A Modular Design for the Common Language Runtime (CLR) Architecture

Report

Author(s):
Fruja, Nicu Georgian

Publication date:
2005

Permanent link:
https://doi.org/10.3929/ethz-a-006787891

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technical Report / ETH Zurich, Department of Computer Science 493

A Modular Design for the Common Language Runtime (CLR) Architecture

Nicu G. Fruja
Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
fruja@inf.ethz.ch

Abstract

This paper provides a modular high-level design of the Common Language Runtime (CLR) architecture. Our design is given in terms of Abstract State Machines (ASMs) and takes the form of an interpreter. We describe the CLR as a hierarchy of nine submachines, which correspond to nine submodules into which the Common Intermediate Language (CIL) instruction set can be decomposed.

1 Introduction

This paper is one outcome of a larger project [12] which aims to establish some outstanding properties of C♯ and CLR by mathematical proofs. Examples are the correctness of the CLR bytecode verifier and the type safety of C♯ (along the lines of the correctness proof [11] for the definite assignment rules). As part of this effort, an ASM model has been developed in [7] to formalize the semantics of C♯. To validate this model, we have refined it and made executable (see [13]) in AsmL [15].

The CLR is the runtime environment for executing .NET applications. A single .NET application may consist of several different languages. Accordingly, the CLR has to support any language compiler intended for the .NET platform. We assume the reader to be knowledgeable about or at least to have a rough understanding of the CLR virtual machine.

We define an abstract interpreter in terms of an ASM model for the CIL language executed by the CLR virtual machine which includes most of the constructs which deal with the interpretation of the procedural, object-oriented and non-verifiable constructs of the .NET CLR. The inputs of the interpreter are CIL programs whose code consists of bytecode instructions. Our interpreter is a *trustful* machine, i.e. it does *not* check the instructions before the execution to satisfy constraints about types, resource bounds, etc. In order to check the faithfulness with respect to the CLR of the modeling decisions we had to take here, we made a series of experiments with the CLR. Another way to test the internal correctness of the model presented in this paper and its conformance to the experiments with the CLR is provided through an executable version implemented in AsmL [15]. Upon completion of the AsmL implementation of the entire CLR model, the full details will be made available in [14].

In [6], a similar ASM model is developed for the Java Virtual Machine (JVM). However, the bytecode run in CLR results not only from the compilation of C♯ but of all .NET compatible languages such as Visual Basic, C++, VBScript, JScript, COBOL, Component Pascal, Modula 2, Eiffel etc. Thus, the CIL instruction set is designed with the objective of supporting multiple languages, and thus needs to support all of the constructs of what Microsoft calls the *Virtual Object System*. [5] describes briefly the differences between the JVM and CLR virtual machines.

An interested reader can find in this paper many other differences. Accordingly, the development of the CLR model becomes more complex than in case of JVM [6].

A type system for a fragment of CIL is developed in [4]. The main theorem proved in [4], asserts type safety. Many key aspects are however omitted in the object model they consider: `null` objects, global fields and methods, static fields and methods (and implicitly the type initialization process). Moreover, their instruction set omits: local variables instructions, arithmetic instructions, arbitrary branching instructions, jumping instructions, tail calls prefix.

The main technical contributions of this paper are the formalizations of the following critical (different wrt JVM [6]) features: *typed evaluation stack*, *memory allocation and de-allocation*, *tail method calls*, *call-by-reference mechanism*, *exception handling mechanism*, *pointer handling*, *value class instance handling*, *typed reference handling*, *method pointer handling*. Similarly as in [6], CIL is described as a hierarchy of nine sublanguages, which correspond to nine submodules into which the CLR can be decomposed: CLR$_{\mathcal{I}}\subset$ CLR$_{\mathcal{C}}\subset$ CLR$_{\mathcal{O}}\subset$ CLR$_{\mathcal{E}}\subset$ CLR$_{\mathcal{P}}\subset$ CLR$_{\mathcal{VC}}\subset$ CLR$_{\mathcal{TR}}\subset$ CLR$_{\mathcal{MP}}\subset$ CLR$_{\mathcal{RM}}$. For each such submodule CLR$_{\mathcal{L}}$ we build a submachine CLR$_L$ which is a conservative extension of its predecessor. The model for the whole CLR is given by the last submachine, i.e. CLR$_{RM}$. We omit here bytecode instructions for monitors.

The exceptions submodule CLR$_{\mathcal{E}}$ including the CLR exception handling mechanism and its analysis is also postponed to a separate paper [2] where the use of ASMs clarified the numerous issues concerning the exception handling which are left open in the ECMA standard [1]. Also, the model in the present paper helps us to discover a *mistake* in the CLR implementation concerning the value class initialization (see Section 2.7).

Since the intuitive understanding of the ASMs machines as pseudo-code over abstract data structures is sufficient for the comprehension of the ASM formalism used throughout this paper, we abstain here from repeating the formal definition of ASMs which can be found in the AsmBook [8]. However, for the readers convenience we summarize here the most important concepts and notations that are used in the ASMs throughout this paper. An abstract state of an ASM is given by a set of dynamic functions. Nullary dynamic functions correspond to ordinary state variables. Formally all functions are total. They may, however, return the special element *undef* if they are not defined at an argument. In each step, the machine updates in parallel some of the functions at certain arguments. The updates are programmed using transition rules P, Q with the following meaning:

$f(s) := t$	update f at s to t
if φ **then** P **else** Q	if φ is true, then execute P, else Q
P Q	execute P and Q in parallel
let $x = t$ **in** P	assign t to x and then execute P
P **seq** Q	execute P and then Q
forall x **in** A **do** P	execute in parallel P for every $x \in A$
choose x **with** φ **do** P	choose an x satisfying φ, and execute P
P **or** Q	execute P or Q

The remainder of the paper proceeds as follows. Section 2 defines the sequence of the five successively extended machines. A global view of the CLR virtual machine is given in Section 2.1 together with a short description of the considered bytecode instructions. Section 2.2 introduces a typed stack machine CLR$_{\mathcal{I}}$ with instructions required for the compilation of imperative programs of a *while* language. CLR$_{\mathcal{I}}$ is extended to CLR$_{\mathcal{C}}$ in Section 2.3 by including instructions used for compilation of static features of classes. Section 2.4 defines an object-based machine CLR$_{\mathcal{O}}$ which supports instructions for object oriented features. The exceptions are introduced in the machine CLR$_{\mathcal{E}}$ defined in Section 2.5. This section introduces also the complex CLR exception handling mechanism. The machine CLR$_{\mathcal{P}}$ defined in Section 2.6 extends CLR$_{\mathcal{E}}$ by adding instructions for dealing with pointers. The machine CLR$_{\mathcal{VC}}$ defined in Section 2.7 pro-

Fig. 1 The CIL instructions

CLR$_{\mathcal{I}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \mathit{Execute}(\mathit{Op}) \\ & \mid & \mathit{Const}(\mathit{ConstType},\mathit{Literal}) \\ & \mid & \mathit{LoadLoc}(\mathit{Local}) \\ & \mid & \mathit{StoreLoc}(\mathit{Local}) \\ & \mid & \mathit{Branch}(\mathit{Pc}) \\ & \mid & \mathit{Cond}(\mathit{Op},\mathit{Pc}) \\ & \mid & \mathit{Dup} \\ & \mid & \mathit{Pop} \\ & \mid & \mathit{Halt} \end{array}$$

CLR$_{\mathcal{C}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{LoadStatic}(\mathit{Type},\mathit{FRef}) \\ & \mid & \mathit{StoreStatic}(\mathit{Type},\mathit{FRef}) \\ & \mid & \mathit{LoadArg}(\mathit{Arg}) \\ & \mid & \mathit{StoreArg}(\mathit{Arg}) \\ & \mid & \mathit{Call}(\mathit{TailCall},\mathit{Type},\mathit{MRef}) \\ & \mid & \mathit{Return} \end{array}$$

CLR$_{\mathcal{O}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{NewObj}(\mathit{MRef}) \\ & \mid & \mathit{LoadField}(\mathit{Type},\mathit{FRef}) \\ & \mid & \mathit{StoreField}(\mathit{Type},\mathit{FRef}) \\ & \mid & \mathit{CallVirt}(\mathit{TailCall},\mathit{Type},\mathit{MRef}) \\ & \mid & \mathit{CastClass}(\mathit{Class}) \\ & \mid & \mathit{IsInstance}(\mathit{Class}) \\ & \mid & \mathit{Jmp}(\mathit{MRef}) \\ & \mid & \mathit{NewArray}(\mathit{Type}) \\ & \mid & \mathit{LoadLength} \\ & \mid & \mathit{LoadElem}(\mathit{ArrayElemType}) \\ & \mid & \mathit{StoreElem}(\mathit{ArrayElemType}) \end{array}$$

CLR$_{\mathcal{E}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{Throw} \\ & \mid & \mathit{Rethrow} \\ & \mid & \mathit{EndFilter} \\ & \mid & \mathit{EndFinally} \\ & \mid & \mathit{Leave}(\mathit{Pc}) \end{array}$$

CLR$_{\mathcal{P}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{LoadArgA}(\mathit{Arg}) \\ & \mid & \mathit{LoadLocA}(\mathit{Local}) \\ & \mid & \mathit{LoadStaticA}(\mathit{Type},\mathit{FRef}) \\ & \mid & \mathit{LoadFieldA}(\mathit{Type},\mathit{FRef}) \\ & \mid & \mathit{LoadInd}(\mathit{LoadIndType}) \\ & \mid & \mathit{StoreInd}(\mathit{StoreIndType}) \\ & \mid & \mathit{InitBlock} \\ & \mid & \mathit{CopyBlock} \\ & \mid & \mathit{LocAlloc} \\ & \mid & \mathit{LoadElemA}(\mathit{Class}) \end{array}$$

CLR$_{\mathcal{VC}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{InitObj}(\mathit{ValueType}) \\ & \mid & \mathit{CopyObj}(\mathit{ValueType}) \\ & \mid & \mathit{Box}(\mathit{ValueType}) \\ & \mid & \mathit{Unbox}(\mathit{ValueType}) \\ & \mid & \mathit{SizeOf}(\mathit{ValueType}) \end{array}$$

CLR$_{\mathcal{TR}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{MkRefAny}(\mathit{Type}) \\ & \mid & \mathit{RefAnyType} \\ & \mid & \mathit{RefAnyVal}(\mathit{Type}) \end{array}$$

CLR$_{\mathcal{MP}}$ instructions

$$\begin{array}{lcl} \mathit{Instr} & = & \ldots \\ & \mid & \mathit{LoadFtn}(\mathit{MRef}) \\ & \mid & \mathit{LoadVirtFtn}(\mathit{MRef}) \\ & \mid & \mathit{CallI}(\mathit{TailCall},\mathit{Sig}) \end{array}$$

vides instructions for dealing with value class instances. The typed references represent the novelty introduced by the machine CLR$_{\mathcal{TR}}$ in Section 2.8. The method pointers are handled by the instructions specific to CLR$_{\mathcal{MP}}$ which is defined in Section 2.9. Section 3 provides the topmost machine, i.e. CLR$_{\mathcal{RM}}$ and describe how one can handle runtime managed methods. Section 4 concludes.

2 The CLR virtual machine

2.1 The overall picture

The real CLR has approximately 200 instructions. Most of them are specified in Fig. 1 as elements of the successively extended universe *Instr*. The real CLR has also the instructions `break` and `nop` which we do not model.

In order to simplify the presentation we assume as [6, §9.1] does that we have also[1] a *Halt* instruction whose occurrence stops the machine execution. One can obtain the real CLR instructions if one extends the parameter universes we describe below. The universe *Op* contains operators, *Local* local variables and *Pc* program counters, i.e. code offsets. The last two universes are synonyms for the universe $\mathbb{N}$ of natural numbers. The types of the constants are declared by the universe *ConstType* – these types are incorporated in the CLR `ldc` instructions. The constants are depicted as literals, i.e. elements of the universe *Literal*. *Type* denotes types, *Arg* method arguments and *TailCall* special boolean flags used for method calls. The universes *FRef* and *MRef* uniquely describe field and method references. *Class* stands for the universe of classes, while *ValueClass* denotes only the value classes. The possible types for an array element are described by the universe *ArrayElemType*. *LoadIndType* and *StoreIndType* denote the type of a value indirectly loaded and stored, respectively from and into the memory. *ValueType* represents the universe of value types which includes the value classes described by *ValueClass* and the built-in value types, i.e. the numeric types.

