(stm)$
- $after(stm) = after(s_n)$
- $before(s_{i+1}) = after(s_i) \cap goto(s_{i+1})$ where $goto(l\!:\!s) = \bigcap\{before(\texttt{goto}\ l\texttt{;}) \mid \texttt{goto}\ l$ is reachable in $stm\}$ and $goto(s) = vars(s)$ if s is not a labeled statement

For $stm = \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2$ one has to require:

- $before(e) = before(stm)$
- $before(s_1) = true(e)$
- $before(s_2) = false(e)$
- $after(stm) = after(s_1) \cap after(s_2)$

Constraints for while statements $stm = \texttt{while (}e\texttt{)}\ s$:

- $before(e) = before(stm)$
- $before(s) = true(e)$
- $after(stm) = false(e) \cap break(s)$ where $break(s) = \bigcap\{before(\texttt{break;}) \mid \texttt{break;}$ reachable in $s\}$

For boolean expressions we have the following general constraints.
If $exp \in \{\texttt{true},\ \texttt{false},\ \texttt{!}e,\ e_1\ \texttt{\&\&}\ e_2,\ e_1\ \texttt{||}\ e_2,\ e_0\ \texttt{?}\ e_1\ \texttt{:}\ e_2\}$, then

- $after(exp) = true(exp) \cap false(exp)$

otherwise

- $true(exp) = after(exp)$
- $false(exp) = after(exp)$

In addition for specific boolean expressions, the following specific constraints are imposed for the eager ('short-circuit') evaluation of Boolean expressions.

- $true(\texttt{true}) = before(\texttt{true})$
- $false(\texttt{true}) = vars(\texttt{true})$
- $true(\texttt{false}) = vars(\texttt{true})$
- $false(\texttt{false}) = before(\texttt{false})$

For negations $exp = \texttt{!}e$:

- $before(e) = before(exp)$
- $true(exp) = false(e)$
- $false(exp) = true(e)$

For conjunctions $exp = (e_1\ \texttt{\&\&}\ e_2)$:

- $before(e_1) = before(exp)$
- $before(e_2) = true(e_1)$
- $true(exp) = true(e_2)$
- $false(exp) = false(e_1) \cap false(e_2)$

For disjunctions $exp = (e_1\ \texttt{||}\ e_2)$:

- $before(e_1) = before(exp)$
- $before(e_2) = false(e_1)$
- $true(exp) = true(e_1) \cap true(e_2)$
- $false(exp) = false(e_2)$

If $exp = (e_0\ \texttt{?}\ e_1\ \texttt{:}\ e_2)$, then

- $before(e_0) = before(exp)$
- $before(e_1) = true(e_0)$
- $before(e_2) = false(e_0)$
- $true(exp) = true(e_1) \cap true(e_2)$
- $false(exp) = false(e_1) \cap false(e_2)$

For general expressions the constraints for definite assignment are as follows.

- $loc \in before(loc)$
- $after(loc) = before(loc)$
- $after(lit) = before(lit)$

For simple assignments $exp = (loc\,\texttt{=}\,e)$ we have

- $before(e) = before(exp)$
- $after(exp) = after(e) \cup \{loc\}$

For compound assignments $exp = (loc\ op\texttt{=}\ e)$ we have

- $loc \in before(exp)$
- $before(e) = before(exp)$
- $after(exp) = after(e)$

If $exp = (e_0\ \texttt{?}\ e_1\ \texttt{:}\ e_2)$, then

- $before(e_0) = before(exp)$
- $before(e_1) = true(e_0)$
- $before(e_2) = false(e_0)$
- $after(exp) = after(e_1) \cap after(e_2)$

In all other cases, if exp is an expression with *direct subexpressions* $e_1, e_2, \ldots, e_n$, then the left-to-right evaluation scheme yields

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

Due to the goto statement the above constraints specify the sets of variables that have to be considered as definitely assigned not in a unique way. For blocks without goto statements, however, it can be shown that the *before* set determines the *after* set in a unique way.

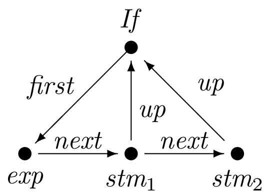
2.4 Transition rules for C$\#_\mathcal{I}$

The dynamic semantics for C$\#_\mathcal{I}$ describes the effect of statement execution and expression evaluation upon the program state, so that the transition rule for C$\#_\mathcal{I}$ (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 abstract 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 $\mathit{pos}\colon \mathit{Pos}$ – 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. A function $\mathit{label}\colon \mathit{Pos} \to \mathit{Label}$ decorates nodes with the information which identifies the grammar rule associated to the node. In the example the *label* of the root node is *If*.



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

As a side effect the values of local variables in the memory are updated using two dynamic function $\mathit{locals}\colon \mathit{Loc} \to \mathit{Adr}$ and $\mathit{mem}\colon \mathit{Adr} \to \mathit{SimpleValue} \cup \{\mathit{Undef}\}$ to assign to local variables memory addresses where the values are stored. Since in C$\#_\mathcal{I}$ the values are of simple types, the equation $\mathit{Value} = \mathit{SimpleValue} \cup \mathit{Adr}$ holds, which will be refined in the extended models to include also references and structs. The uniquely identified local variables are modeled by stipulating $\mathit{Loc} = \mathit{Identifier} \times \mathit{Pos}$, where Pos is the set of positions in the abstract syntax tree.

The indirection through memory addresses is not really needed in C$\#_\mathcal{I}$. In C$\#_\mathcal{I}$ 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$\#_\mathcal{I}$ the reasons of abruption are from the set $\mathit{Abr} = \mathit{Break} \mid \mathit{Continue} \mid \mathit{Goto}(\mathit{Lab})$, to be refined for the extended models. We use an auxiliary dynamic function $\mathit{values}\colon \mathit{Pos} \to \mathit{Result}$ to store intermediate evaluation results from the set $\mathit{Result} = \mathit{SimpleValue} \cup \mathit{Abr} \cup \{\mathit{Undef}, \mathit{Norm}\}$. For the initial state we assume

- $\mathit{mem}(i) = \mathit{Undef}$ for every $i \in \mathit{Adr}$
- pos = root position of the abstract syntax tree
- $\mathit{locals}(x) \in \mathit{Adr}$ for every variable x[3]

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[4]. For a succinct formulation we use a macro $\mathit{context}(\mathit{pos})$ to describe the context of the currently to be handled expression or statement or intermediate result, which

[3]This comes up to assume that the compiler chooses an address for each variable.
[4]The specializations can be regained instruction-wise by mere parameter expansion, a form of refinement that is easily proved to be correct.

has to be matched against the syntactically possible cases (in the textual order of the rule) to select the appropriate computation step. If the subtree at the position *pos* is already evaluated and *pos* carries its result in *values*, then *context*(*pos*) is the parent note of *pos* together with its children where *pos* is marked with the cursor (▶); otherwise, *context*(*pos*) consists of the node at *pos* together with its children. 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 C$\#_{\mathcal{I}}$ the memory value of a given type t at a given address *adr* is defined by $memValue(adr, t) = mem(adr)$). This is described by the following two macros:

$$
\begin{array}{l}
\text{Yield}(val, p) \equiv \\
\quad values(p) := val \\
\quad pos := p
\end{array}
$$

$$
\begin{array}{l}
\text{YieldIndirect}(adr, p) \equiv \\
\quad \textbf{if } AddressPos(p) \textbf{ then } \text{Yield}(adr, p) \textbf{ else } \text{Yield}(memValue(adr, type(p)), p)
\end{array}
$$

We will use the macros in the two forms $\text{Yield}(val) \equiv \text{Yield}(val, pos)$ and $\text{YieldUp}(val) \equiv \text{Yield}(val, up(pos))$, similarly for $\text{YieldIndirect}(adr)$ and $\text{YieldUpIndirect}(adr)$.

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 macro $\text{WriteMem}(adr, t, val)$ denotes here $mem(adr) := val$; it will be refined in the model for C$\#_{\mathcal{O}}$.