Some instructions are never verifiable, i.e. a program containing such instructions will always fail the bytecode verification. The other instructions are either always verifiable or verifiable under certain conditions which we do not detail here (this is part of future work – see Section 4). However, we specify all the instructions independent on their verifiability status.

Naming conventions We summarize here the major naming conventions used in the rest of the paper. Thus, t will represent a type, val a value, r an object reference and adr a memory address, c a class name, vc a value class name, vt a value type, tr a typed reference, fp a function pointer, sig a method signature, f a field name and m a method name.

2.2 The CLR$_{\mathcal{I}}$ submachine

The CLR$_I$ machine is a typed stack machine which supports the instructions necessary to implement a so-called imperative *while* language.

Environment and State The list of bytecode instructions of the current method is maintained in $code : List(Instr)$. CLR$_{\mathcal{I}}$ is dealing only with a single method whose local variable types are given by the list *locTypes*. The universe *DivOp* consists of the division operators `div`, `div.un`, `rem`, `rem.un`. Some operators perform overflow checks; examples are `add.ovf.un` and `sub.ovf.un`. We denote by *OvfOp* the universe of these operators. Upon these definitions, the following set relations hold: $DivOp \subset Op$ and $OvfOp \subset Op$. The universe *ConstType* describes the types of the constants CLR can deal with.

$ConstType =$ `int32` | `int64` | `float32` | `float64`

The dynamic state of the CLR$_{\mathcal{I}}$ consists of a frame containing a program counter pc, local variable addresses *locAdr* and an evaluation stack *evalStack*. The pc runs over the set of natural numbers $Pc = \mathbb{N}$. The *locAdr* carries the addresses of the local variables and not their values. Although the addresses are not needed for CLR$_{\mathcal{I}}$, they are later addressable with the instructions added in CLR$_{\mathcal{P}}$. The universe of addresses *Adr* is the interval $Adr = 0 .. maxAdr$ bounded by a non-negative integer *maxAdr* which depends on the target architecture. The *mem* is used to store values into the memory locations. The second column of the following declaration defines the initial values of the dynamic functions in the first column:

$pc : Pc$	$pc = 0$
$locAdr : Map(Local, Adr)$	$locAdr(n) \in Adr, \forall n \in Local$
$evalStack : List(Val \times CLRStackType)$	$evalStack = [\,]$
$mem : Map(Adr, Val \cup \{undef\})$	$mem(adr) = undef, \forall adr \in Adr$

Typed evaluation stack The *evalStack* is specified as a list of typed values. This is a crucial difference wrt the untyped JVM operand stack. The latter is made of uniform 32-bits wide

[1] *Halt* is not a real CIL instruction

locations modulo some issues with the atomicity of pushes of 64-bit quantities. The values are described as elements of *Val*. The CLR requires values on the *evalStack* to be of types described by *CLRStackType*. The type `native int` describes both signed and unsigned integers and becomes efficient when the target machine architecture is not known until run-time when CLR maps `native int` to the natural size of the specific architecture: `int32` on a 32-bits architecture and `int64` on a 64-bits architecture. The floating point numbers are represented on the *evalStack* using an internal floating-point type `F`. The CLR supports generic instructions such as `add`, `div` because, in contrast to JVM, the CIL code has been designed for JIT compilation and not to be interpreted. Our interpreter needs to track the types of the values on the *evalStack*, in particular, for executing generic operators. The types referred by *CLRStackType* are the types as tracked by the CLR rather than the more detailed types used by the CLR bytecode verifier. The *CLRStackTypes* are used for the following purposes:

- to define the semantic function *CLRResVal*;
- to specify the cases when an operator can throw an exception;
- to determine if a bytecode program is *valid* - the validity condition is a necessary condition for a program to be verifiable (see [1, Partition III];

The bytecode verifier tests requirements for *correct generated* CIL bytecode and also specific verification conditions (the formalization of the bytecode verifier is future work).

$\mathit{CLRStackType} = \texttt{int32} \mid \texttt{int64} \mid \texttt{native int} \mid \texttt{F}$

The *valuesOf* selects the value component of a list of *evalStack* slots, i.e. *valuesOf* applied to $[(val_1, t_1), \ldots, (val_n, t_n)]$ returns $[val_1, \ldots, val_n]$. Similarly, *typesOf* selects the type component of a list of *evalStack* slots. The *CLRTypeOf* extends "upwards" a type to a *CLRStackType*. Thus, every integral subtype of `int32` is mapped to `int32`, `native int` is mapped to itself, every other integral type is mapped to `int64`, and the floating types are mapped to `F`.

Beside the usual list operations (e.g. *push*, *pop*, *take*, "$\cdot$")[2], we use different operations for dealing with the *evalStack*: $split(evalStack, n)$ splits off the last n slots of the *evalStack*. More exactly, $split(evalStack, n)$ is the pair $(evalStack', ns)$ of two lists where $evalStack' \cdot ns = evalStack$ and $length(ns) = n$. Similarly, $getVals(evalStack, n)$ splits off the last n slots of the *evalStack* by retrieving *only* their values.

Rules The ASM rules in Fig. 2 describe the dynamic semantics of $\mathrm{CLR}_{\mathcal{I}}$. The machine CLR_I fires the $\textsc{exec}\mathrm{CLR}_I$ rules for the current instruction $code(pc)$.

Remark We assume that all the submachines are guarded by $halt = undef$. If *halt* gets assigned a value, the machines will stop executing.

$Execute(op)$ takes the topmost $opNo(op)$ values of the *evalStack* where $opNo(op)$ returns the number of operands of the operator *op*. If there is no exception case, the result of the semantic function $CLRResVal(op, slots)$ is loaded on the *evalStack*. The result type *CLRResType* of all the operators is defined by the ECMA standard [1, Partition III,§1.5] and is a function of the operator and of the operands' types. The cases when an exception is thrown are: (a) division by zero for operators of integral types; (b) operations that perform an overflow check and whose results cannot be represented in the result type; (c) values that are not "normal" numbers are checked for finiteness or `div`ision/`rem`ainder operations are executed for a minimal value of an integral type and -1. In the following formalizations, *vals* stands for $valuesOf(slots)$ and *types* for $typesOf(slots)$.

$$\begin{aligned} ExceptionCase(op, slots) \quad \Leftrightarrow \quad & DivByZeroCase(op, slots) \lor OverflowCase(op, slots) \\ \lor \quad & InvNrCase(op, slots) \end{aligned}$$

[2]The "$\cdot$" denotes the append operation for lists.

Fig. 2 The execution of CLR$_\mathcal{I}$ instructions

```
$\mathrm{CLR}_I \equiv \textsc{exec}\mathrm{CLR}_I(code(pc))$

$\textsc{exec}\mathrm{CLR}_I(instr) \equiv$ **match** $instr$
  $Execute(op) \rightarrow$
    **let** $(evalStack', slots) = split(evalStack, opNo(op))$ **in**
      **if** $\neg ExceptionCase(op, slots)$ **then**
        **let** $(val, t) = (CLRResVal(op, slots), CLRResType(op, typesOf(slots)))$ **in**
          $evalStack := evalStack' \cdot [(val, t)]$
          $pc := pc + 1$
  $Const(t, lit) \rightarrow$  $evalStack := evalStack \cdot [(lit, CLRTypeOf(t))]$
                   $pc := pc + 1$
  $LoadLoc(n) \rightarrow$ **if** $zeroInit(meth)$ **then**
                  **let** $t = locTypes(n)$ **in**
                    $evalStack := evalStack \cdot [(memVal(locAdr(n), t), CLRTypeOf(t))]$
                    $pc := pc + 1$
  $StoreLoc(n) \rightarrow$ **let** $(evalStack', [val]) = getVals(evalStack, 1)$ **in**
                   $\textsc{WriteMem}(locAdr(n), locTypes(n), val)$
                   $evalStack := evalStack'$
                   $pc := pc + 1$
  $Branch(t) \rightarrow$ $pc := t$
  $Cond(op, t) \rightarrow$ **let** $(evalStack', slots) = split(evalStack, opNo(op))$ **in**
                   $evalStack := evalStack'$
                   $pc :=$ **if** $CLRResVal(op, slots)$ **then** $t$ **else** $pc + 1$
  $Dup \rightarrow$ **let** $(evalStack', [(val, t)]) = split(evalStack, 1)$ **in**
           $evalStack := evalStack' \cdot [(val, t), (val, t)]$
           $pc := pc + 1$
  $Pop \rightarrow$ $pop(evalStack)$
         $pc := pc + 1$
```

$$\begin{aligned}
DivByZeroCase(op, slots) \;&\Leftrightarrow\; op \in DivOp \wedge vals(1) = 0\\
&\quad\wedge\; types(i) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\},\ i = 0, 1\\
OverflowCase(op, slots) \;&\Leftrightarrow\; op \in OvfOp\\
&\quad\wedge\; Overflow(CLRResVal(op, slots), CLRResType(op, types))\\
InvNrCase(op, slots) \;&\Leftrightarrow\; (op = \texttt{ckfinite} \wedge vals(0) \in \{\texttt{NaN}, \texttt{+infinity}, \texttt{-infinity}\})\\
&\quad\vee\; (op \in \{\texttt{div}, \texttt{rem}\} \wedge vals(0) = min(types(0)) \wedge vals(1) = -1\\
&\qquad \wedge\, types(i) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\},\ i = 0, 1)
\end{aligned}$$

The $Const(t,lit)$ instruction pushes the constant lit of type t on top of the $evalStack$. However, the type of the stack location is determined with $CLRTypeOf$. If the $zeroInit$ flag of current method is set, the $LoadLoc$ instruction loads the value of a local variable. The value of the local variable n of the declared type t is determined using the derived function $memVal$: $Map(Adr, Val)$ applied to the address of n and t. The $memVal$ builds up the value of a given type stored in memory at a certain address. In CLR$_\mathcal{I}$, $memVal(adr, t) := mem(adr)$. The type t becomes relevant when we refine the universe Val in CLR$_\mathcal{C}$. $StoreLoc$ writes the value of the topmost slot into the memory at the local variable's address. The "write in memory" is defined through the WriteMem that also considers the type of the stored value although this is not needed in CLR$_\mathcal{I}$.

$\textsc{WriteMem}(adr, t, val) \equiv mem(adr) := val$

The jump instruction $Branch(t)$ simply sets the pc to t. A conditioned jump can be executed with $Cond$. If the operator op returns $True$, then the pc is set to t, otherwise the pc is incremented. The topmost slot of the $evalStack$ can be duplicated and popped off with the instructions Dup and Pop, respectively.

2.3 The CLR$_\mathcal{C}$ submodule

CLR$_\mathcal{C}$ extends CLR$_\mathcal{I}$ by instructions which deal with "read"/"write" static fields, "read"/"write" method arguments and "call of"/"return from" static methods.