$$
\begin{array}{l}
\text{ExecCsharpExp}_I \equiv \textbf{match } context(pos) \\
\quad lit \to \text{Yield}(ValueOfLiteral(lit)) \\
\quad loc \to \text{YieldIndirect}(locals(loc)) \\[4pt]
\quad uop\ exp \ \to pos := exp \\
\quad uop\ {}^\blacktriangleright val \to \textbf{if } \neg UopException(uop, val) \textbf{ then } \text{YieldUp}(Apply(uop, val)) \\[4pt]
\quad exp_1\ bop\ exp_2 \ \to pos := exp_1 \\
\quad {}^\blacktriangleright val\ bop\ exp \ \to pos := exp \\
\quad val_1\ bop\ {}^\blacktriangleright val_2 \to \textbf{if } \neg BopException(bop, val_1, val_2) \textbf{ then } \text{YieldUp}(Apply(bop, val_1, val_2)) \\[4pt]
\quad exp_0 \ ?\ exp_1 \ :\ exp_2 \to pos := exp_0 \\
\quad {}^\blacktriangleright val \ ?\ exp_1 \ :\ exp_2 \to \textbf{if } val \textbf{ then } pos := exp_1 \textbf{ else } pos := exp_2 \\
\quad True \ ?\ {}^\blacktriangleright val \ :\ exp \to \text{YieldUp}(val) \\
\quad False \ ?\ exp \ :\ {}^\blacktriangleright val \to \text{YieldUp}(val) \\[4pt]
\quad loc \texttt{ = } exp \ \to pos := exp \\
\quad loc \texttt{ = } {}^\blacktriangleright val \to \{\text{WriteMem}(locals(loc), type(loc), val),\ \text{YieldUp}(val)\} \\[4pt]
\quad (type)\ exp \ \to pos := exp \\
\quad (type)\ {}^\blacktriangleright val \to \textbf{if } \neg UopException(type, val) \textbf{ then } \text{YieldUp}(Convert(type, val)) \\[4pt]
\quad vexp\ op\texttt{=}\ exp \to pos := vexp \\
\quad {}^\blacktriangleright adr\ op\texttt{=}\ exp \to pos := exp \\
\quad adr\ op\texttt{=}\ {}^\blacktriangleright val \to \textbf{let } t = type(up(pos)) \textbf{ and } v = memValue(adr, t) \textbf{ in} \\
\qquad\qquad \textbf{if } \neg BopException(op, v, val) \textbf{ then} \\
\qquad\qquad\quad \textbf{let } w = Apply(op, v, val) \textbf{ and } result = Convert(t, w) \textbf{ in} \\
\qquad\qquad\qquad \{\text{WriteMem}(adr, t, result),\ \text{YieldUp}(result)\} \\[4pt]
\quad vexp\ op \ \to pos := vexp \qquad\qquad // \text{ where } op \in \{\texttt{++}, \texttt{--}\} \\
\quad {}^\blacktriangleright adr\ op \to \textbf{let } old = memValue(adr, type(pos)) \textbf{ in} \\
\qquad\qquad \textbf{if } \neg UopException(op, old) \textbf{ then} \\
\qquad\qquad\quad \{\text{WriteMem}(adr, type(up(pos)), Apply(op, old)),\ \text{YieldUp}(old)\} \\[4pt]
\quad \texttt{checked}(exp) \quad \to pos := exp \\
\quad \texttt{checked}({}^\blacktriangleright val) \quad \to \text{YieldUp}(val) \\
\quad \texttt{unchecked}(exp) \ \to pos := exp \\
\quad \texttt{unchecked}({}^\blacktriangleright val) \to \text{YieldUp}(val)
\end{array}
$$

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

$$\begin{aligned}&\mathit{AddressPos}(\alpha) \iff \mathit{FirstChild}(\alpha) \wedge (\mathit{label}(\mathit{up}(\alpha)) \in \{\texttt{++}, \texttt{--}\} \vee \mathit{label}(\mathit{up}(\alpha)) \in \mathit{Aop})\\ &\textbf{where } \mathit{FirstChild}(\alpha) \iff \mathit{first}(\mathit{up}(\alpha)) = \alpha\end{aligned}$$

Similarly for being in a checked context which is used to define whether operators throw an overflow exception (in which case a rule will be added in the model for $C\#_{\mathcal{E}}$). 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.

$$\begin{aligned}&\mathit{Checked}(\alpha) \iff \mathit{label}(\alpha) = \mathit{Checked} \vee\\ &\quad (\mathit{label}(\alpha) \neq \mathit{Unchecked} \wedge \mathit{up}(\alpha) \neq \mathit{Undef} \wedge \mathit{Checked}(\mathit{up}(\alpha)))\\ &\mathit{UopException}(\mathit{uop}, \mathit{val}) \iff \mathit{Checked}(\mathit{pos}) \wedge \mathit{Overflow}(\mathit{uop}, \mathit{val})\\ &\mathit{BopException}(\mathit{bop}, \mathit{val}_1, \mathit{val}_2) \iff\\ &\quad \mathit{DivisonByZero}(\mathit{bop}, \mathit{val}_2) \vee \mathit{DecimalOverflow}(\mathit{bop}, \mathit{val}_1, \mathit{val}_2) \vee\\ &\quad (\mathit{Checked}(\mathit{pos}) \wedge \mathit{Overflow}(\mathit{bop}, \mathit{val}_1, \mathit{val}_2))\end{aligned}$$

Similarly, the machine 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 below 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. For the sake of brevity we skip the analogous transition rules for statements `do`, `for`, `switch`, `goto case`, `goto default`.

$$\begin{aligned}&\text{ExecCsharpStm}_I \equiv \textbf{match } \mathit{context}(\mathit{pos})\\
&\quad \texttt{;} \to \text{Yield}(\mathit{Norm})\\
&\quad \mathit{exp}\,\texttt{;} \to \mathit{pos} := \mathit{exp}\\
&\quad {}^{\blacktriangleright}\mathit{val}\,\texttt{;} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{break;} \to \text{Yield}(\mathit{Break})\\
&\quad \texttt{continue;} \to \text{Yield}(\mathit{Continue})\\
&\quad \texttt{goto}\ \mathit{lab}\,\texttt{;} \to \text{Yield}(\mathit{Goto}(\mathit{lab}))\\
&\quad \texttt{if (}\mathit{exp}\texttt{)}\ \mathit{stm}_1\ \texttt{else}\ \mathit{stm}_2 \to \mathit{pos} := \mathit{exp}\\
&\quad \texttt{if (}{}^{\blacktriangleright}\mathit{val}\texttt{)}\ \mathit{stm}_1\ \texttt{else}\ \mathit{stm}_2 \to \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \mathit{stm}_1 \textbf{ else } \mathit{pos} := \mathit{stm}_2\\
&\quad \texttt{if (}\mathit{True}\texttt{)}\ {}^{\blacktriangleright}\mathit{Norm}\ \texttt{else}\ \mathit{stm} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{if (}\mathit{False}\texttt{)}\ \mathit{stm}\ \texttt{else}\ {}^{\blacktriangleright}\mathit{Norm} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{while (}\mathit{exp}\texttt{)}\ \mathit{stm} \to \mathit{pos} := \mathit{exp}\\
&\quad \texttt{while (}{}^{\blacktriangleright}\mathit{val}\texttt{)}\ \mathit{stm} \to \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \mathit{stm} \textbf{ else } \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{while (}\mathit{True}\texttt{)}\ {}^{\blacktriangleright}\mathit{Norm} \to \{\mathit{pos} := \mathit{up}(\mathit{pos}),\ \text{ClearValues}(\mathit{up}(\mathit{pos}))\}\\
&\quad \texttt{while (}\mathit{True}\texttt{)}\ {}^{\blacktriangleright}\mathit{Break} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{while (}\mathit{True}\texttt{)}\ {}^{\blacktriangleright}\mathit{Continue} \to \{\mathit{pos} := \mathit{up}(\mathit{pos}),\ \text{ClearValues}(\mathit{up}(\mathit{pos}))\}\\
&\quad \texttt{while (}\mathit{True}\texttt{)}\ {}^{\blacktriangleright}\mathit{abr} \to \text{YieldUp}(\mathit{abr})\\
&\quad \mathit{type}\ \mathit{loc}\,\texttt{;} \to \text{Yield}(\mathit{Norm})\\
&\quad \mathit{lab}\texttt{:}\ \mathit{stm} \to \mathit{pos} := \mathit{stm}\\
&\quad \mathit{lab}\texttt{:}\ {}^{\blacktriangleright}\mathit{Norm} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{checked}\ \mathit{block} \to \mathit{pos} := \mathit{block}\\
&\quad \texttt{checked}\ {}^{\blacktriangleright}\mathit{Norm} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \texttt{unchecked}\ \mathit{block} \to \mathit{pos} := \mathit{block}\\
&\quad \texttt{unchecked}\ {}^{\blacktriangleright}\mathit{Norm} \to \text{YieldUp}(\mathit{Norm})\\
&\quad \ldots\ {}^{\blacktriangleright}\mathit{abr}\ \ldots \to \textbf{if } \mathit{up}(\mathit{pos}) \neq \mathit{Undef} \wedge \mathit{PropagatesAbr}(\mathit{up}(\mathit{pos})) \textbf{ then } \text{YieldUp}(\mathit{abr})\end{aligned}$$

$$
\begin{array}{ll}
\texttt{\{ \}} & \to \textsc{Yield}(\mathit{Norm})\\
\texttt{\{}\mathit{stm}\ \ldots\texttt{\}} & \to \mathit{pos} := \mathit{stm}\\
\texttt{\{}\ldots {}^{\blacktriangleright}\mathit{Norm}\texttt{\}} & \to \textsc{YieldUp}(\mathit{Norm})\\
\texttt{\{}\ldots {}^{\blacktriangleright}\mathit{Norm}\ \mathit{stm}\ \ldots\texttt{\}} & \to \mathit{pos} := \mathit{stm}\\
\texttt{\{}\ldots {}^{\blacktriangleright}\mathit{Goto}(l)\ \ldots\texttt{\}} & \to \mathbf{let}\ \alpha = \mathit{GotoTarget}(\mathit{first}(\mathit{up}(\mathit{pos})), l)\\
 & \quad\ \mathbf{if}\ \alpha \neq \mathit{Undef}\ \mathbf{then}\ \{\mathit{pos} := \alpha,\ \textsc{ClearValues}(\mathit{up}(\mathit{pos}))\}\\
 & \quad\ \mathbf{else}\ \textsc{YieldUp}(\mathit{Goto}(l))\\
\texttt{\{}\ldots {}^{\blacktriangleright}\mathit{abr}\ \ldots\texttt{\}} & \to \textsc{YieldUp}(\mathit{abr})
\end{array}
$$

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

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

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

$$
\begin{array}{l}
\mathit{GotoTarget}(\alpha, l) =\\
\quad \mathbf{if}\ \mathit{label}(\alpha) = \mathit{Lab}(l)\ \mathbf{then}\ \alpha\\
\quad \mathbf{elseif}\ \mathit{next}(\alpha) = \mathit{Undef}\ \mathbf{then}\ \mathit{Undef}\\
\quad \mathbf{else}\ \mathit{GotoTarget}(\mathit{next}(\alpha), l)
\end{array}
$$

The auxiliary macro $\textsc{ClearValues}(\alpha)$ 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]:

$$
\begin{array}{l}
\textsc{ClearValues}(\alpha) \equiv\\
\quad \mathit{values}(\alpha) := \mathit{Undef}\\
\quad \mathbf{if}\ \mathit{first}(\alpha) \neq \mathit{Undef}\ \mathbf{then}\ \textsc{ClearValuesSeq}(\mathit{first}(\alpha))
\end{array}
$$

$$
\begin{array}{l}
\textsc{ClearValuesSeq}(\alpha) \equiv\\
\quad \textsc{ClearValues}(\alpha)\\
\quad \mathbf{if}\ \mathit{next}(\alpha) \neq \mathit{Undef}\ \mathbf{then}\ \textsc{ClearValuesSeq}(\mathit{next}(\alpha))
\end{array}
$$

3 Refining C$\#_{\mathcal{I}}$ by static class features

In this section we refine the imperative core C$\#_{\mathcal{I}}$ to C$\#_{\mathcal{C}}$ 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 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. $\mathit{PropagatesAbr}(\alpha)$, 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.

In C$\#_{\mathcal{C}}$ 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[6]) 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 [3]. 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$\#_{\mathcal{C}}$. The new set *Arg* of arguments appearing here reflects that besides value parameters also `ref` and `out` parameters can be used.

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

[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 [2] for a precise definition.

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

The type constraints for the new expressions and the return statement are spelt out in [3]. 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.

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):

$$Meth = Type \times Msig$$
$$TypeState = Linked \mid InProgress \mid Initialized$$
$$Frame = Meth \times Pos \times Loc \times Values, \quad \text{where } Values = (Pos \to Result)$$

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:

$$globals\colon Type \times Field \to Adr \qquad frames\colon List(Frame)$$
$$typeState\colon Type \to TypeState \qquad meth\colon Meth$$

We extend the stipulations for the initial state as follows:

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

The submachine $\textsc{ExecCsharp}_C$ extends the machine $\textsc{ExecCsharp}_I$ 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 of $\textsc{ExecCsharp}$ in the sections below consist only in the parallel addition of submachines $\textsc{ExecCsharp}_O$, $\textsc{ExecCsharp}_E$, $\textsc{ExecCsharp}_D$, $\textsc{ExecCsharp}_T$, and $\textsc{ExecCsharp}_U$.

$\textsc{ExecCsharp}_C \equiv$
  $\textsc{ExecCsharpExp}_C$
  $\textsc{ExecCsharpStm}_C$

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

$\textsc{ExecCsharpExp}_C \equiv \textbf{match}\ context(pos)$
  $c.f \rightarrow \textbf{if}\ Initialized(c)\ \textbf{then}\ \textsc{YieldIndirect}(globals(c{::}f))\ \textbf{else}\ \textsc{Initialize}(c)$

  $c.f\ \texttt{=}\ exp \rightarrow pos := exp$
  $c.f\ \texttt{=}^{\blacktriangleright} val \rightarrow \textbf{if}\ Initialized(c)\ \textbf{then}$
      $\textsc{WriteMem}(globals(c{::}f), type(c{::}f), val)$
      $\textsc{YieldUp}(val)$
    $\textbf{else}\ \textsc{Initialize}(c)$

  $c.m\texttt{(}args\texttt{)} \rightarrow pos := \texttt{(}args\texttt{)}$
  $c.m^{\blacktriangleright}\texttt{(}vals\texttt{)} \rightarrow \textsc{InvokeStatic}(c{::}m, vals)$

  $\texttt{ref}\ vexp \rightarrow pos := vexp$
  $\texttt{ref}\ {}^{\blacktriangleright} adr \rightarrow \textsc{YieldUp}(adr)$

  $\texttt{out}\ vexp \rightarrow pos := vexp$
  $\texttt{out}\ {}^{\blacktriangleright} adr \rightarrow \textsc{YieldUp}(adr)$

  $\texttt{()} \rightarrow \textsc{Yield}([])$
  $\texttt{(}arg\texttt{,}\ldots\texttt{)} \rightarrow pos := arg$
  $\texttt{(}val_1\texttt{,}\ldots\texttt{,}^{\blacktriangleright} val_n\texttt{)} \rightarrow \textsc{YieldUp}([val_1,\ldots,val_n])$
  $\texttt{(}\ldots{}^{\blacktriangleright} val\texttt{,}arg\ldots\texttt{)} \rightarrow pos := arg$

The macro $\textsc{InvokeStatic}$ invokes the method – if the class is initialized, otherwise it initializes the class. For methods which are not declared external, $\textsc{InvokeMethod}$ updates the frame stack and the current frame in the expected way, allocating via $\textsc{InitLocals}$ 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 $\textsc{InvokeExtern}$, to be defined separately depending on the class of external (e.g. library) methods[7].

$\textsc{InvokeStatic}(c{::}m, vals) \equiv$
  $\textbf{if}\ Initialized(c)\ \textbf{then}\ \textsc{InvokeMethod}(c{::}m, vals)\ \textbf{else}\ \textsc{Initialize}(c)$

$\textsc{InvokeMethod}(c{::}m, vals) \equiv$
  $\textbf{if}\ \texttt{extern} \in modifiers(c{::}m)\ \textbf{then}\ \textsc{InvokeExtern}(c{::}m, vals)$
  $\textbf{else let}\ p = \textbf{if}\ StaticCtor(c{::}m)\ \textbf{then}\ pos\ \textbf{else}\ up(pos)\ \textbf{in}$
    $frames := push(frames, (meth, p, locals, values))$
    $meth := c{::}m$
    $pos := body(c{::}m)$
    $values := \emptyset$
    $\textsc{InitLocals}(c{::}m, vals)$

[7]For an illustration of this use of external methods see below the model for delegates.

We remind the reader that 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 [9] and [5, 2.4.4] for a justification of this assumption. See also the model for C$\#_{\mathcal{O}}$ where we provide a complete 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 paramter x in the signature of $c::m$.

$\textsc{InitLocals}(c::m, vals) \equiv$
  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**
      $locals(x) := adr$
      $\textsc{WriteMem}(adr, type(x), vals(paramIndex(c::m, x)))$
  forall $x \in RefParams(c::m) \cup OutParams(c::m)$ **do** // ref and out arguments
    $locals(x) := vals(paramIndex(c::m, x))$

The rules for method return in $\textsc{ExecCsharpStm}_C$ trigger an abruption upon returning from a method, resulting in the execution of $\textsc{ExitMethod}$.

$\textsc{ExecCsharpStm}_C \equiv$ **match** $context(pos)$
  `return` exp`;` $\rightarrow pos := exp$
  `return` $^{\blacktriangleright} val$`;` $\rightarrow \textsc{YieldUp}(Return(val))$
  `return;` $\rightarrow \textsc{Yield}(Return)$

  $Return \rightarrow$ **if** $pos = body(meth) \wedge \neg Empty(frames)$ **then** $\textsc{ExitMethod}(Norm)$
  $Return(val) \rightarrow$ **if** $pos = body(meth) \wedge \neg Empty(frames)$ **then** $\textsc{ExitMethod}(val)$

  $^{\blacktriangleright} Norm$`;` $\rightarrow \textsc{YieldUp}(Norm)$

The machine $\textsc{ExitMethod}$ 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 $\textsc{FreeLocals}$ to free the memory used for local variables and value parameters, using an abstract notion $\textsc{FreeMemory}$ of how addresses of local variables and value parameters are actually de-allocated.[8]

$\textsc{ExitMethod}(result) \equiv$
  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
      $values := oldValues \oplus \{oldPos \mapsto result\}$
    $\textsc{FreeLocals}$

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

Following [7, §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 $\textsc{Initialize}$ below) or has terminated normally (see the update of $typeState(c)$ to *Initialized* in $\textsc{ExitMethod}$ above). We therefore define:

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

[8] 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 $\textsc{FreeLocals}$.

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

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

Note that the initialization of a class does not trigger the initialization of its direct base class (as it is the case for Java).

With respect to the execution of initializers of static class fields the Ecma standard [7, §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. We do not model the last behavior, since Microsoft's C# compiler currently creates a static constructor in this case.

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

In this section we refine the static class features of C$\#_{\mathcal{C}}$ by adding objects (for class instances, comprising arrays and structs) together with *instance* fields, methods and constructors[9] as well as inheritance (including overriding and overloading of methods). Accordingly we extend the ASM universes and functions of C$\#_{\mathcal{C}}$ 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$\#_{\mathcal{O}}$. 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 [3].