Environment and State *FRef* and *MRef* consist of field and method references, respectively. The field references are pairs of class and field names, while the method references are triples of class names, method names and signatures. *Class* consists in CLR$_\mathcal{C}$ only of the object classes defined by the universe *ObjClass* (in CLR$_\mathcal{VC}$ we add also value classes). *Field* and *Method* are universes that stand for field and method identifiers, respectively. *Sig* specifies the universe of "stand alone *sig*natures". A *sig*nature includes not only a return type, number, order and types of the parameters but also information about the *calling convention* to be used when invoking the corresponding method. Method arguments are specified as natural numbers and are elements of the universe *Arg*. The real CLR instructions for calling methods (see also CLR$_\mathcal{O}$) may be prefixed by `tail` whose presence is indicated by a boolean flag – element of *TailCall* – in the abstract *Call* instruction.

$$\begin{array}{lll} \mathit{FRef} = \mathit{Class} \times \mathit{Field} & \mathit{Class} = \mathit{ObjClass} & \mathit{TailCall} = \mathit{Bool} \\ \mathit{MRef} = \mathit{Class} \times \mathit{Method} \times \mathit{Sig} & \mathit{Arg} = \mathbb{N} & \end{array}$$

The static function *code* : $\mathit{Map}(\mathit{MRef}, \mathit{List}(\mathit{Instr}))$ associates to every method reference the list of bytecode instruction contained in its body. The set of static and instance fields of a class is obtained by applying to the class name the functions *statFields* and *instFields*, respectively[3]. With these definitions, it becomes obvious how the predicates *static* and *instance* decide if a field reference is to a static or an instance field (see CLR$_\mathcal{O}$). The function *type* applied to a field reference returns the declared type of the field. The functions *paramTypes* and *retType* selects the list of parameter types and the return type of a method signature. We often use these two functions as applied to method references to determine the corresponding data. The functions *classNm* and *methNm* select the class name and the method name of a method reference, respectively. The function *paramNo* is derived from *paramTypes* and assigns to a signature the length of the parameter types list. The *locTypes* assigns to every method reference the list of its local variable types ("locals signature"). The predicates *static* and *instance* defined for (method) signatures decide whether the calling convention embedded in the signature refers to a static or instance signature. We use these predicates many times also applied directly to method references, but they are actually computed as being applied to the signatures. The *zeroInit* is a flag in the method headers which indicates whether the local variables should be automatically initialized to zero by the CLR.

$$\begin{array}{ll} & \mathit{retType} : \mathit{Map}(\mathit{Sig}, \mathit{Type}) \\ \mathit{type} : \mathit{Map}(\mathit{FRef}, \mathit{Type}) & \mathit{paramTypes} : \mathit{Map}(\mathit{Sig}, \mathit{List}(\mathit{Type})) \\ \mathit{statFields} : \mathit{Map}(\mathit{Class}, \mathcal{P}(\mathit{FRef})) & \mathit{classNm} : \mathit{Map}(\mathit{MRef}, \mathit{Class}) \\ \mathit{instFields} : \mathit{Map}(\mathit{Class}, \mathcal{P}(\mathit{FRef})) & \mathit{methNm} : \mathit{Map}(\mathit{MRef}, \mathit{Meth}) \\ & \mathit{locTypes} : \mathit{Map}(\mathit{MRef}, \mathit{List}(\mathit{Type})) \\ & \mathit{zeroInit} : \mathit{Map}(\mathit{MRef}, \mathit{Bool}) \end{array}$$

In CLR$_\mathcal{C}$, the procedural abstraction is added in the form of static methods. Unlike in CLR$_\mathcal{I}$ where we had a single method frame, in CLR$_\mathcal{C}$ we have a stack *frameStack* of call frames described by *Frame*.

$$\begin{array}{l} \mathit{Frame} = \mathit{Pc} \times \mathit{Map}(\mathit{Local}, \mathit{Adr}) \times \mathit{Map}(\mathit{Arg}, \mathit{Adr}) \times \mathcal{P}(\mathit{Adr}) \times \mathit{List}(\mathit{Val} \times \mathit{CLRStackType}) \times \mathit{MRef} \\ \mathit{frameStack} : \mathit{List}(\mathit{Frame}) \end{array}$$

A frame in CLR$_\mathcal{C}$ is enriched with more information than in CLR$_\mathcal{I}$. The frame components are, in order, the following: a program counter *pc*, local variables addresses *locAdr*, arguments addresses *argAdr* : $\mathit{Map}(\mathit{Arg}, \mathit{Adr})$, the set of stack-allocated addresses *StackAdr* (including also the addresses allocated for the frames on the *frameStack*), an evaluation stack *evalStack* and a method reference *meth* : *MRef*. As in case of *locAdr*, the *argAdr* holds the arguments' addresses and not their values. Although these addresses are not useful in CLR$_\mathcal{C}$, they become addressable

[3]We denote by $\mathcal{P}(A)$ the power set of A.

in $\mathrm{CLR}_{\mathcal{P}}$. Unlike CLR, JVM does not have separate instructions for method arguments. We model in a simple manner also the memory allocation (see the end of this section where we provide an abstract specification of the memory management). Upon exiting a method, the addresses allocated for the method's *evalStack* have to be deallocated. Therefore, it is crucial to "remember" the set of stack-allocated addresses for the invoker frame. We extend the stipulations for the initial state as follows:

$locAdr = \emptyset \qquad argAdr = \emptyset \qquad StackAdr = \emptyset \qquad meth =$ `Object::.entrypoint`
$frameStack = [\,]$

Assumption As specified by the standard, the entrypoint has to be a static method. If this method is declared by a class that is not marked `beforefieldinit`, then the type initializer of this class has to be executed before the entrypoint method, i.e. the entrypoint is not the first method to execute. In order to solve this inconvenience, we assume as suggested in [6, §10.1] that the entrypoint is not the method marked with in the IL code with `entrypoint` but a method declared by `Object` that has the code:

$Call(False,$ `void`$, CIL\_entrypoint)$
$Halt$

We denote by *frame* the currently executed frame:

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

Note that we separate the current *frame* from the stack of frames, i.e. we do not include *frame* in the *frameStack*.
We consider the approach from [6] to separate the methods transfer and the execution of method bodies. We introduce a switch machine SWITCHCLR that is responsible for the methods transfer. The universe *Switch* defines the states of this machine:

$$\begin{aligned} Switch \quad = \quad & Noswitch \mid Invoke(TailCall, MRef, List(Val)) \mid InitClass(Class) \\ \mid \quad & Result(List(Val \times CLRStackType)) \end{aligned}$$

The current state of the submachine SWITCHCLR is specified by the dynamic function $switch : Switch$ whose initial value is *Noswitch*.
Unlike in JVM, in CLR, beside static fields and methods, there are also global fields and methods (declared outside of any type). The CLR defines a class, named `<Module>`, that has as members all the global fields and methods, which does not have a base type and does not implement any interfaces [1, Partition II,§9.8]. Accordingly, we treat the global members exactly in the same way as we treat the static members. $globals : Map(FRef, Adr)$ holds the addresses of all static fields (including the global fields).

The locations for the static fields declared by a class c are created when c is loaded but they are initialized when the c is initialized [1, Partition I,§8.11.2]. To simplify the presentation, we do not model the types loading. Therefore we assume that these locations are already allocated upon the start of the machine $\mathrm{CLR}_{\mathcal{C}}$. To simplify the presentation, we do not model this allocation – it is however obvious that there is always sufficient space to be allocated at the beginning of the execution.
Class initialization The ECMA standard imposes several rules in [1, Partition I,§8.9.5] concerning the class initialization (for both object classes and value classes as we will see in $\mathrm{CLR}_{\mathcal{VC}}$). The class initialization implies the execution of its initializer. A class may have or may not have an initializer. In the latter case, the CLR creates one which usually contains assignments to the static fields. In this case, the CLR marks the class with the attribute `beforefieldinit`. If a class is not marked with `beforefieldinit`, then the initializer is executed at the first access to any static or instance field (see $\mathrm{CLR}_{\mathcal{O}}$) of that class, or the first invocation of any static, instance or virtual method (see $\mathrm{CLR}_{\mathcal{O}}$) of that class. If the class is marked with `beforefieldinit`, the invocation of a static method (declared by the class) does not trigger the initialization.

In such a case, the initialization is triggered only by a static field access. Let us denote by $\mathit{beforefieldinit} : \mathit{Map}(\mathit{Class}, \mathit{Bool})$ the predicate that specifies what classes are marked with `beforefieldinit`. The universe *ClassState* specifies the initialization state of a class: before being initialized, a class is in state *Linked*, while, following the initialization, a class is *Initialized*. *classState* keeps track of the initialization state of classes. In the initial state, all the classes are *Linked* except `Object` and `<Module>` which are *Initialized*:

$$\begin{aligned}
&\mathit{ClassState} = \mathit{Linked} \mid \mathit{Initialized}\\
&\mathit{classState} : \mathit{Map}(\mathit{Class}, \mathit{ClassState})\\
&\mathit{classState}(c) = \mathit{Linked}\,, \forall c \in \mathit{Class} \setminus \{\texttt{Object}, \texttt{<Module>}\}\\
&\mathit{classState}(\texttt{Object}) = \mathit{classState}(\texttt{<Module>}) = \mathit{Initialized}
\end{aligned}$$

CLR vs. JVM Unlike in JVM, execution of any type's initializer in CLR does not trigger automatic execution of any initializer methods defined by its base type.

The predicate *initialized* is derived from *classState*. Thus $\mathit{initialized}(c)$ holds for a class c if $\mathit{classState}(c) = \mathit{Initialized}$. It is useful to define also the predicate *reqinit* that is checked every time a method is invoked; a c's method invocation requires the initialization of c if c is neither initialized nor marked with `beforefieldinit`:

$$\mathit{reqinit}(c) \Leftrightarrow \neg \mathit{initialized}(c) \wedge \neg \mathit{beforefieldinit}(c)$$

Rules The ASM rules for $\mathrm{CLR}_{\mathcal{C}}$ are defined in Fig. 3. The machine $\mathrm{CLR}_{\mathcal{C}}$ executes the macro $\mathit{execScheme}_{C}$ which is parameterized by the machines $\textsc{exec}\mathrm{CLR}_{\mathcal{C}}$ and $\textsc{switch}\mathrm{CLR}$. The macro $\mathit{execScheme}_{C}$ is defined as follows[4]. If *switch* is set, i.e. it has a value other than *Noswitch*, then the control is passed to the machine $\textsc{switch}\mathrm{CLR}$. Otherwise, as a consequence of the `beforefieldinit` semantics, either a `beforefieldinit` class is initialized through $\textsc{InitializeClass}$ or $\textsc{exec}\mathrm{CLR}$ fires a rule for the current instruction. We use the abbreviation "**or**" as defined in [8, §2.2.5] for the special case of non-deterministic choice among two rules. $\textsc{InitializeClass}$ arbitrarily chooses a `beforefieldinit` class that is not yet initialized. It then passes the control to the $\textsc{switch}\mathrm{CLR}$ to initialize the class.

```
InitializeClass ≡ choose $c \in \mathit{Class}$ with $\neg\mathit{initialized}(c) \wedge \mathit{beforefieldinit}(c)$ do
                      $\mathit{switch} := \mathit{InitClass}(c)$
```

The following explanations are for the $\textsc{exec}\mathrm{CLR}$ rules assuming in cases of field accesses and method calls, that no class initialization is required (if one needs to initialize a class, the *switch* is updated to *InitClass*). The type t in $\mathit{LoadStatic}(t, c{::}f)$ is the declared type of the field reference $c{::}f$. The same explanation applies for $\mathit{StoreStatic}(t, c{::}f)$. In case of $\mathit{Call}(\_, t, c{::}m)$, t denotes the return type of the considered method reference. $\mathit{LoadStatic}(t,c{::}f)$ pushes on the *evalStack* the value of the field f stored at the address $\mathit{globals}(c{::}f)$. The value of the topmost *evalStack* slot is stored by *StoreStatic* into the address of $c{::}f$. To formalize *LoadArg* and *StoreArg*, we need to determine the types of the current method arguments: $\mathit{argTypes} : \mathit{Map}(\mathit{MRef}, \mathit{List}(\mathit{Type}))$ yields for a method reference the list of argument types. We denote by *argNo* the length of *argTypes*. For every static method reference $c{::}m$, *argTypes* is defined[5] as follows:

$$\mathit{argTypes}(c{::}m) = \mathit{paramTypes}(c{::}m)$$

Note We assume that for each of the functions *code*, *locTypes*, *argTypes*, there is a derived function *having the same name*, that suppresses the method reference and abbreviates the data path to select the corresponding component.

The value of the argument n is loaded on the *evalStack* through $\mathit{LoadArg}(n)$. The instruction $\mathit{StoreArg}(n)$ writes the value of the topmost *evalStack* slot into the address of argument n.

[4]The $\mathit{execScheme}_{C}$ is redefined to $\mathit{execScheme}_{E}$ in $\mathrm{CLR}_{\mathcal{E}}$ while introducing exceptions. If an exception is thrown, the $\mathit{execScheme}_{E}$ passes the control to the exception handling mechanism. Thus, it prevents the execution of the $\textsc{exec}\mathrm{CLR}$ machines.

[5]This definition is refined in $\mathrm{CLR}_{\mathcal{O}}$ and $\mathrm{CLR}_{\mathcal{VC}}$.