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$\#_{\mathcal{D}}$ to comprise also delegates. Value types represent a feature that distinguishes C# from Java. A *RefExp* is an expression of a reference type and an *ArrayExp* is an expression of an array type.

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \ldots \mid \text{`}\texttt{null}\text{'} \mid \text{`}\texttt{this}\text{'} \mid \text{`}\texttt{typeof}\text{'}\ \text{`}\texttt{(}\text{'}\ \mathit{RetType}\ \text{`}\texttt{)}\text{'} \mid \mathit{Exp}\ \text{`}\texttt{is}\text{'}\ \mathit{Type} \mid \mathit{Exp}\ \text{`}\texttt{as}\text{'}\ \mathit{RefType} \\
 & \mid & \text{`}\texttt{(}\text{'}\ \mathit{Type}\ \text{`}\texttt{)}\text{'}\ \mathit{Exp} \mid \text{`}\texttt{new}\text{'}\ \mathit{NonArrayType}\ \text{`}\texttt{[}\text{'}\ \mathit{Exps}\ \text{`}\texttt{]}\text{'}\ \{\mathit{Rank}\}\ [\mathit{ArrayInitializer}] \\
\mathit{Vexp} & ::= & \ldots \mid \mathit{Vexp}\ \text{`}\texttt{.}\text{'}\ \mathit{Field} \mid \mathit{RefExp}\ \text{`}\texttt{.}\text{'}\ \mathit{Field} \mid \text{`}\texttt{base}\text{'}\ \text{`}\texttt{.}\text{'}\ \mathit{Field} \mid \mathit{ArrayExp}\ \text{`}\texttt{[}\text{'}\ \mathit{Exps}\ \text{`}\texttt{]}\text{'} \\
\mathit{Sexp} & ::= & \ldots \mid \text{`}\texttt{new}\text{'}\ \mathit{Type}\ \texttt{(}\ [\mathit{Args}]\ \texttt{)} \mid \mathit{Exp}\ \text{`}\texttt{.}\text{'}\ \mathit{Meth}\ \texttt{(}\ [\mathit{Args}]\ \texttt{)} \mid \text{`}\texttt{base}\text{'}\ \text{`}\texttt{.}\text{'}\ \mathit{Meth}\ \texttt{(}\ [\mathit{Args}]\ \texttt{)} \\
\mathit{Exps} & ::= & \mathit{Exp}\ \{\text{`}\texttt{,}\text{'}\ \mathit{Exp}\} \\
\mathit{Rank} & ::= & \text{`}\texttt{[}\text{'}\ \{\ \text{`}\texttt{,}\text{'}\ \}\ \text{`}\texttt{]}\text{'}
\end{array}
$$

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 reassumed by Fig. 4. We refer the reader to [3] 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 follows:

- T any type $\Longrightarrow T \preceq$ `object` and $T \preceq T$
- class S derived from $T \Longrightarrow S \preceq T$

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

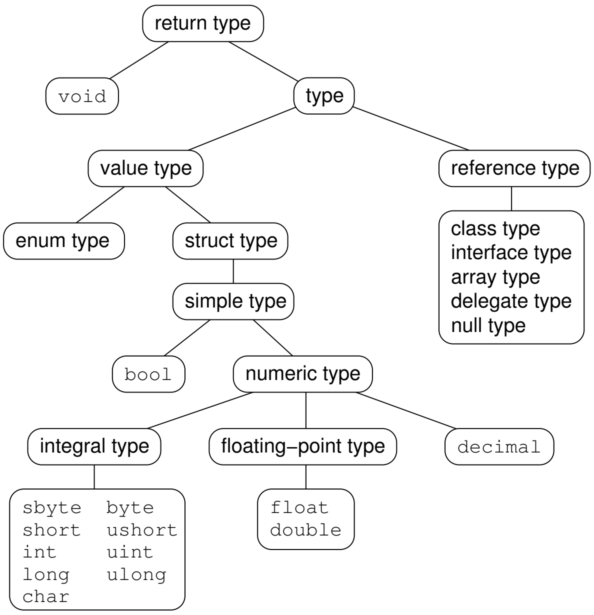


Figure 4: The classification of types of C#.

- 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, $S \preceq T \Longrightarrow S\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]} \preceq T\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]}$

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 $\iff$ $p.f$ is definitely assigned for each instance field f of S.

We assume that as a result of field and method resolution the abstract syntax tree is annotated with 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 $T{::}m\texttt{(}args\texttt{)}$ where m is the signature of a method declared in type T. Moreover, certain expressions are reduced to basic expressions at compile-time.

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 a field f is declared.
- `base.`$m\texttt{(}args\texttt{)}$ in class C is replaced by `this.`$B{::}M\texttt{(}args\texttt{)}$, where $B{::}M$ is the method signature of the method selected by the compiler (the set of applicable methods is constructed starting in the *base class* of C). This selection algorithm is described in [3], formalizing the conditions stated in [7, §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 `new` $T{::}M\texttt{(}args\texttt{)}$ is replaced by `new` $T\texttt{.}T{::}M\texttt{(}args\texttt{)}$.

Hence we split an instance creation expression into a creation part and an invocation of an instance constructor. We assume that class instance constructors return the value of `this`.

Instance constructors of structs need an address for `this`.

- Let S be a struct type. Then $vexp$ `=` `new` $S{::}M\texttt{(}args\texttt{)}$ is replaced by $vexp\texttt{.}S{::}M\texttt{(}args\texttt{)}$.

- Otherwise, `new` S::M`(`$args$`)` is replaced by x`.`S::M`(`$args$`)`, where x is a new temporary local variable of type S. We assume that constructors of structs return the value of `this`.

For automatic boxing we have:

- $vexp$ `=` 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 $type(exp) \preceq T$.
- 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 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. $structField(val, T, f) = val(f)$.

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. 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 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 $runTimeType(ref) = c$ 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. `this` is treated as first parameter and is passed by value. Therefore $paramIndex(c::m, \texttt{this}) = 0$ and `this` is element of both $LocalVars(c::m)$ and $ValueParams(c::m)$.

4.1 Transition rules for C#$_{\mathcal{O}}$

For the refinement of the ExecCsharp transition rules it suffices to add the ExecCsharp$_O$ rules, defined by two submachines to evaluate the new expressions and to execute the new statements respectively:

ExecCsharp$_O$ ≡
  ExecCsharpExp$_O$
  ExecCsharpStm$_O$

ExecCsharpExp$_O$ contains rules for each of the numerous forms of new expressions. For better readability we organize them 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). The rules below for calls of instance methods contain no class initialization test, since an instance method which is not an instance constructor can be called only when the corresponding class is already initialized.

ExecCsharpExp$_O$ ≡ **match** $context(pos)$
  `null` → $null$
  `this` → YieldIndirect($locals$(`this`))
  TestCastExp$_O$
  FieldExp$_O$

$$
\begin{array}{l}
\texttt{new}\ c \to \textbf{let}\ \mathit{ref} = \mathit{new}(\mathit{Ref}, c)\ \textbf{in} \\
\qquad\qquad \mathit{runTimeType}(\mathit{ref}) := c \\
\qquad\qquad \textbf{forall}\ f \in \mathit{instanceFields}(c)\ \textbf{do} \\
\qquad\qquad\quad \textbf{let}\ \mathit{adr} = \mathit{fieldAdr}(\mathit{ref}, f)\ \textbf{and}\ t = \mathit{type}(f)\ \textbf{in} \\
\qquad\qquad\qquad \text{WRITEMEM}(\mathit{adr}, t, \mathit{defaultValue}(t)) \\
\qquad\qquad \text{YIELD}(\mathit{ref}) \\
\mathit{exp}\,.\,T{::}M\,\texttt{(}\mathit{args}\texttt{)} \ \to \mathit{pos} := \mathit{exp} \\
{}^{\blacktriangleright}\mathit{val}\,.\,T{::}M\,\texttt{(}\mathit{args}\texttt{)} \to \textbf{if}\ \mathit{StructValueInvocation}(\mathit{up}(\mathit{pos}))\ \textbf{then} \\
\qquad\qquad\qquad\quad \textbf{let}\ \mathit{adr} = \mathit{new}(\mathit{Adr}, \mathit{type}(\mathit{pos}))\ \textbf{in} \quad \text{// create home for struct value} \\
\qquad\qquad\qquad\qquad \text{WRITEMEM}(\mathit{adr}, \mathit{type}(\mathit{pos}), \mathit{val}) \\
\qquad\qquad\qquad\qquad \mathit{values}(\mathit{pos}) := \mathit{adr} \\
\qquad\qquad\qquad \mathit{pos} := \texttt{(}\mathit{args}\texttt{)} \\
\mathit{val}\,.\,T{::}M^{\blacktriangleright}\texttt{(}\mathit{vals}\texttt{)} \to \textbf{if}\ \mathit{InstanceCtor}(M) \wedge \neg \mathit{Initialized}(T)\ \textbf{then}\ \text{INITIALIZE}(T) \\
\qquad\qquad\qquad \textbf{elseif}\ \mathit{val} \neq \mathit{null}\ \textbf{then}\ \text{INVOKEINSTANCE}(T{::}M, \mathit{val}, \mathit{vals}) \\
\text{ARRAYEXP}_O
\end{array}
$$

A struct value invocation is a method invocation on a struct value.

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

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.

$$
\begin{array}{l}
\text{TESTCASTEXP}_O \equiv \\
\quad \texttt{typeof(}t\texttt{)} \to \text{YIELD}(\mathit{typeObj}(t)) \\
\quad \mathit{exp}\ \texttt{is}\ t \ \to \mathit{pos} := \mathit{exp} \\
\quad {}^{\blacktriangleright}\mathit{val}\ \texttt{is}\ t \to \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{ValueType}\ \textbf{then} \\
\qquad\qquad\qquad \text{YIELDUP}(\mathit{type}(\mathit{pos}) \preceq t) \quad \text{// compile-time property} \\
\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad \text{YIELDUP}(\mathit{val} \neq \mathit{null} \wedge \mathit{runTimeType}(\mathit{val}) \preceq t) \\
\quad \mathit{exp}\ \texttt{as}\ t \ \to \mathit{pos} := \mathit{exp} \\
\quad {}^{\blacktriangleright}\mathit{val}\ \texttt{as}\ t \to \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{ValueType}\ \textbf{then} \\
\qquad\qquad\qquad \text{YIELDUPBOX}(\mathit{type}(\mathit{pos}), \mathit{val}) \quad \text{// box a copy of the value} \\
\qquad\qquad\quad \textbf{elseif}\ (\mathit{val} \neq \mathit{null} \wedge \mathit{runTimeType}(\mathit{val}) \preceq t)\ \textbf{then} \\
\qquad\qquad\qquad \text{YIELDUP}(\mathit{val}) \quad \text{// pass reference through} \\
\qquad\qquad\quad \textbf{else}\ \text{YIELDUP}(\mathit{null}) \quad \text{// convert to null reference} \\
\quad \texttt{(}t\texttt{)}\,\mathit{exp} \ \to \mathit{pos} := \mathit{exp} \\
\quad \texttt{(}t\texttt{)}^{\blacktriangleright}\mathit{val} \to \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{ValueType}\ \textbf{then} \\
\qquad\qquad\qquad \textbf{if}\ t = \mathit{type}(\mathit{pos})\ \textbf{then}\ \text{YIELDUP}(\mathit{val}) \quad \text{// compile-time identity} \\
\qquad\qquad\qquad \textbf{if}\ t \in \mathit{RefType}\ \textbf{then}\ \text{YIELDUPBOX}(\mathit{type}(\mathit{pos}), \mathit{val}) \quad \text{// box value} \\
\qquad\qquad\quad \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{RefType}\ \textbf{then} \\
\qquad\qquad\qquad \textbf{if}\ t \in \mathit{RefType} \wedge (\mathit{val} = \mathit{null} \vee \mathit{runTimeType}(\mathit{val}) \preceq t)\ \textbf{then} \\
\qquad\qquad\qquad\quad \text{YIELDUP}(\mathit{val}) \quad \text{// pass reference through} \\
\qquad\qquad\qquad \textbf{if}\ t \in \mathit{ValueType} \wedge \mathit{val} \neq \mathit{null} \wedge t = \mathit{runTimeType}(\mathit{val})\ \textbf{then} \\
\qquad\qquad\qquad\quad \text{YIELDUP}(\mathit{memValue}(\mathit{valueAdr}(\mathit{val}), t)) \quad \text{// un-box a copy of the value}
\end{array}
$$

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. We use $\mathit{type}(\mathit{exp}\,.\,t{::}f) = \mathit{type}(t{::}f)$.

$$
\begin{array}{l}
\text{FIELDEXP}_O \equiv \\
\quad \mathit{exp}\,.\,t{::}f \ \to \mathit{pos} := \mathit{exp} \\
\quad {}^{\blacktriangleright}\mathit{val}\,.\,t{::}f \to \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{ValueType} \wedge \mathit{val} \notin \mathit{Adr}\ \textbf{then} \\
\qquad\qquad\qquad \text{YIELDUP}(\mathit{structField}(\mathit{val}, \mathit{type}(\mathit{pos}), t{::}f)) \\
\qquad\qquad\quad \textbf{elseif}\ \mathit{val} \neq \mathit{null}\ \textbf{then} \\
\qquad\qquad\qquad \text{YIELDUPINDIRECT}(\mathit{fieldAdr}(\mathit{val}, t{::}f))
\end{array}
$$

$$\begin{array}{lll}
\mathit{exp}_1\,\texttt{.}\,t{::}f\;\texttt{=}\;\mathit{exp}_2 & \to & \mathit{pos} := \mathit{exp}_1\\
\blacktriangleright \mathit{val}\,\texttt{.}\,t{::}f\;\texttt{=}\;\mathit{exp} & \to & \mathit{pos} := \mathit{exp}\\
\mathit{val}_1\,\texttt{.}\,t{::}f\;\texttt{=}\;\blacktriangleright \mathit{val}_2 & \to & \textbf{if } \mathit{val}_1 \neq \mathit{null} \textbf{ then}\\
 & & \quad \textsc{SetField}(\mathit{val}_1, t{::}f, \mathit{val}_2)\\
 & & \quad \textsc{YieldUp}(\mathit{val}_2)
\end{array}$$

$C\#_{\mathcal{O}}$ 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 $\mathit{dim}(n)$ we denote a sequence of $n-1$ commas, hence $T\,\texttt{[}\mathit{dim}(n)\texttt{]}$ 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 $\mathit{dimLength}(\mathit{ref}, i)$.

$$\begin{array}{l}
\textsc{ArrayExp}_O \equiv\\
\quad \texttt{new}\ T\,\texttt{[}\mathit{exp}_1\texttt{,}\dots\texttt{,}\mathit{exp}_n\texttt{]}\,\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]} \to \mathit{pos} := \mathit{exp}_1\\
\quad \texttt{new}\ T\,\texttt{[}l_1\texttt{,}\dots\texttt{,}\blacktriangleright l_n\texttt{]}\,\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]} \to\\
\qquad \textbf{if } \forall i \in [1\,..\,n]\,(0 \le l_i) \textbf{ then}\\
\qquad\quad \textbf{let } S = T\,\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]} \textbf{ in}\\
\qquad\quad \textbf{let } \mathit{ref} = \mathit{new}(\mathit{Ref}, [l_1,\dots,l_n], S) \textbf{ in}\\
\qquad\qquad \mathit{runTimeType}(\mathit{ref}) := T\,\texttt{[}\mathit{dim}(n)\texttt{]}\,\texttt{[}R_1\texttt{]}\cdots\texttt{[}R_k\texttt{]}\\
\qquad\qquad \textbf{forall } i \in [1\,..\,n] \textbf{ do } \mathit{dimLength}(\mathit{ref}, i-1) := l_i\\
\qquad\qquad \textbf{forall } \alpha \in [0\,..\,l_1-1] \times \cdots \times [0\,..\,l_n-1] \textbf{ do}\\
\qquad\qquad\quad \textsc{WriteMem}(\mathit{elemAdr}(\mathit{ref}, \alpha), S, \mathit{defaultValue}(S))\\
\qquad\qquad \textsc{YieldUp}(\mathit{ref})\\
\quad \mathit{exp}_0\,\texttt{[}\mathit{exp}_1\texttt{,}\dots\texttt{,}\mathit{exp}_n\texttt{]} \to \mathit{pos} := \mathit{exp}_0\\
\quad \blacktriangleright \mathit{ref}\,\texttt{[}\mathit{exp}_1\texttt{,}\dots\texttt{,}\mathit{exp}_n\texttt{]} \to \mathit{pos} := \mathit{exp}_1\\
\quad \mathit{ref}\,\texttt{[}i_1\texttt{,}\dots\texttt{,}\blacktriangleright i_n\texttt{]} \to\\
\qquad \textbf{if } \mathit{ref} \neq \mathit{null} \land \forall k \in [1\,..\,n]\,(0 \le i_k < \mathit{dimLength}(\mathit{ref}, k-1)) \land\\
\qquad\quad (\mathit{RefOrOutArg}(\mathit{up}(\mathit{pos})) \land \mathit{type}(\mathit{up}(\mathit{pos})) \in \mathit{RefType} \to\\
\qquad\qquad \mathit{elementType}(\mathit{runTimeType}(\mathit{ref})) = \mathit{type}(\mathit{up}(\mathit{pos})))\\
\qquad \textbf{then}\\
\qquad\quad \textsc{YieldUpIndirect}(\mathit{elemAdr}(\mathit{ref}, (i_1,\dots,i_n)))\\
\quad \mathit{exp}_0\,\texttt{[}\mathit{exp}_1\texttt{,}\dots\texttt{,}\mathit{exp}_n\texttt{]}\;\texttt{=}\;\mathit{exp}_{n+1} \to \mathit{pos} := \mathit{exp}_0\\
\quad \blacktriangleright \mathit{ref}\,\texttt{[}\mathit{exp}_1\texttt{,}\dots\texttt{,}\mathit{exp}_n\texttt{]}\;\texttt{=}\;\mathit{exp} \to \mathit{pos} := \mathit{exp}_1\\
\quad \mathit{ref}\,\texttt{[}i_1\texttt{,}\dots\texttt{,}\blacktriangleright i_n\texttt{]}\;\texttt{=}\;\mathit{exp} \to \mathit{pos} := \mathit{exp}\\
\quad \mathit{ref}\,\texttt{[}i_1\texttt{,}\dots\texttt{,}i_n\texttt{]}\;\texttt{=}\;\blacktriangleright \mathit{val} \to\\
\qquad \textbf{let } T = \mathit{elementTyperunTimeType}(\mathit{ref})) \textbf{ in}\\
\qquad\quad \textbf{if } \mathit{ref} \neq \mathit{null} \land \forall k \in [1\,..\,n]\,(0 \le i_k < \mathit{dimLength}(\mathit{ref}, k-1)) \land\\
\qquad\qquad (\mathit{type}(\mathit{pos}) \in \mathit{RefType} \to \mathit{runTimeType}(\mathit{val}) \preceq T)\\
\qquad\quad \textbf{then}\\
\qquad\qquad \textsc{WriteMem}(\mathit{elemAdr}(\mathit{ref}, (i_1,\dots,i_n)), T, \mathit{val})\\
\qquad\qquad \textsc{YieldUp}(\mathit{val})
\end{array}$$