Fig. 3 The execution of CLR$_\mathcal{C}$ instructions

```
$\mathrm{CLR}_C \equiv \mathit{execScheme}_C(\textsc{exec}\mathrm{CLR}_C, \textsc{switch}\mathrm{CLR})$

$\mathit{execScheme}_C(\textsc{exec}\mathrm{CLR}, \textsc{switch}\mathrm{CLR}) \equiv$
  $\mathbf{if}\ \mathit{switch} \neq \mathit{Noswitch}\ \mathbf{then}\ \textsc{switch}\mathrm{CLR}$
  $\mathbf{else}\ \textsc{InitializeClass}\ \mathbf{or}\ \textsc{exec}\mathrm{CLR}(\mathit{code}(\mathit{pc}))$

$\textsc{exec}\mathrm{CLR}_C(\mathit{instr}) \equiv$
  $\textsc{exec}\mathrm{CLR}_I(\mathit{instr})$
  $\mathbf{match}\ \mathit{instr}$
    $\mathit{LoadStatic}(t, c{::}f) \rightarrow$
      $\mathbf{if}\ \mathit{initialized}(c)\ \mathbf{then}$
        $\mathit{evalStack} := \mathit{evalStack} \cdot [(\mathit{memVal}(\mathit{globals}(c{::}f), t), \mathit{CLRTypeOf}(t))]$
        $\mathit{pc} := \mathit{pc} + 1$
      $\mathbf{else}\ \mathit{switch} := \mathit{InitClass}(c)$
    $\mathit{StoreStatic}(t, c{::}f) \rightarrow\ \mathbf{let}\ (\mathit{evalStack}', [\mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 1)\ \mathbf{in}$
                               $\mathbf{if}\ \mathit{initialized}(c)\ \mathbf{then}$
                                 $\textsc{WriteMem}(\mathit{globals}(c{::}f), t, \mathit{val})$
                                 $\mathit{evalStack} := \mathit{evalStack}'$
                                 $\mathit{pc} := \mathit{pc} + 1$
                               $\mathbf{else}\ \mathit{switch} := \mathit{InitClass}(c)$
    $\mathit{LoadArg}(n) \rightarrow$
      $\mathbf{let}\ t = \mathit{argTypes}(n)\ \mathbf{in}$
        $\mathit{evalStack} := \mathit{evalStack} \cdot [(\mathit{memVal}(\mathit{argAdr}(n), t), \mathit{CLRTypeOf}(t))]$
        $\mathit{pc} := \mathit{pc} + 1$
    $\mathit{StoreArg}(n) \rightarrow\ \mathbf{let}\ (\mathit{evalStack}', [\mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 1)\ \mathbf{in}$
                       $\textsc{WriteMem}(\mathit{argAdr}(n), \mathit{argTypes}(n), \mathit{val})$
                       $\mathit{evalStack} := \mathit{evalStack}'$
                       $\mathit{pc} := \mathit{pc} + 1$
    $\mathit{Call}(\mathit{tail}, \_, c{::}m) \rightarrow$
      $\mathbf{if}\ \mathit{static}(c{::}m)\ \mathbf{then}$
        $\mathbf{if}\ \neg\mathit{reqinit}(c)\ \mathbf{then}$
          $\mathbf{let}\ (\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{argNo}(c{::}m))\ \mathbf{in}$
            $\mathit{evalStack} := \mathit{evalStack}'$
            $\mathit{switch} := \mathit{Invoke}(\mathit{tail}, c{::}m, \mathit{vals})$
        $\mathbf{else}\ \mathit{switch} := \mathit{InitClass}(c)$
    $\mathit{Return} \rightarrow\ \mathbf{let}\ \mathit{slots} = \mathit{take}(\mathit{evalStack}, n)\ \mathbf{in}\ \mathit{switch} := \mathit{Result}(\mathit{slots})$
                 $\mathbf{where}\ n = \mathbf{if}\ \mathit{retType}(\mathit{meth}) = \texttt{void}\ \mathbf{then}\ 0\ \mathbf{else}\ 1$
```

CLR vs. JVM Unlike CLR, the JVM does not have separate instructions for dealing with method arguments. The reason is that the arguments' values appear as the first values in the local variables array of a method.

For calling a method, a *Call* takes the necessary number of arguments from the *evalStack* and transfers the control to SWITCHCLR. It forwards the boolean information concerning a possible tail call through the switch value *Invoke* passed to SWITCHCLR. The *Return* takes from the *evalStack* zero or one value depending on the return type of the current method and transfers the control to SWITCHCLR together with the returned value (if any).

CLR vs. JVM In JVM, there are many *Return* instructions for terminating the execution of a method. The JVM offers specialized instructions for the possible return types.

The switch machine The rules of the submachine SWITCHCLR are presented in Fig. 4. The rule *Invoke*$(\mathit{tail}, c{::}m, \mathit{args})$ handles the context transfer from the current method to the method $c{::}m$. The CLR supports tail calls (this is a crucial difference wrt JVM) since there are .NET languages like Haskell, Scheme, Mercury and SMLNET, where the recursion is the only way to express repetition. If the prefix `tail` is attached to a call instruction, i.e. *tail* is *True*, then the caller's stack frame is discarded through POPFRAME prior making the call.

Fig. 4 The $\textsc{switch}$CLR machine

```
$\textsc{switch}\text{CLR} \equiv$ **match** $switch$
  $Invoke(tail, c{::}m, args) \rightarrow$ **if** $tail$ **then** $\textsc{PopFrame}(0, [\,])$
                                  **seq**
                                     **if** $SpaceFor([sizes]) \neq \emptyset$ **then**
                                        $push(frameStack, frame)$
                                        $\textsc{SetFrame}(c{::}m, args)$
                                        $switch := Noswitch$
                               **where** $sizes = sizeOf^*(argTypes(c{::}m) \cdot locTypes(c{::}m))$
  $InitClass(c) \rightarrow$ **if** $classState(c) = Linked$ **then**
                       **if** $SpaceFor([sizes]) \neq \emptyset$ **then**
                          $classState(c) := Initialized$
                          $\textsc{InitStatFields}(c)$
                          $push(frameStack, frame)$
                          $\textsc{SetFrame}(c{::}\texttt{.cctor}, [\,])$
                          $switch := Noswitch$
                     **where** $sizes = sizeOf^*(locTypes(c{::}\texttt{.cctor}))$
  $Result(slots) \rightarrow$ **if** $methNm(meth) = \texttt{.cctor}$ **then** $\textsc{PopFrame}(0, [\,])$
                       **else** $\textsc{PopFrame}(1, slots)$
                       $switch := Noswitch$
```

The following explanations are for the macro PopFrame. If the current method is a `.cctor`, i.e. it has been implicitly called and *vals* is $[\,]$, then the current frame is discarded and the invoker frame becomes the current frame. If the current method is not a `.cctor`, then the current frame is given by the invoker frame with *vals* pushed on the *evalStack* and the *pc* incremented by 1. The memory allocated on the stack for the current method is reclaimed through DeAllocMem[6].

```
$\textsc{PopFrame}(k, slots) \equiv$
  **let** $(frameStack', [(pc', locAdr', argAdr', StackAdr', evalStack', meth')])$
    $= split(frameStack, 1)$ **in**
      $pc$          $:= pc' + k$
      $locAdr$      $:= locAdr'$
      $argAdr$      $:= argAdr'$
      $evalStack$   $:= evalStack' \cdot slots$
      $meth$        $:= meth'$
      $frameStack$  $:= frameStack'$
      $\textsc{DeAllocMem}(StackAdr')$
```

Since the first method on *frameStack* is always our `Object::entrypoint`, which does not have a *Return*, the *frameStack* is non-empty whenever PopFrame is invoked and consequently the *split* in the definition of PopFrame always succeeds.

In the second step of the *Invoke* rule, assuming there is enough memory space to be allocated for the runtime stack of $c{::}m$ (see the paragraph on the abstract memory management at the end of this section for the definitions of *SpaceFor* and $sizeOf^*$), the frame for invoking $c{::}m$ with the list of arguments *args* becomes the current frame. When setting up this frame through SetFrame, memory is allocated on the stack for arguments and local variables through MakeArgLoc (see the paragraph on the abstract memory management). The macro MakeArgLoc writes the values of the incoming arguments in the addresses allocated for arguments and values of "zero" or *undef* in the addresses allocated for local variables. Note that $defVal : Map(Type, Val)$ assigns to every type its "zero", i.e. its default value. The function $zero : Map(Type \times MRef, Val \cup \{undef\})$ computes the value a local variable has upon entering the corresponding method. We use this function for both a type and a list of types.

[6]The macro DeAllocMem is defined in the paragraph on memory management.

```
$\textsc{SetFrame}(c{::}m, args) \equiv$
    $pc := 0$
    $evalStack := [\,]$
    $meth := c{::}m$
    $\textsc{MakeArgLoc}(argTypes(c{::}m), locTypes(c{::}m), args \cdot zero(locTypes(c{::}m), c{::}m))$
```

$zero(t, c{::}m) = \textbf{if } zeroInit(c{::}m) \textbf{ then } defVal(t) \textbf{ else } undef$

Remark The tail calls (in particular *Jmp* in $\mathrm{CLR}_{\mathcal{O}}$) cannot occur in `try` blocks. This is the reason there is no worry concerning the following scenario: a tail call requires first a call to a type initializer which in turn returns an exception. The exception is going to exit the method which did the tail call as if the current frame has been discarded.
The rule *InitClass* for initializing a class, sets to "zero" the static fields through InitStatFields, saves the current frame on *frameStack* and prepares the frame for invoking the type initializer `.cctor`. It does all these only if there is enough memory space to allocate for the runtime stack of the constructor.

```
$\textsc{InitStatFields}(c) \equiv \textbf{forall } f \in statFields(c) \textbf{ do}$
        $\textsc{WriteMem}(globals(f), type(f), defVal(type(f)))$
```

Unlike in JVM, the execution of any CLR type initializer does not trigger automatic execution of any initializer methods defined by its base type. *Result*(*slots*) terminates the execution of the current method and returns the result *vals* to the caller method through PopFrame.
Abstract memory management We explain here the details on how the addresses – elements of *Adr* – are allocated. We stick our formalization to the two kinds of allocations – stack and heap allocation – without considering a garbage collector. However, we provide a simple notion of stack de-allocation. We consider two dynamic functions that keep track of the addresses allocated on the stack and on the heap: $StackAdr : \mathcal{P}(Adr)$ and $HeapAdr : \mathcal{P}(Adr)$. Accordingly to their definitions, in the initial state of CLR_C, the *StackAdr* and the *HeapAdr* are $\emptyset$.

The number of addresses allocated for a value depends on the value's type. We use an external function $sizeOf : Map(Type, \mathbb{N})$ to determine the size of the block of addresses allocated for a value of a given type.

From now on, we will consider the values described by *Val* as encoded by sequences of the bytes described by *Byte*. Accordingly, the *mem* is redefined as $mem : Map(Adr, Byte \cup \{undef\})$ and the definitions of *memVal* and WriteMem are refined as follows:

$$memVal(adr, t) = [mem(adr + i) \mid i \in [0..sizeOf(t) - 1]]$$

$$\textsc{WriteMem}(adr, t, val) \equiv \textbf{forall } i \in [0..sizeOf(t) - 1] \textbf{ do } mem(adr + i) := val(i)$$

We use the partial functions *encode* and *decode* to determine the sequence of bytes associated to a value of a simple value type[7] or of a pointer type and to obtain the value associated to a sequence of bytes. For a value *val* of a type *t*, the functions *encode* and *decode* satisfy the following equations:

$length(encode(val)) = sizeOf(t)$ and $decode(t, encode(val)) = val$

We are now able to determine when there is enough space in *Adr* to be allocated for a list of values[8]. Given a finite list *sizes*, the set *SpaceFor*(*sizes*) contains the un-allocated memory blocks (if any) where can be stored values whose types have the sizes given by *sizes*. We consider the memory blocks that start at a valid address. Since, the notion of "validity" depends in general

[7]By simple value type we mean a value type which is not a value class.

[8]Due to the definition of MakeArgLoc, we have a generalized definition, i.e. for a list of values and not for a single value.

on the target architecture, we assume the external function $\mathit{validAdr} : \mathit{Map}(\mathit{Adr}, \mathit{Bool})$ decides whether an address is valid. One can find more details on this function in Section 2.6. In the definition below, n stands for $\mathit{length}(\mathit{sizes})$:

$$\begin{aligned}\mathit{SpaceFor}(\mathit{sizes}) \quad = \quad & \{(\mathit{adr}_i)_{i=0}^{n-1} \in \mathit{Adr}^n \mid \mathit{validAdr}(\mathit{adr}_i)\ \forall i = 0, n-1 \text{ and} \\ & \biguplus_{i=0}^{n-1} [\mathit{adr}_i, \mathit{adr}_i + \mathit{sizes}(i)) \subseteq \mathit{Adr} \setminus (\mathit{StackAdr} \cup \mathit{HeapAdr})\}\end{aligned}$$

If an attempt to allocate on the stack fails, a `StackOverflowException` is thrown (see CLR$_\mathcal{E}$). Similarly, if one needs to allocate on the heap, then an `OutOfMemoryException` is raised.