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.

$$\begin{array}{ll}
\textsc{InvokeInstance}(T{::}M, \mathit{val}, \mathit{vals}) \equiv & \\
\quad \textbf{if } \mathit{callKind}(T{::}M) = \mathit{Virtual} \textbf{ then} & \text{// indirect call, } \mathit{val} \in \mathit{Ref}\\
\qquad \textbf{let } S = \mathit{lookup}(\mathit{runTimeType}(\mathit{val}), T{::}M) \textbf{ in} & \\
\qquad \textbf{let } \mathit{this} = \textbf{if } S \in \mathit{StructType} \textbf{ then } \mathit{valueAdr}(\mathit{val}) \textbf{ else } \mathit{val} \textbf{ in} & \\
\qquad\quad \textsc{InvokeMethod}(S{::}M, [\mathit{this}] \cdot \mathit{vals}) & \\
\quad \textbf{if } \mathit{callKind}(T{::}M) = \mathit{NonVirtual} \textbf{ then} & \text{// direct call, } \mathit{val} \in \mathit{Adr} \cup \mathit{Ref}\\
\qquad \textsc{InvokeMethod}(T{::}M, [\mathit{val}] \cdot \mathit{vals}) &
\end{array}$$

In $C\#_{\mathcal{O}}$ the notion of reading from the memory is refined by extending the simple equation $\mathit{memValue}(\mathit{adr}, t) = \mathit{mem}(\mathit{adr})$ of $C\#_{\mathcal{I}}$ 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.

$$\begin{array}{l} memValue(adr, t) = \\ \quad \textbf{if } t \in SimpleType \cup RefType \textbf{ then } mem(adr) \\ \quad \textbf{elseif } t \in StructType \textbf{ then } \{f \mapsto getField(adr, f) \mid f \in instanceFields(t)\} \end{array}$$

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

Also 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:

$$\begin{array}{l} \text{WRITEMEM}(adr, t, val) \equiv \\ \quad \textbf{if } t \in SimpleType \cup RefType \textbf{ then } mem(adr) := val \\ \quad \textbf{elseif } t \in StructType \textbf{ then} \\ \qquad \textbf{forall } f \in instanceFields(t) \textbf{ do } \text{SETFIELD}(adr, f, val(f)) \end{array}$$

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

The notion of *AddressPos* from $C\#_{\mathcal{I}}$ is refined to include also lvalue nodes of *StructType*.

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

Address positions are: `ref` $\square$, `out` $\square$, $\square$`++`, $\square$`--`, $\square$ *op*`=` *exp*, $\square$`.`f, $\square$`.`m`(`*args*`)`.

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

$$\begin{array}{l} \text{YIELDUPBOX}(t, val) \equiv \textbf{let } ref = new(Ref) \textbf{ and } adr = new(Adr, t) \textbf{ in} \\ \quad runTimeType(ref) := t \\ \quad valueAdr(ref) := adr \\ \quad \text{WRITEMEM}(adr, t, val) \\ \quad \text{YIELDUP}(ref) \end{array}$$

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

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

In the context of the machine EXECCSHARP this comes up to specify an abstract memory management. In fact **let** $adr = new(Adr, T)$ **in** P stands for the sequential execution of a new address allocation followed by P:

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

where the operator **seq** for sequential execution of two ASMs M, N is to be understood as defined for turbo ASMs in [4] (alternatively see [5, 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** $ref = new(Ref, T)$ **in** P stands for the sequential execution of address allocation for all instance fields of a given type followed by P:

$$\begin{array}{l} \textbf{let } ref = new(Ref, T) \textbf{ in } P \equiv \\ \quad \textbf{import } ref \textbf{ do} \\ \qquad Ref(ref) := True \\ \qquad \text{ALLOCFIELDS}(ref, instanceFields(T)) \\ \quad \textbf{seq } P \end{array}$$

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

$$
\begin{array}{l}
\textbf{let } \mathit{ref} = \mathit{new}(\mathit{Ref}, [l_1, \ldots, l_n], T) \textbf{ in } P \equiv \\
\quad \textbf{import } \mathit{ref} \textbf{ do} \\
\quad\quad \mathit{Ref}(\mathit{ref}) := \mathit{True} \\
\quad\quad \textbf{forall } \alpha \in [0 \mathinner{..} l_1 - 1] \times \cdots \times [0 \mathinner{..} l_n - 1] \textbf{ do} \\
\quad\quad\quad \textbf{import } \mathit{adr} \textbf{ do} \\
\quad\quad\quad\quad \mathit{elemAdr}(\mathit{ref}, \alpha) := \mathit{adr} \\
\quad\quad\quad\quad \textsc{AllocAdr}(\mathit{adr}, T) \\
\quad \textbf{seq } P
\end{array}
$$

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 [2] (or see alternatively [5, Ch.4.1.2]):

$$
\begin{array}{l}
\textsc{AllocAdr}(\mathit{adr}, T) \equiv \\
\quad \mathit{Adr}(\mathit{adr}) := \mathit{True} \\
\quad \textbf{if } T \in \mathit{StructType} \textbf{ then } \textsc{AllocFields}(\mathit{adr}, \mathit{instanceFields}(T))
\end{array}
$$

$$
\begin{array}{l}
\textsc{AllocFields}(x, \mathit{fs}) \equiv \\
\quad \textbf{forall } f \in \mathit{fs} \textbf{ import } \mathit{adr} \textbf{ do} \\
\quad\quad \mathit{fieldAdr}(x, f) := \mathit{adr} \\
\quad\quad \textsc{AllocAdr}(\mathit{adr}, \mathit{type}(f))
\end{array}
$$

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

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

- 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 $i < j \Longrightarrow E_j \not\preceq E_i$ (and obviously $E_i \preceq$ `System.Exception`) holds for every try-catch statement `try` *block* `catch (`$E_1\ x_1$`)` $\mathit{block}_1 \ldots$ `catch (`$E_n\ x_n$`)` block_n

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.

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

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:

```
try TryBlock                                try {
catch (E_1 x_1) CatchBlock_1                  try TryBlock
  ⋮                                           catch (E_1 x_1) CatchBlock_1
catch (E_n x_n) CatchBlock_n        ⟹          ⋮
finally FinallyBlock                          catch (E_n x_n) CatchBlock_n
                                            } finally FinallyBlock
```

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);
  }
}
```

For *stm* $\equiv$ `try` *tryBlock* `catch (...)` $catchBlock_1$ ... `catch (...)` $catchBlock_n$ the reachability rules and the definite assignment constraints 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*) 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*)
- $after(stm) = after(tryBlock) \cup after(finallyBlock)$

5.1 Transition rules for C#$_\mathcal{E}$

The transition rules for ExecCsharp$_E$ are defined by two submachines. 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$_E$ $\equiv$
  ExecCsharpExp$_E$
  ExecCsharpStm$_E$

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 paramter of FailUp.

ExecCsharpExp$_E$ $\equiv$ **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

$\text{FailUp}(E) \equiv \text{InvokeMethod}(\mathit{ExcSupport}::\texttt{Throw}E, [\,])$

$\text{CastExceptions} \equiv \textbf{match}\ \mathit{context}(\mathit{pos})$
$\quad \texttt{(}t\texttt{)}{}^{\blacktriangleright}\mathit{val} \rightarrow$
$\quad\quad \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{RefType}\ \textbf{then}$
$\quad\quad\quad \textbf{if}\ t \in \mathit{RefType} \wedge \mathit{val} \neq \mathit{Null} \wedge \mathit{runTimeType}(\mathit{val}) \not\preceq t\ \textbf{then}$
$\quad\quad\quad\quad \text{FailUp}(\texttt{InvalidCastException})$
$\quad\quad\quad \textbf{if}\ t \in \mathit{ValueType}\ \textbf{then}$ // attempt to unbox
$\quad\quad\quad\quad \textbf{if}\ \mathit{val} = \mathit{Null}\ \textbf{then}\ \text{FailUp}(\texttt{NullReferenceException})$
$\quad\quad\quad\quad \textbf{elseif}\ t \neq \mathit{runTimeType}(\mathit{val})\ \textbf{then}\ \text{FailUp}(\texttt{InvalidCastException})$
$\quad\quad \textbf{if}\ \mathit{type}(\mathit{pos}) \in \mathit{SimpleType} \wedge t \in \mathit{SimpleType} \wedge \mathit{Checked}(\mathit{pos}) \wedge \mathit{Overflow}(t, \mathit{val})$
$\quad\quad\quad \textbf{then}\ \text{FailUp}(\texttt{OverflowException})$

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

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.

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

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. The semantics of `throw;` is explained in terms of the exception Stack *excStack*. When an exception is caught, it is pushed on top of the exception stack. 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.

$\text{ExecCsharpStm}_E \equiv \textbf{match}\ \mathit{context}(\mathit{pos})$
$\quad \texttt{throw}\ \mathit{exp}\texttt{;} \rightarrow \mathit{pos} := \mathit{exp}$
$\quad \texttt{throw}\ {}^{\blacktriangleright}\mathit{ref}\texttt{;} \rightarrow \textbf{if}\ \mathit{ref} = \mathit{Null}\ \textbf{then}\ \text{FailUp}(\texttt{NullReferenceException})$
$\quad\quad\quad\quad\quad \textbf{else}\ \{\text{InitStackTrace}(\mathit{ref}, \mathit{meth}),\ \text{YieldUp}(\mathit{Exc}(\mathit{ref}))\}$
$\quad \texttt{throw;} \rightarrow \text{Yield}(\mathit{Exc}(\mathit{top}(\mathit{excStack})))$
$\quad \texttt{try}\ \mathit{block}\ \texttt{catch (}E\ x\texttt{)}\ \mathit{stm}\ \ldots \rightarrow \mathit{pos} := \mathit{block}$
$\quad \texttt{try}\ {}^{\blacktriangleright}\mathit{Norm}\ \texttt{catch (}E\ x\texttt{)}\ \mathit{stm}\ \ldots \rightarrow \text{YieldUp}(\mathit{Norm})$
$\quad \texttt{try}\ {}^{\blacktriangleright}\mathit{Exc}(\mathit{ref})\ \texttt{catch(}E_1\ x_1\texttt{)}\ \mathit{stm}_1\ \ldots\ \texttt{catch(}E_n\ x_n\texttt{)}\ \mathit{stm}_n \rightarrow$

$$
\begin{array}{l}
\quad\quad \textbf{if } \exists i \in [1\mathinner{..}n]\ \mathit{runTimeType}(\mathit{ref}) \preceq E_i \textbf{ then}\\
\quad\quad\quad \textbf{let } j = \min\{i \in [1\mathinner{..}n] \mid \mathit{runTimeType}(\mathit{ref}) \preceq E_i\} \textbf{ in}\\
\quad\quad\quad\quad \mathit{pos} := \mathit{stm}_j\\
\quad\quad\quad\quad \mathit{excStack} := \mathit{push}(\mathit{ref}, \mathit{excStack})\\
\quad\quad\quad\quad \textsc{WriteMem}(\mathit{locals}(x_j), \texttt{object}, \mathit{ref})\\
\quad\quad \textbf{else } \textsc{YieldUp}(\mathit{Exc}(\mathit{ref}))\\
\quad \texttt{try } {}^{\blacktriangleright}\mathit{abr}\ \texttt{catch(}E_1\ x_1\texttt{)}\ \mathit{stm}_1\ \ldots\ \texttt{catch(}E_n\ x_n\texttt{)}\ \mathit{stm}_n\ \rightarrow \textsc{YieldUp}(\mathit{abr})\\
\quad \texttt{try } \mathit{Exc}(\mathit{ref})\ \ldots\ \texttt{catch(}\ldots\texttt{)}\ {}^{\blacktriangleright}\mathit{res}\ \ldots \rightarrow \{\mathit{excStack} := \mathit{pop}(\mathit{excStack}),\ \textsc{YieldUp}(\mathit{res})\}\\
\quad \texttt{try } \mathit{tryBlock}\ \texttt{finally}\ \mathit{finallyBlock} \rightarrow \mathit{pos} := \mathit{tryBlock}\\
\quad \texttt{try } {}^{\blacktriangleright}\mathit{res}\ \texttt{finally}\ \mathit{finallyBlock} \quad \rightarrow \mathit{pos} := \mathit{finallyBlock}\\
\quad \texttt{try } \mathit{res}\ \texttt{finally}\ {}^{\blacktriangleright}\mathit{Norm} \quad\quad \rightarrow \textsc{YieldUp}(\mathit{res})\\
\quad \texttt{try } \mathit{res}\ \texttt{finally}\ {}^{\blacktriangleright}\mathit{Exc}(\mathit{ref}) \quad \rightarrow \textsc{YieldUp}(\mathit{Exc}(\mathit{ref}))\\
\quad \mathit{Exc}(\mathit{ref}) \rightarrow \textbf{if } \mathit{pos} = \mathit{body}(\mathit{meth}) \wedge \neg \mathit{Empty}(\mathit{frames}) \textbf{ then}\\
\quad\quad \textbf{if } \mathit{StaticCtor}(\mathit{meth}) \textbf{ then } \mathit{typeState}(\mathit{type}(\mathit{meth})) := \mathit{Exc}(\mathit{ref})\\
\quad\quad \textbf{else } \textsc{AppendStackTrace}(\mathit{ref}, \mathit{meth}(\mathit{top}(\mathit{frames})))\\
\quad\quad \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 C$\#_{\mathcal{C}}$ re-throws the exception object of a type which had an exception in the static constructor, thus preventing its re-initialization.

$$
\begin{array}{l}
\textsc{Initialize}(c) \equiv\\
\quad \ldots\\
\quad \textbf{if } \mathit{typeState}(c) = \mathit{Exc}(\mathit{ref}) \textbf{ then } \textsc{Yield}(\mathit{Exc}(\mathit{ref}))
\end{array}
$$

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.

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:

$\texttt{delegate } T\ D\texttt{(}S_1\ x_1\texttt{, }\ldots\texttt{ ,}S_n\ x_n\texttt{);}$

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 an $\mathit{invocationList}\colon \mathit{Ref} \rightarrow \mathit{Delegate}^* \cup \{\mathit{Undef}\}$ 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 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 $\mathit{Delegate} = \mathit{Meth} \cup (\mathit{Ref} \times \mathit{Meth})$. 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:

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

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 [3].

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*.[10] It then suffices to refine the ASM rule InvokeExtern defined in 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 analogously of `_length`.

$\text{InvokeExtern}(T{::}M, vals) \equiv$
  if $T \in \mathit{DelegateType}$ **then**
    if $name(M) =$ `_length` **then** $\text{DelegateLength}(vals(0))$
    if $name(M) =$ `_invoke` **then** $\text{InvokeDelegate}(vals(0), vals(1), drop(vals, 2))$

$\text{DelegateLength}(ref) \equiv$
  $\text{YieldUp}(length(invocationList(ref)))$

$\text{InvokeDelegate}(ref, i, vals) \equiv$
  match $invocationList(ref)(i)$
    $T{::}M \rightarrow \text{InvokeStatic}(T{::}M, vals)$
    $(target, T{::}M) \rightarrow \text{InvokeInstance}(T{::}M, target, vals)$

Since there are no new statements appearing in $C\#_{\mathcal{D}}$, the addition of ExecCsharp_D consists in the following ASM ExecCsharpExp_D, which defines the meaning of delegate instance creation.

$\text{ExecCsharpExp}_D \equiv$ **match** $context(pos)$
  `new` D`(`$T{::}M$`)` $\rightarrow$
    let $d = new(Ref, D)$ **in**
      $runTimeType(d) := D$
      $invocationList(d) := [T{::}M]$
      $\text{Yield}(d)$
  `new` D`(`exp`.`$T{::}M$`)` $\rightarrow pos := exp$
  `new` D`(`$^{\blacktriangleright} ref$`.`$T{::}M$`)` $\rightarrow$
    if $ref = Null$ **then** $\text{FailUp}($`NullReferenceException`$)$
    else let $d = new(Ref, D)$ **in**
      $runTimeType(d) := D$
      $invocationList(d) := [(ref, T{::}M)]$

[10] In [7, §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.

$\text{YIELDUP}(d)$

$\texttt{new } D\texttt{(}exp\texttt{)} \;\to pos := exp$

$\texttt{new } D\texttt{(}{}^{\blacktriangleright} ref\texttt{)} \to$

if $ref = Null$ **then** $\text{FAILUP}(\texttt{NullReferenceException})$

else let $d = new(Ref, D)$ **in**

$runTimeType(d) := D$

$invocationList(d) := invocationList(ref)$ // Ecma §14.5.10.3

// Microsoft .NET Framework:

// $invocationList(d) := [(ref, D{::}\texttt{Invoke}(S_1, \ldots, S_n))]$

$\text{YIELDUP}(d)$

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, and they can be *combined* (concatenated with '`+`') and elements 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==`.

$\text{INVOKEEXTERN}(T{::}M, vals) \equiv$

...

if $T \in DelegateType$ **then**

if $name(M) = \texttt{operator+}$ **then** $\text{DELEGATECOMBINE}(T, vals(0), vals(1))$

if $name(M) = \texttt{operator-}$ **then** $\text{DELEGATEREMOVE}(T, vals(0), vals(1))$

if $name(M) = \texttt{operator==}$ **then** $\text{DELEGATEEQUAL}(vals(0), vals(1))$

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.

$\text{DELEGATECOMBINE}(D, r_1, r_2) \equiv$