The following explanations are for the macro MakeArgLoc, which we used when setting up the frame for a method call. MakeArgLoc is applied to three lists: two lists of types types_1 and types_2 and a list of values *vals*. It assumes that the sum of the first two lists' lengths is equal with the length of *vals*. The macro resets first the maps *locAdr* and *argAdr* since they have to overwrite the same maps of the previous current method. Then, it allocates addresses on the stack that would be needed for arguments and local variables of types types_1 and types_2 and writes the values *vals* in these addresses. It does not assume anything concerning the order of the arguments and local variables on the stack. At the same time, the allocated addresses are pushed on the *StackAdr*. The *switch* is set to *Noswitch* only if there is sufficient place to be allocated on the stack[9]. The function sizeOf^* which is applied to the list *types* has in CLR$_\mathcal{C}$ the same definition as the function *sizeOf*[10].

```
MakeArgLoc($\mathit{types}_1, \mathit{types}_2, \mathit{vals}$) ≡
  $\mathit{locAdr} := \emptyset$
  $\mathit{argAdr} := \emptyset$
  seq
    choose $(\mathit{adr})_{i=0}^{m+n-1} \in \mathit{SpaceFor}(\mathit{sizes})$ do
      forall $i = 0, m + n - 1$ do
        if $i < m$ then $\mathit{argAdr}(i) := \mathit{adr}_i$
        else $\mathit{locAdr}(i - m) := \mathit{adr}_i$
        WriteMem($\mathit{adr}_i, \mathit{types}(i), \mathit{vals}(i)$)
      $\mathit{StackAdr} := \mathit{StackAdr} \cup \bigcup_{i=0}^{m+n-1} [\mathit{adr}_i, \mathit{adr}_i + \mathit{sizes}(i))$
  where $\mathit{types} = \mathit{types}_1 \cdot \mathit{types}_2$ and $\mathit{sizes} = \mathit{sizeOf}^*(\mathit{types})$
        and $m = \mathit{length}(\mathit{types}_1)$ and $n = \mathit{length}(\mathit{types}_2)$
```

The macro DeAllocMem has been used when exiting a method. It reclaims the memory space allocated on the stack for arguments and local variables but also the memory allocated in the local memory pool defined in Section 2.6 (see CLR$_\mathcal{P}$).

```
DeAllocMem($A$) ≡
  $\mathit{StackAdr} := A$
  forall $\mathit{adr}$ in $\mathit{StackAdr} \setminus A$ do $\mathit{mem}(\mathit{adr}) := \mathit{undef}$
```

2.4 The CLR$_\mathcal{O}$ submodule

CLR$_\mathcal{O}$ extends CLR$_\mathcal{C}$ by object-oriented features like objects creation and initialization, instance fields and methods (including instance constructors) and type casts. CLR$_\mathcal{O}$ includes also an optimization of the tail calls described in CLR$_\mathcal{C}$.

[9]To avoid a conflict in the update of *switch*, we set it in the MakeArgLoc. Note that, in case there is not sufficient space for the allocation, then the *switch* will point to the exception handling mechanism – see the refinement of MakeArgLoc in CLR$_\mathcal{E}$.

[10]The function sizeOf^* is refined in CLR$_\mathcal{O}$ while introducing object references.

Environment and State The universe *ObjType* describes object types. An object type is a reference type of a self-describing value. Some object types (e.g. abstract classes) are only a partial description of a value. The object types are the object classes (not the value classes) and the array types defined by the universe *Array*: $ObjType = ObjClass \cup Array$.

The universe *CLRStackType* is extended with the special type O corresponding to *ObjType*. The function *CLRTypeOf* maps every object type to the special type O.

$CLRStackType = \ldots \mid O$

The possible array element types are described by the universe *ArrayElemType*:

$SignedInt = \texttt{int8} \mid \texttt{int16} \mid \texttt{int32} \mid \texttt{int64}$

$UnsignedInt = \texttt{uint8} \mid \texttt{uint16} \mid \texttt{uint32} \mid \texttt{uint64}$

$FloatType = \texttt{float32} \mid \texttt{float64}$

$ArrayElemType = SignedInt \mid UnsignedInt \mid FloatType \mid \texttt{native int} \mid \texttt{ref}$

Beside static fields, in CLR$_{\mathcal{O}}$ we have also instance fields which in contrast to the static fields belong to the class objects and not to the classes.

CLR vs. JVM There is a subtle difference between CLR and JVM concerning the instance field/method references. In the case of CLR, the class specified with the field is the class from which the object on the stack inherits the field/method, i.e. the class that defines the field/method. In JVM, the class represents the class of the object on the stack.

The external function *fieldOffSet* computes for every instance field declared by a class the *field offset* within an instance of the given class. We denote by *ObjRef* the universe of object references (note that the *evalStack* does not work directly with objects but with references to objects). The *fieldAdr* assigns to every instance field of an object reference its allocated address.

$fieldOffSet : Map(Class \times FRef, \mathbb{N}) \qquad fieldAdr : Map(ObjRef \times FRef, Adr)$

For the arrays handling, we define a dynamic function *elemAdr* that records the addresses of the array elements. In the initial state, these addresses are undefined for every array reference. The function *arraySize* assigns to every array reference the length of the array object on the heap. The element type of an array reference can be computed with the derived function *elemType*.

$elemAdr : Map(ObjRef \times \mathbb{N}, Adr)$

$elemAdr(r, i) = undef,\ \forall r \in ObjRef$ and $i \in \mathbb{N}$

$arraySize : Map(ObjRef, \mathbb{N})$

$elemType : Map(ObjRef, Type)$

$elemType(r) = t$, where $actualTypeOf(r) = Array(t)$

The CLR provides support for a special kind of instance methods, namely virtual methods. They are usually used with the *CallVirt* instruction when the method to be invoked is looked up dynamically (with the function *lookup*) using the virtual method embedded in *CallVirt*. $lookup : Map(Type \times MRef, MRef)$ is defined as follows: $lookup(t, c{::}m)$ yields $d{::}m$, if $d{::}m$ is the first implementation of the method $c{::}m$ provided by a supertype of t, starting with t itself.

The objects, i.e. the instances of object classes and the arrays, are allocated on the heap. All objects on the heap are known as "boxed objects" in contrast with the value type instances introduced in CLR$_{\mathcal{VC}}$ and known as "unboxed objects". A class object is represented by its type and the addresses (and not the values as in JVM [6]) of its instance fields. The function $actualTypeOf : Map(ObjRef, Type)$ records the actual type of an object on the heap.
Due to the object references, *mem* is refined as $mem : Map(Adr, Byte \cup ObjRef \cup \{undef\})$ and the *memVal* and WriteMem are refined as follows:

```
$memVal(adr, t) = $ if $t \in ObjType$ then $mem(adr)$
                  else $[mem(adr + i) \mid i \in [0..sizeOf(t) - 1]]$

$\text{WriteMem}(adr, t, val) \equiv$ if $t \in ObjType$ then $mem(adr) := val$
                        else forall $i \in [0..sizeOf(t) - 1]$ do
                               $mem(adr + i) := val(i)$
```

Fig. 5 The execution of CLR$_{\mathcal{O}}$ instructions

```
$\mathrm{CLR}_O \equiv \mathit{execScheme}_C(\textsc{execCLR}_O, \textsc{switchCLR})$

$\textsc{execCLR}_O(\mathit{instr}) \equiv$
  $\textsc{execCLR}_C(\mathit{instr})$
  match $\mathit{instr}$
    $\mathit{NewObj}(c{::}\texttt{.ctor}) \rightarrow$
      if $c \in \mathit{ObjClass}$ then $\textsc{CreateAndInitObj}(c{::}\texttt{.ctor})$
    $\mathit{LoadField}(t, c{::}f) \rightarrow$
      if $\mathit{initialized}(c)$ then
        let $(\mathit{evalStack}', [x]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
          if $\mathit{static}(c{::}f)$ then
            $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{globals}(c{::}f), t), \mathit{CLRTypeOf}(t))]$
            $pc := pc + 1$
          elseif $x \neq \texttt{null}$ then
            $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{fieldAdr}(x, c{::}f), t), \mathit{CLRTypeOf}(t))]$
            $pc := pc + 1$
      else $\mathit{switch} := \mathit{InitClass}(c)$
    $\mathit{StoreField}(t, c{::}f) \rightarrow$ if $\mathit{initialized}(c)$ then
                                  let $(\mathit{evalStack}', [x, \mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 2)$ in
                                    if $\mathit{static}(c{::}f)$ then
                                      $\textsc{WriteMem}(\mathit{globals}(c{::}f), t, \mathit{val})$
                                      $pc := pc + 1$
                                    elseif $x \neq \texttt{null}$ then
                                      $\textsc{WriteMem}(\mathit{fieldAdr}(x, c{::}f), t, \mathit{val})$
                                      $pc := pc + 1$
                                    $\mathit{evalStack} := \mathit{evalStack}'$
                                else $\mathit{switch} := \mathit{InitClass}(c)$
    $\mathit{Call}(\mathit{tail}, \_, c{::}m) \rightarrow$
      if $\mathit{instance}(c{::}m)$ then
        if $\neg \mathit{reqinit}(c)$ then
          let $(\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{argNo}(c{::}m))$ in
            if $\mathit{vals}(0) \neq \texttt{null}$ then
              $\mathit{evalStack} := \mathit{evalStack}'$
              $\mathit{switch}     := \mathit{Invoke}(\mathit{tail}, c{::}m, \mathit{vals})$
        else $\mathit{switch} := \mathit{InitClass}(c)$
```

The function sizeOf^* which was used in the definition of MakeArgLoc in Section 2.3 is refined to take into account object references. Thus, $\mathit{sizeOf}^*(t) = \mathit{sizeOf}(\texttt{native int})$ if $t \in \mathit{ObjType}$, and $\mathit{sizeOf}^*(t) = \mathit{sizeOf}(t)$ otherwise.

Rules Fig. 5, 6 and 8 define the rules for CLR$_{\mathcal{O}}$. The rule *NewObj* is applicable only for object classes[11] and is defined through the macro CreateAndInitObj.

```
$\textsc{CreateAndInitObj}(c{::}\texttt{.ctor}) \equiv$
  if $\neg \mathit{reqinit}(c)$ then
    if $\mathit{SpaceFor}([\mathit{sizeOf}(c)]) \neq \emptyset$ then
      let $(\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{paramNo}(c{::}\texttt{.ctor}))$ in
        let $r = \mathit{new}(\mathit{ObjRef}, c)$ in
          $\mathit{evalStack} := \mathit{evalStack}' \cdot [(r, O)]$
          $\mathit{actualTypeOf}(r) := c$
          forall $f \in \mathit{instFields}(c)$ do
            $\textsc{WriteMem}(\mathit{fieldAdr}(r, f), \mathit{type}(c{::}f), \mathit{defVal}(\mathit{type}(f)))$
          $\mathit{switch} := \mathit{Invoke}(\mathit{False}, c{::}\texttt{.ctor}, [r] \cdot \mathit{vals})$
  else $\mathit{switch} := \mathit{InitClass}(c)$
```

[11] In Section 2.7 it is defined a *NewObj* rule for value classes.

Fig. 6 The execution of $\text{CLR}_{\mathcal{O}}$ instructions (*continued*)

```
$CallVirt(tail, \_, c::m) \rightarrow$
  let $(evalStack', [r] \cdot vals) = getVals(evalStack, argNo(c::m))$ in
    let $d::m = lookup(actualTypeOf(r), c::m)$ in
      if $\neg reqinit(d)$ then
        if $r \neq$ null then
          $evalStack := evalStack'$
          $switch \quad := Invoke(tail, d::m, [r] \cdot vals)$
      else $switch := InitClass(d)$
$CastClass(c) \rightarrow$ let $(r, \_) = top(evalStack)$ in
                   if $r =$ null $\vee$ $actualTypeOf(r) \preceq c$ then $pc := pc + 1$
$IsInstance(c) \rightarrow$ let $(evalStack', [r]) = getVals(evalStack, 1)$ in
                   $pc := pc + 1$
                   if $r =$ null $\vee$ $actualTypeOf(r) \not\preceq c$ then
                     $evalStack := evalStack' \cdot [($null$, O)]$
$\text{ArrayInstr}_{\mathcal{O}}$
$\text{NonVerifiable}_{\mathcal{O}}$
```

An instance of the given object class is allocated provided that the allocation does not require the class initialization, otherwise it proceeds with the initialization. The allocation succeeds if there is sufficient memory to be allocated (on the heap). The object reference and the object on the heap are created through the following macro:

```
let $r = new(ObjRef, t)$ in $P \equiv$
  import $r$ do
    $ObjRef(r) := True$
    choose $(adr) \in SpaceFor([n])$ do
      $HeapAdr := HeapAdr \cup [adr, adr + n)$
      forall $f \in instFields(t)$ do
        let $a = adr + fieldOffSet(t, f)$ in
          $fieldAdr(r, f) := a$
          $\text{AllocFields}(a, type(f))$
  seq $P$
  where $n = sizeOf(t)$
```

```
$\text{AllocFields}(adr, t) \equiv$
  skip
```

The object is allocated in the *HeapAdr*. The addresses of the instance fields are computed using the object starting address and the field offsets. In $\text{CLR}_{\mathcal{VC}}$, the fields might be of a value class type. In that case, one has to compute also the addresses of the instance fields of the corresponding value class instance. At this point, the macro AllocFields does nothing but it will be refined in Section 2.7.
Beside the allocation, CreateAndInitObj initializes all the instance fields of the newly created reference and invokes ("non-tail") the instance constructor embedded in *NewObj* with the necessary number of values present on the *evalStack*.
CLR vs. JVM The JVM instruction `new` is used to create and load on the stack a class instance. The `newobj` instruction from CIL does the same with a single difference: it calls then an instance constructor (specified in the CIL instruction) of the class to set the instance. In JVM, this has to be done explicitly through an `invokespecial` instruction.

The field values can be read with *LoadField* and be written with *StoreField*. *LoadField* takes the value of the topmost *evalStack* slot which is an object reference or a pointer to a value type instance (see $\text{CLR}_{\mathcal{VC}}$). The loaded value is the value stored at the field address which is given by the *globals* if the field is static or by the *fieldAdr* otherwise. *StoreField* takes from the *evalStack* the values of the two topmost slots: the first is an object reference or a pointer and the second is the value to be stored at the field address. For both *LoadField* and *StoreField*, if the field is declared by an uninitialized class, the class is first initialized.

Fig. 7 The execution of CLR$_{\mathcal{O}}$ instructions (*continued*)

```
ArrayInstr$_{\mathcal{O}}$ ≡
  $NewArray(t) \rightarrow$ let $(evalStack', [n]) = getVals(evalStack, 1)$ in
      if $n \geq 0$ then
        if $SpaceFor([n * sizeOf(t)]) \neq \emptyset$ then
          let $r = new(ObjRef, t, n)$ in
            $actualTypeOf(r) := Array(t)$
            $arraySize(r) := n$
            $evalStack := evalStack' \cdot [(r, O)]$
            $pc := pc + 1$
  $LoadLength \rightarrow$ let $(evalStack', [r]) = getVals(evalStack, 1)$ in
      if $r \neq$ null then
        $evalStack := evalStack' \cdot [(arraySize(r),$ native int$)]$
        $pc := pc + 1$
  $LoadElem(t) \rightarrow$ let $(evalStack', [r, i]) = getVals(evalStack, 2)$ in
      if $r \neq$ null $\wedge (i \geq 0 \wedge i < arraySize(r))$ then
        $evalStack := evalStack' \cdot [(memVal(elemAdr(r, i), elemType(r)), CLRTypeOf(t))]$
        $pc := pc + 1$
  $StoreElem(t) \rightarrow$ let $(evalStack', [r, i, val]) = getVals(evalStack, 3)$ in
      if $r \neq$ null $\wedge (i \geq 0 \wedge n < arraySize(r))$
      $\wedge(t \neq$ ref $\vee\ val =$ null $\vee\ actualTypeOf(val) \preceq elemType(r))$ then
        WriteMem$(elemAdr(r, i), elemType(r), val)$
        $evalStack := evalStack'$
        $pc := pc + 1$
```

Remark *LoadField*, *StoreField* and *CallVirt* (but also *LoadFieldA* in CLR$_{\mathcal{P}}$) can be applied also to static members but they require anyway on *evalStack* also an object reference or a pointer to a value type instance (see CLR$_{\mathcal{VC}}$). This is another difference wrt JVM [6].

The *Call* rule that we define in CLR$_{\mathcal{O}}$ can be fired only for instance methods (for statics the *Call* rule from CLR$_{\mathcal{C}}$ can be fired). If there is no need to initialize the class declaring the called method, the method is invoked through switchCLR which considers also information about a possible "tail call". *Call* pops from the *evalStack* the target reference (assumed non-null) representing the instance whose method is invoked and the arguments. The number and types of the arguments are given by the derived function *argTypes* whose definition is refined in CLR$_{\mathcal{O}}$ as follows: for every instance method reference $c::m$ where c is an object class:

$argTypes(c::m) = [c] \cdot paramTypes(c::m)$

In case of *CallVirt*, the only difference wrt *Call* is that the method is late bound. The method to be invoked is looked up dynamically by means of *lookup*. *CastClass* checks whether the reference on top of the *evalStack* is of the required class. If the attempted cast does not succeed, *CastClass* throws an exception (see CLR$_{\mathcal{E}}$). *IsInstance* pops from *evalStack* a reference to a boxed object. If the reference is null or the actual type of the reference is not compatible with the given class, then a null reference is pushed on *evalStack*. Otherwise, it lets the *evalStack* unchanged.

CLR vs. JVM The CLR instruction *IsInstance* is different than the JVM *InstanceOf*. Depending on whether the cast succeeds or not, the former loads a null on the stack or lets the stack unchanged while the later loads on the stack either 1 or 0.

Fig. 7 contains the instructions for the array handling. One can create an array object with the *NewArray* instruction. The number of array elements is popped from the stack – the array creation succeeds only if the given number is non-negative and there is sufficient memory to allocate the array on the heap. The array is allocated through the following macro:

Fig. 8 The execution of non-verifiable $\mathrm{CLR}_{\mathcal{O}}$ instructions

```
$\textsc{NonVerifiable}_{\mathcal{O}} \equiv$
  $Jmp(c{::}m) \rightarrow$
    if $\neg reqinit(c)$ then
      let $args = [memVal(argAdr(i), argTypes(i)) \mid i = 0, argNo(meth) - 1]$ in
        $switch := Invoke(True, c{::}m, args)$
    else $switch := InitClass(c)$
```

```
let $r = new(ObjRef, t, n)$ in $P \equiv$
  import $r$ do
    $ObjRef(r) := True$
    choose $(adr) \in SpaceFor([n * sizeOf(t)])$ do
      $HeapAdr := HeapAdr \cup [adr, adr + n * m)$
      forall $i = 0, n - 1$ do $elemAdr(r, i) := adr + (i - 1) * m$
  seq $P$
  where $m = sizeOf(t)$
```

The elements addresses are recorded by *elemAdr*. The memory space occupied by the array is simply a contiguous block of bytes allocated from the arbitrarily chosen address. The size of the block depends on the element type and length of the array.

CLR vs. JVM In comparison with JVM, the CLR provides less instructions that deal with arrays. For example, JVM has a specialized instruction `anewarray` that constructs only arrays of object references. Additionally, JVM has an instruction `newarray` that creates arrays of primitive type values. In JVM, there is also the instruction `multianewarray` which creates a multidimensional array. In CLR, the multidimensional arrays are created using `newobj` and therefore they are treated in our abstraction as the classes objects.

The length of an array can be obtained using the *LoadLength* instruction. The *LoadElem* loads on the stack an array element given by the index and array reference on the *evalStack*. If one needs to write the value of an array element, then we use the *StoreElem* instruction. The store operation succeeds if the array reference is not `null`, the given index is in the array range and, in case of reference types, the type of the value to be stored is compatible with the array element type.

Although *Jmp* is not verifiable, we still model its semantics (see Fig. 8). If the method is declared by a class which requires initialization, *Jmp* initializes the class first. *Jmp* is an optimization of a tail call. Consequently, the current frame is discarded while invoking the given method. Note that, the given method is also the method to be invoked - there is no lookup. The invocation arguments are exactly the arguments of the current frame *at the time* when *Jmp* is fired. The memory allocated on the stack for the current frame is reclaimed through DeAllocMem.

2.5 The $\mathrm{CLR}_{\mathcal{E}}$ submachine

$\mathrm{CLR}_{\mathcal{E}}$ extends $\mathrm{CLR}_{\mathcal{O}}$ by instructions dealing with exception handling. $\mathrm{CLR}_{\mathcal{E}}$ also extends in the expected way the rules defined so far in the submachines where run-time exceptions might occur.

This section is organized as follows. Section 2.5.1 gives an overview of the CLR exception handling mechanism. The elements of the formalization are introduced in Section 2.5.2. Section 2.5.3 defines the so-called *StackWalk* pass of the exception mechanism. The other two passes, *Unwind* and *Leave* are defined in Section 2.5.4 and Section 2.5.5, respectively. The execution rules of CLR_{E} are introduced in Section 2.5.6.

2.5.1 The Overall Picture

Every time an exception occurs, the control is transfered from "normal" execution to a so-called "exception handling mechanism" which we model as a submachine EXCCLR. In order to support the EXCCLR, the universe *Switch* is extended with a new value which indicates the handling of an exception: when *switch* gets *ExcMech*, the control is passed to the exception handling mechanism.

$Switch = \ldots \mid ExcMech$

The exception handling mechanism proceeds in two passes. In the first pass, the run-time system runs a "stack walk" searching, in the exception handling array associated by $excHA : Map(MRef, List(Exc))$ to current method for the first handler that might want to handle the exception:

- a `catch` handler whose *type* is a supertype of the type of the exception, or
- a `filter` handler – to see whether a `filter` wants to handle an exception, one has first to execute (in the first pass) the code in the filter region: if it returns 1, then it is chosen to handle the exception; if it returns 0, this handler is not good to handle the exception.

Visual Basic and Managed C++ have special `catch` blocks which can "filter" the exceptions based on the exception type and / or any conditional expression. These are compiled into `filter` handlers in the Common Intermediate Language (CIL) bytecode. As we will see, the `filter` handlers bring a lot of complexity to the exceptions mechanism.

The ECMA standard does not clarify what happens if the execution of the `filter` or of a method called by it throws an exception. The currently handled exception is known as an *outer exception* while the newly occured exception is called an *inner exception*. As we will see below, the outer exception is not discarded but its context is saved by EXCCLR while the inner exception becomes the outer exception.

If a match is not found in the *faulting frame*, i.e. the frame where the exception has been raised, the calling method is searched, and so on. This search eventually terminates since the *excHA* of the `entrypoint` method has as last entry a so-called *backstop entry* placed there by the operating system. When a match is found, the first pass terminates and in the second pass, called "unwinding of the stack", CLR walks once more through the stack of call frames to the handler determined in the first pass, but this time executing the `finally` and `fault`[12] handlers and popping their frames. It then starts the corresponding exception handler.

2.5.2 The Global View of EXCCLR

In this section, we provide some detail on the elements, functions and predicates needed to turn the overall picture into a rigorous model.

The elements of an exception handling array $excHA : Map(MRef, List(Exc))$ are known as *handlers* and can be of four kinds. They are elements of a set *Exc*:

[12]Currently, no language (other than CIL) exposes `fault` handlers directly. A `fault` handler is simply a `finally` handler that only executes in the exceptional case. It is never executed if the associated `try` block terminates normally.