if $r_1 = Null$ **then** $\text{YIELDUP}(r_2)$

elseif $r_2 = Null$ **then** $\text{YIELDUP}(r_1)$

else let $d = new(Ref, D)$ **in**

$runTimeType(d) := D$

$invocationList(d) := invocationList(r_1) \cdot invocationList(r_2)$

$\text{YIELDUP}(d)$

$\text{DELEGATEREMOVE}(D, r_1, r_2) \equiv$

if $r_1 = Null$ **then** $\text{YIELDUP}(Null)$

elseif $r_2 = Null$ **then** $\text{YIELDUP}(r_1)$

else let $l_1 = invocationList(r_1)$ **and** $l_2 = invocationList(r_2)$ **in**

if $l_1 = l_2$ **then** $\text{YIELDUP}(Null)$

elseif $Subword(l_2, l_1)$ **then let** $d = new(Ref, D)$ **in**

$runTimeType(d) := D$

$invocationList(d) := prefix(l_2, l_1) \cdot suffix(l_2, l_1)$

$\text{YIELDUP}(d)$

else $\text{YIELDUP}(r_1)$

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 $suffix(u, v)$ the part of v after the last occurrence of u in v.

$\text{DELEGATEEQUAL}(r_1, r_2) \equiv$

if $r_1 = Null \vee r_2 = Null$ **then** $\text{YIELDUP}(r_1 = r_2)$

else let $l_1 = invocationList(r_1)$ **and** $l_2 = invocationList(r_2)$ **in**

$\text{YIELDUP}(length(l_1) = length(l_2) \wedge \forall i < length(l_1)\, (l_1(i) = l_2(i)))$

6.2 Properties, events and further features in C#$_\mathcal{D}$

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.

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[11], 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:

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

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

This translation comprises also expressions of the form $\mathit{exp}_1\,\texttt{.}\,P\ \mathit{op}\texttt{=}\ \mathit{exp}_2$, since they can be assumed to be compiled to $\langle x \texttt{ = } \mathit{exp}_1,\ y \texttt{ = } x\texttt{.get\_P()}\ \mathit{op}\ \mathit{exp}_2,\ x\texttt{.set\_P(}y\texttt{)},\ y\rangle$ 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:

$$
\begin{array}{ll}
\langle \mathit{exp}\,\textbf{,}\,\ldots\rangle & \to \mathit{pos} := \mathit{exp} \\
\langle \mathit{val}_1\,\textbf{,}\,\ldots\,\textbf{,}\,{}^{\blacktriangleright}\mathit{val}_n\rangle & \to \textsc{YieldUp}(\mathit{val}_n) \\
\langle \ldots{}^{\blacktriangleright}\mathit{val}\,\textbf{,}\,\mathit{exp}\ldots\rangle & \to \mathit{pos} := \mathit{exp}
\end{array}
$$

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 properties is reduced to the semantics of arrays and methods via the following compile-time translation (and corresponding operator expression translation as explained for properties):

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

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

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

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

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 `void add_`X`(`D `value)` and `void remove_`X`(`D `value)`, which are reserved signatures 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 [6][12].

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

Further constructs. For similar syntactical reductions to those given above, which 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 see [3].

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. This 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. As an alternative to pinning, data of unmanaged type can also be '`stackalloc`'ated, instead of using the heap.

Java has no such unsafe extension. The refinement consists mainly in a definition of the *mem*ory function in terms of byte sequences, using an encoding of simple types and a corresponding refinement of the function *structField*.

[12] If one prefers not to depend on 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 multipe thread execution model.

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

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

$$Type ::= ValueType \mid RefType \mid PointerType \quad PointerType ::= UnmanagedType\ `\texttt{*}' \mid `\texttt{void}'\ `\texttt{*}'$$

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 '`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.

$$\begin{array}{lcl}
Exp & ::= & \ldots \mid `\texttt{\&}'\ Vexp \mid Exp\ `\texttt{->}'\ Meth\ \texttt{(}\ [Args]\ \texttt{)} \mid Exp\ `\texttt{->}'\ Field \\
 & \mid & `\texttt{sizeof}'\ `\texttt{(}'\ UnmanagedType\ `\texttt{)}' \\
Vexp & ::= & \ldots \mid `\texttt{*}'\ Exp \\
Stm & ::= & \ldots \mid `\texttt{unsafe}'\ Block \mid `\texttt{fixed}'\ `\texttt{(}'\ PointerType\ Loc\ \texttt{=}\ Exp\ `\texttt{)}'\ Stm \\
Bstm & ::= & \ldots \mid PointerType\ Loc\ `\texttt{=}'\ `\texttt{stackalloc}'\ UnmanagedType\ `\texttt{[}'\ Exp\ `\texttt{]}'\ `\texttt{;}'
\end{array}$$

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 following constraints are imposed.

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

We assume e`->` m to be translated to `(*`e`).`m and e`[`i`]` to `*(`e `+` i`)`.

For statements the following type constraints are assumed:

Statement	Constraints
T`*` p `= stackalloc` T`[`exp`];`	$type(exp) = \texttt{int}$, T unmanaged
`fixed (char*` p `=` exp`)` stm	$type(exp) = \texttt{string}$, p read-only in stm
`fixed (`T`*` p `=` exp`)` stm	$type(exp) = T\texttt{[}R\texttt{]}$, 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

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. Hence, the garbage collection can determine the pinned objects by looking at the values of pinned local variables on the frames stack.

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 ($Adr = \mathbb{N}$):

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

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 $sizeOf(T\texttt{*}) = sizeOf(\texttt{void*})$.

A function *fieldOffset*: $UnmanagedStructType \times Field \rightarrow \mathbb{N}$ is used to describe the layout of unmanaged structs. It has to satisfy the following constraint for ever 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 $runTimeType(ref) = T\texttt{[}dim(n)\texttt{]}$, where T is an unmanaged type and $l_i = dimLength(ref, i-1)$ for $i \in [1 \mathinner{..} n]$, we assume the following:

$$elemAdr(ref, [i_1, i_2, \ldots, i_n]) = elemAdr(ref, [0, \ldots, 0]) + (\ldots(i_1 \cdot l_2 + i_2) \cdot l_3 + \ldots + i_n) \cdot sizeOf(T)$$

7.2 Transition rule refinement for unsafe code

Besides the rules below which define the semantics of the new expressions and statements we have to refine the notions of reading from and writing to memory for values of unmanaged type.

$memValue(adr, t) =$
  if $t \in RefType$ **then** $mem(adr)$
  elseif $t \in UnmanagedType$ **then**
    $[mem(adr + i) \mid i \in [0 \mathinner{..} sizeOf(t) - 1]\,]$
  elseif $t \in StructType$ **then**
    $\{f \mapsto getField(adr, f) \mid f \in instanceFields(t)\}$

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

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

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

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:

$structField(val, T, f) =$
  if $T \in ManagedType$ **then** $val(f)$
  else let $n = fieldOffset(T, f)$ **in** $[val(i) \mid n \leq i < n + sizeOf(type(f))]$

In the rules for $\textsc{ExecCsharpExp}_U$ 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.

$\textsc{ExecCsharpExp}_U \equiv$ **match** $context(pos)$
  `sizeof(`T`)` $\rightarrow \textsc{Yield}(sizeOf(T))$
  `&`$exp \quad \rightarrow pos := exp$
  `&`$^{\blacktriangleright} adr \rightarrow \textsc{YieldUp}(adr)$

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

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

The rules for pointer arithmetic can be summarized as follows:

$Apply(\texttt{+}(T\texttt{*}, \texttt{int}), adr, i) = adr + i \cdot sizeOf(T)$
$Apply(\texttt{+}(\texttt{int}, T\texttt{*}), i, adr) = adr + i \cdot sizeOf(T)$
$Apply(\texttt{-}(T\texttt{*}, T\texttt{*}), adr_1, adr_2) = (adr_1 - adr_2)/sizeOf(T)$
$Convert(T\texttt{*}, adr) = adr = Convert(S, adr) \quad$ 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.

ExecCsharpStm$_U$ $\equiv$ **match** $context(pos)$
unsafe $block \to pos := block$
unsafe $^{\blacktriangleright} Norm \to$ YieldUp($Norm$)

T* loc = stackalloc T[exp]; $\to pos := exp$
T* loc = stackalloc T[$^{\blacktriangleright} i$]; $\to$ **let** $adr = new(Adr, T, i)$ **in**
WriteMem($locals(loc)$, T*, adr)
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 [7].

References

[1] T. Archer and A. Whitechapel. *Inside C#*. Microsoft Press, 2002.

[2] 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.

[3] E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A complete formal definition of the semantics of C#. Technical report, In preparation, 2003.

[4] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.

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

[6] E. Börger and R. F. Stärk. An ASM model for C# threads. Technical report, Computer Science Department, ETH Zürich, 2003.

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

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

[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] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley, 2003.

[11] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, 2002.

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

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

[14] J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.

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