Fig. 9 The predicates *isInTry*, *isInHandler* and *isInFilter*

$$\begin{array}{lcl}
\mathit{isInTry}(\mathit{pos}, h) & \Leftrightarrow & \mathit{tryStart}(h) \leq \mathit{pos} < \mathit{tryStart}(h) + \mathit{tryLength}(h) \\
\mathit{isInHandler}(\mathit{pos}, h) & \Leftrightarrow & \mathit{handlerStart}(h) \leq \mathit{pos} < \mathit{handlerStart}(h) + \mathit{handlerLength}(h) \\
\mathit{isInFilter}(\mathit{pos}, h) & \Leftrightarrow & \mathit{filterStart}(h) \leq \mathit{pos} < \mathit{handlerStart}(h)
\end{array}$$

$$\begin{array}{lcl}
\mathit{ClauseKind} & = & \texttt{catch} \mid \texttt{filter} \\
 & \mid & \texttt{finally} \mid \texttt{fault}
\end{array}$$

$$\begin{array}{lllll}
\mathit{Exc} = \mathit{Exc}\ (& \mathit{clauseKind} & : & \mathit{ClauseKind} & \\
 & \mathit{tryStart} & : & \mathit{Pc} & \\
 & \mathit{tryLength} & : & \mathbb{N} & \\
 & \mathit{handlerStart} & : & \mathit{Pc} & \\
 & \mathit{handlerLength} & : & \mathbb{N} & \\
 & \mathit{type} & : & \mathit{ObjClass} & \\
 & \mathit{filterStart} & : & \mathit{Pc} &)
\end{array}$$

Any 7-tuple of the above form describes a handler of kind *clauseKind* which "protects" the region[13] that starts at *tryStart* and has the length *tryLength*, handles the exception in an area of instructions that starts at *handlerStart* and has the length *handlerLength* – we refer to this area as the *handler region*; if the handler is of kind `catch`, then the *type* of exceptions it handles is provided while if the handler is of kind `filter`, then the first instruction of the `filter` *region* is at *filterStart*. In case of a `filter` handler, the handler region starting at *handlerStart* immediately follows the `filter` region – consequently we have $\mathit{filterStart} < \mathit{handlerStart}$. We often refer to the sequence of instructions between *filterStart* and $\mathit{handlerStart} - 1$ as the `filter` *region*. We assume that a *filterStart* is defined for a handler if and only if the handler is of kind `filter`, otherwise *filterStart* is undefined.
To simplify the further presentation, we define the predicates in Fig. 9 for an instruction located at program counter position $\mathit{pos} \in \mathit{Pc}$ and a handler $h \in \mathit{Exc}$. Note that if the predicate *isInFilter* is true, then *filterStart* is defined and therefore h is of kind `filter`. Based on the lexical nesting constraints of protected blocks specified in [1, Partition I,§12.4.2.7], one can prove the following property:

Disjointness 1 *The predicates isInTry, isInHandler and isInFilter are pairwise disjoint.*

We assume all the constraints concerning the lexical nesting of handlers specified in the standard [1, Partition I,§12.4.2.7]. The ECMA standard [1, Partition I,§12.4.2.5] ordering assumption on handlers is:

Ordering assumption *If handlers are nested, the most deeply nested try blocks shall come in the exception handling array before the try blocks that enclose them.*

Only one handler region per `try` block? The ECMA standard specifies in [1, Partition I,§12.4.2] that a single `try` block shall have exactly one handler region associated with it. But the IL assembler `ilasm` does accept also `try` blocks with more than one `catch` handler block. This discrepancy is solved if we assume that every `try` block with more than one `catch` block which is accepted by the `ilasm` is translated in a semantics preserving way as follows:

[13]We will refer to this region as *protected region* or `try` block.

```
.try {                          .try {
  block                           .try {
} catch block_1      ⟹              block
  catch block_2                   } catch block_1
                                } catch block_2
```

To handle an exception, the EXCCLR needs to record:

- the exception reference *exc*,
- the handling *pass*,
- a *stackCursor* – i.e. the position currently reached in the stack of call frames (a frame) and in the exception handling array (an index in *excHA*),
- the suitable *handler* determined at the end of the *StackWalk* pass (if any) is the handler that is going to handle the exception in the pass *Unwind* – until the end of the *StackWalk* pass, *handler* is undefined;

According to the ECMA standard, every normal execution of a `try` block or a `catch`/`filter` handler region must end with a *Leave*(*pos*) instruction. When doing this, EXCCLR has to record the current *pass* and *stackCursor* together with the *target* up to which every included `finally` code has to be executed.

$$
\begin{array}{l}
\mathit{ExcRec} = \\
\quad \mathit{ExcRec}\ (\ \begin{array}{lcl} \mathit{exc} & : & \mathit{ObjRef} \\ \mathit{pass} & : & \{\mathit{StackWalk}, \mathit{Unwind}\} \\ \mathit{stackCursor} & : & \mathit{Frame} \times \mathbb{N} \\ \mathit{handler} & : & \mathit{Frame} \times \mathbb{N}\ ) \end{array} \\
\mathit{LeaveRec} = \\
\quad \mathit{LeaveRec}\ (\ \begin{array}{lcl} \mathit{pass} & : & \{\mathit{Leave}\} \\ \mathit{stackCursor} & : & \mathit{Frame} \times \mathbb{N} \\ \mathit{target} & : & \mathit{Pc}\ ) \end{array}
\end{array}
$$

We list some constraints which will be needed below to understand the treatment of these *Leave* instructions.

Syntactic constraints:

1. It is not legal to exit with a *Leave* instruction a `filter` region, a `finally`/`fault` handler region.

2. It is not legal to branch with a *Leave* instruction into a handler region from outside the region.

3. It is legal to exit with a *Leave* a `catch` handler region and branch to any instruction within the associated `try` block, so long as that branch target is not protected by yet another `try` block.

The nesting of passes determines EXCCLR to maintain an initially empty stack of exception or leave records for the passes that are still to be performed.

$$excRecStack : List(ExcRec \cup LeaveRec)$$
$$excRecStack = [\,]$$

In the initial state of EXCCLR, there is no pass to be executed, i.e. $pass = undef$.

We can now summarize the overall behavior of EXCCLR, which is defined in Fig. 10 and analyzed in detail in the following sections, by saying that if there is a handler in the frame defined by *stackCursor*, then EXCCLR will try to find (when *StackWalk*ing) or to execute (when *Unwind*ing) or to leave (when *Leave*ing) the corresponding handler; otherwise it will continue its work in the invoker frame or end its *Leave* pass at the *target*.

2.5.3 *StackWalk* pass

During a *StackWalk* pass, EXCCLR starts in the current *frame* to search for a suitable handler of the current exception in this frame. Such a handler exists if the search position n in the current frame has not yet reached the last element of the handlers array *excHA* of the corresponding method m.

$$existsHanWithinFrame((\_,\_,\_,\_,\_,m),n) \Leftrightarrow n < length(excHA(m))$$

If there are no (more) handlers in the frame pointed to by *stackCursor*, then the search has to be continued at the invoker frame. This means to reset the *stackCursor* to point to the invoker frame.

```
SEARCHINVFRAME($f$) ≡
  let $\_ \cdot [f', f] \cdot \_ = frameStack \cdot [frame]$ in
    RESET($stackCursor, f'$)
```

There are three groups of possible handlers h EXCCLR is looking for in a given frame during its *StackWalk*:

- a `catch` handler whose `try` block protects the program counter *pc* of the frame pointed at by *stackCursor* and whose *type* is a supertype of the exception type;

$$matchCatch(pos,t,h) \Leftrightarrow isInTry(pos,h) \wedge clauseKind(h) = \texttt{catch} \wedge t \preceq type(h)$$

- a `filter` handler whose `try` block protects the *pc* of the frame pointed at by *stackCursor*;

$$matchFilter(pos,h) \Leftrightarrow isInTry(pos,h) \wedge clauseKind(h) = \texttt{filter}$$

- a `filter` handler whose `filter` region contains *pc* of the frame pointed at by *stackCursor*. This corresponds to an outer exception and will be described in more detail below.

The order of the **if** clauses in the **let** statement from the rule *StackWalk* is not important. This is justified by the following property:

Disjointness 2 *For every type t, the predicates* $matchCatch^t$, *matchFilter and isInFilter are pairwise disjoint*[14].

[14]By $matchCatch^t$ we understand the predicate defined by the set $\{(pos,h) \mid matchCatch(pos,t,h)\}$.

The above property can be easily proved using the definitions of the three predicates and the property *Disjointness* 1.

If the handler pointed to by the *stackCursor*, namely
$hanWithinFrame((\_, \_, \_, \_, \_, m), n) = excHA(m)(n)$,
is not of any of the above types, the *stackCursor* is incremented to point to the next handler in the *excHA*:

```
GoToNxtHan ≡ $stackCursor := (f, n + 1)$
where $stackCursor = (f, n)$
```

The *Ordering assumption* stated in Section 2.5.2 and the lexical nesting constraints stated in [1, Partition I,§12.4.2.7] ensure that, if the *stackCursor* points to a handler of one of the above types, then this handler is the first handler in the exception handling array (starting at the position indicated in the *stackCursor*) of any of the above types.

If the handler pointed to by the *stackCursor* is a matching `catch`, then this handler becomes the *handler* to handle the exception in the pass *Unwind*. The *stackCursor* is reset to be reused for the *Unwind* pass: it shall point to the faulting frame, i.e. the current *frame*. Note that during *StackWalk*, *frame* always points to the faulting frame except in case a `filter` region is executed. However, the frame built to execute a `filter` is never searched for a handler corresponding to the current exception.

```
FoundHandler ≡
  $pass := Unwind$
  $handler := stackCursor$

Reset$(s, f) ≡ s := (f, 0)$
```

If the handler is a `filter`, then by means of ExecFilter its `filter` region is executed. The execution is performed in a separate frame constructed especially for this purpose. However this important detail is omitted by the ECMA standard [1]. The currently to be executed frame becomes the frame for executing the `filter` region. The faulting exception frame is pushed on the *frameStack*. The current frame points now to the method, local variables and arguments of the frame in which *stackCursor* is, it has the exception reference on the evaluation stack *evalStack* and the program counter *pc* set to the beginning *filterStart* of the `filter` region. The *switch* is set to *Noswitch* in order to pass the control to the normal machine execCLR$_E$.

```
ExecFilter$(h)$ ≡
  $pc := filterStart(h)$
  $evalStack := [exc]$
  $locAdr := locAdr'$
  $argAdr := argAdr'$
  $meth := meth'$
  $push(frameStack, frame)$
  $switch := Noswitch$
  where $stackCursor = ((\_, locAdr', argAdr', \_, \_, meth'), \_)$
```

Exceptions in `filter` region? It is not documented in the ECMA standard what happens if an (inner) exception is thrown while executing the `filter` region during the *StackWalk* pass of an outer exception. The following cases are to be considered:

Fig. 10 The exception handling machine EXCCLR

```
EXCCLR ≡
  match $pass$
    $StackWalk \rightarrow$ if $existsHanWithinFrame(stackCursor)$ then
                         let $h = hanWithinFrame(stackCursor)$ in
                           if $matchCatch(pos, actualTypeOf(exc), h)$ then
                             FOUNDHANDLER
                             RESET$(stackCursor, frame)$
                           elseif $matchFilter(pos, h)$ then EXECFILTER$(h)$
                           elseif $isInFilter(pos, h)$ then EXITINNEREXC
                           else GOTONXTHAN
                       else  SEARCHINVFRAME$(f)$
                       where $stackCursor = (f, \_)$ and $f = (pos, \_, \_, \_, \_, \_)$

    $Unwind \rightarrow$ if $existsHanWithinFrame(stackCursor)$ then
                      let $h = hanWithinFrame(stackCursor)$ in
                        if $matchTargetHan(handler, stackCursor)$ then
                          EXECHAN$(h)$
                        elseif $matchFinFault(pc, h)$ then
                          EXECHAN$(h)$
                          GOTONXTHAN
                        elseif $isInHandler(pc, h)$ then
                          ABORTPREVPASSREC
                          GOTONXTHAN
                        elseif $isInFilter(pc, h)$ then
                          CONTINUEOUTEREXC
                        else GOTONXTHAN
                    else
                      if $meth = c$::.cctor then $classState(c) := Exc(exc)$
                      POPFRAME$(0, [\,])$
                      SEARCHINVFRAME$(frame)$

    $Leave \rightarrow$ if $existsHanWithinFrame(stackCursor)$ then
                     let $h = hanWithinFrame(stackCursor)$ in
                       if $isFinFromTo(h, pc, target)$ then EXECHAN$(h)$
                       if $isRealHanFromTo(h, pc, target)$ then ABORTPREVPASSREC
                       GOTONXTHAN
                   else
                     $pc := target$
                     POPREC
                     $switch := Noswitch$
```

- if the exception is taken care of in the `filter` region, i.e. it is successfully handled by a `catch`/`filter` handler or it is aborted because it occured in yet another `filter` region of a nested handler (see the *isInFilter* clause), then the given `filter` region continues executing normally (after the exception has been taken care of);

- if the exception is not taken care of in the `filter` region, then the exception is not propagated further, but its *StackWalk* is exited (see Fig. 10). The exception will be discarded but only after the EXCCLR runs its *Unwind* pass to execute all the `finally` and `fault` handlers (see `Tests` 6, 8 and 9 in [3]).

```
EXITINNEREXC ≡
  $pass := Unwind$
  RESET($stackCursor, frame$)
```

2.5.4 *Unwind* pass

As soon as the pass *StackWalk* terminates, the EXCCLR starts the *Unwind* pass with the *stackCursor* pointing to the faulting exception frame. Starting there one has to walk down to the *handler* determined in the *StackWalk*, executing on the way every `finally`/`fault` handler region. This happens also in case *handler* is *undef*. When *Unwind*ing, the EXCCLR searches for

- the matching target handler, i.e. the *handler* determined at the end of the *StackWalk* pass (if any) – *handler* can be *undef* if the search in the *StackWalk* has been exited because the exception was thrown in a `filter` region. Also the two *handler* and *stackCursor* frames in question have to coincide. We say that two frames are the same if the address arrays of their local variables and arguments as well their method names coincide.

$$matchTargetHan((f1, n1), (f2, n2)) \Leftrightarrow sameFrame(f1, f2) \wedge n1 = n2$$

$$sameFrame(f1, f2) \Leftrightarrow pr_i(f1) = pr_i(f2), \forall i \in \{2, 3, 6\}$$

- a matching `finally`/`fault` handler whose associated `try` block protects the *pc*;

$$matchFinFault(pos, h) \Leftrightarrow isInTry(pos, h) \wedge clauseKind(h) \in \{\texttt{finally}, \texttt{fault}\}$$

- a handler whose handler region contains *pc*;
- a `filter` handler whose `filter` region contains *pc*;

The order of the last three **if** clauses in the **let** statement from the rule *Unwind* is not important. It matters only that the first clause is guarded by *matchTargetHan*.

Disjointness 3 *The following predicates are pairwise disjoint: matchFinFault, isInHandler and isInFilter.*

The property can be proved using the definitions of the predicates and the property *Disjointness* 1.

The *Ordering assumption* in Section 2.5.2 the lexical nesting constraints given in [1, Partition I,§12.4.2.7] ensure that, if the *stackCursor* points to a handler of one of the above types, then this handler is the first handler in the exception handling array (starting at the position indicated in the *stackCursor*) of any of the above types.

If the handler pointed to by the *stackCursor* is the *handler* found in the *StackWalk*, its handler region is executed through EXECHAN: the *pc* is set to the beginning of the handler region, the exception reference is loaded on the evaluation stack (when EXECHAN is applied for executing `finally`/`fault` handler regions the current exception is not pushed on *evalStack*) and the control switches to EXECCLR$_E$.

```
EXECHAN($h$) $\equiv$
  $pc := handlerStart(h)$
  $evalStack :=$
    if $clauseKind(h) \in \{$catch, filter$\}$ then
      $[exc]$
    else
      $[\,]$
  $switch := Noswitch$
```

If the handler pointed to by the *stackCursor* is a matching `finally`/`fault` handler, its handler region is executed with initially empty evaluation stack. At the same time, the *stackCursor* is incremented through GoToNxtHan.

Let us assume that the handler pointed to by *stackCursor* is an arbitrary handler whose handler region contains *pc*.

Exceptions in handler region? The ECMA standard does not specify what should happen if an exception is raised in a handler region. The experimentation in [3] can be resumed by the following rules of thumb for exceptions thrown in a handler region similarly to the case of nested exceptions in `filter` code:

- if the exception is taken care of in the handler region, i.e. it is successfully handled by a `catch`/`filter` handler or it is discarded (because it occured in a `filter` region of a nested handler), then the handler region continues executing normally (after the exception is taken care of);

- if the exception is not taken care of in the handler region, i.e, escapes the handler region, then

  – the previous pass of excCLR is aborted through AbortPrevPassRec;

```
AbortPrevPassRec $\equiv pop(excRecStack)$
```

  – the exception is propagated further, i.e. the *Unwind* pass continues via GoToNxtHan (see Fig. 10) which sets the *stackCursor* to the next handler in *excHA*.

The execution of a handler region can occur only when excCLR runs in the *Unwind* and *Leave* passes: in *Unwind* handler regions of any kind are executed while in *Leave* only `finally` handler regions are executed. If the raised exception occured while excCLR runs an *Unwind* pass for handling an outer exception, the *Unwind* pass of the outer exception is stopped and the corresponding pass record is popped from *excRecStack* (see `Tests` 1, 3 and 4 in [3]). If the exception has been thrown while excCLR runs a *Leave* pass for executing `finally` handlers on the way from a *Leave* instruction to its target, then this pass is stopped and its associated pass record is popped off *excRecStack* (see `Test` 2 in [3]).
In this way an exception can go "unhandled" without taking down the process, namely if an outer exception goes unhandled, but an inner exception is successfully handled (see the second case of the preceding case distinction).

If the handler pointed to by the *stackCursor* is a `filter` handler whose `filter` region contains *pc*, then the current (inner) exception is aborted and the `filter` considered as not providing a handler for the outer exception. So there is no way to exit a `filter` region with an exception. This ensures that the frame built by ExecFilter for executing a `filter` region is used only for this purpose. The handling of the outer exception is continued through ContinueOuterExc (see Fig. 10) which pops the frame built for executing the `filter` region, pops from the *excRecStack* the pass record corresponding to the inner exception and reestablishes the pass context of the outer exception, but with the *stackCursor* pointing to the handler following the just inspected `filter` handler. The updates of the *stackCursor* in PopRec and GotoNxtHan are done **seq**uentially such that the update in GotoNxtHan overwrites the update in PopRec.

```
ContinueOuterExc ≡
   PopFrame$(0, [\,])$
   PopRec seq GoToNxtHan
```

```
PopRec ≡
  if $excRecStack = [\,]$ then
    SetRecUndef
    $switch := Noswitch$
  else let $(excRecStack', [r]) =$
             $split(excRecStack, 1)$ in
   if $r \in ExcRec$ then
     let $(exc', pass', stackCursor', handler') = r$ in
       $exc$         $:= exc'$
       $pass$        $:= pass'$
       $stackCursor$ $:= stackCursor'$
       $handler$     $:= handler'$
   if $r \in LeaveRec$ then
     let $(pass', stackCursor', target') = r$ in
       $pass$        $:= pass'$
       $stackCursor$ $:= stackCursor'$
       $target$      $:= target'$
   $excRecStack := excRecStack'$

 SetRecUndef ≡
   $exc$         $:= undef$
   $pass$        $:= undef$
   $stackCursor$ $:= undef$
   $target$      $:= undef$
   $handler$     $:= undef$
```

If the handler pointed to by the *stackCursor* is not of any of the above types, the *stackCursor* is incremented to point to the next handler in the *excHA*.

If the *Unwind* pass exhausted all the handlers in the frame indicated in *stackCursor*, then the current frame is popped from *frameStack* and the *Unwind* pass continues in the invoker frame of the current frame.

Exceptions in class initializers? If an exception occurs in a type initializer `.cctor`, then the type shall be marked as being in an erroneous state and a `TypeInitializationException` is thrown. This means that an exception can and will escape the body of an initializer only by the specific exception `TypeInitializationException`. This is not specified by the ECMA

Fig. 11 The switchCLR machine (*continued*)

```
switchCLR ≡ match $switch$
  ...
  $InitClass(c) \rightarrow$ if $classState(c) = Exc(r)$ then
                        $\text{LoadRec}((r, StackWalk, (frame, 0), undef))$
```

standard but it seems to correspond to the actual CLR implementation and it complies with the related specification for C♯ in the ECMA standard (see `Test` 7 in [3]). Therefore we assume that the code sequence of every `.cctor` is embedded into a `catch` handler. This `catch` handler catches exceptions of type `Object`, i.e. any exception, occured in `.cctor`, discards it, creates an object of type `TypeInitializationException`[15] and throws the new exception.

To support the types in an erroneous state, a new value is injected into the universe of class states:

$ClassState = \ldots \mid Exc(ObjRef)$

The meaning is that, if a class is in a state $Exc(r)$, then the class initializer has been exited with a `TypeInitializationException` reference r.

The *InitClass* rule of the switchCLR submachine is refined in Fig. 11 to take into account attempts to initialize a class that is in an erroneous state. The macro LoadRec is defined in Section 2.5.6 and basically loads into the exception mechanism the same exception reference that is embedded in the class state value.

Since every attempt to use a class in an $Exc(\_)$ state shall throw the same initially thrown exception reference, we need to refine the predicate *reqinit* as follows:

$$reqinit(c) \Leftrightarrow (\neg initialized(c) \vee \neg beforefieldinit(c)) \vee classState(c) = Exc(\_)$$

When a field is accessed, the predicate *initialized* is checked. If the corresponding class is in an $Exc(\_)$ state, then the predicate *initialized* is *False*. Consequently, the *switch* is set to *InitClass* which will throw the same initially thrown exception reference.

2.5.5 *Leave* pass

The excCLR machine gets into the *Leave* pass when execCLR$_E$ executes a *Leave* instruction upon the normal termination of a `try` block or of a `catch`/`filter` handler region. One has to execute the handler regions of all `finally` handlers on the way from the *Leave* instruction to the instruction whose program counter is given by the *Leave target* parameter. The *stackCursor* used in the *Leave* pass is initialized by the *Leave* instruction. In the *Leave* pass, the excCLR machine searches for

- `finally` handlers that are "on the way" from the *pc* to the *target*,
- real handlers, i.e. `catch`/`filter` handlers that are "on the way" from the *pc* to the *target* – more details are given below.

If the handler pointed to by *stackCursor* is a `finally` handler on the way from *pc* to the *target* position of the current *Leave* pass record, then the handler region of this handler is executed (see Fig. 10). If the *stackCursor* points to a `catch`/`filter` handler on the way from *pc* to *target*, then the previous pass record on *excRecStack* is discarded (see Fig. 10).

[15] In the real CLR implementation, the exception thrown in `.cctor` is embedded as an inner exception in the `TypeInitializationException`. We do not model this aspect here.

The discarded record can only be referring to an *Unwind* pass for handling an exception. By discarding this record, the mechanism terminates the handling of the corresponding exception.

$$\begin{array}{l} isFinFromTo(h, pos_1, pos_2) \Leftrightarrow \\ isInTry(pos_1, h) \wedge clauseKind(h) = \texttt{finally} \wedge \neg isInTry(pos_2, h) \wedge \neg isInHandler(pos_2, h) \\ isRealHanFromTo(h, pos_1, pos_2) \Leftrightarrow \\ clauseKind(h) \in \{\texttt{catch}, \texttt{filter}\} \wedge isInHandler(pos_1, h) \wedge \neg isInHandler(pos_2, h) \end{array}$$

For each handler EXCCLR inspects also the next handler in *excHA*. When the handlers in the current method are exhausted, *pc* is set to *target*, the context of the previous pass record on *excRecStack* is reestablished and the control is passed to normal EXECCLR$_E$ execution (see Fig. 10).

2.5.6 The rules of CLR$_\mathcal{E}$

Fig. 12 defines the rules of CLR$_\mathcal{E}$. This submachine is defined through the *execScheme*$_E$ which extends the previous scheme *execScheme*$_C$. The new scheme takes into account also the exception handling mechanism EXCCLR.

The rules of EXECCLR$_E$ in Fig. 12 specify the effect of the CIL instructions related to exceptions. Each of these rules transfers the control to EXCCLR. The *Throw* instruction is defined through the macro THROW:

```
THROW($r$) $\equiv$
  if $r \neq \texttt{null}$ then
    LOADREC(($r$, $StackWalk$, ($frame$, 0), $undef$))
  else RAISE(NullReferenceException)
```

The topmost element of the *evalStack*, which is supposed to be an exception reference, is popped off (see Remark below). The pass record associated to the given exception is loaded on the EXCCLR: the *stackCursor* is initialized by the current *frame* and 0. If the exception mechanism is already working in a pass, i.e. $pass \neq undef$, then the current pass record is pushed on the *excRecStack*.

```
LOADREC($r$) $\equiv$
  if $r \in ExcPass$ then
    let $(exc', pass', stackCursor', handler') = r$ in
      $exc$         $:= exc'$
      $pass$        $:= pass'$
      $stackCursor$ $:= stackCursor'$
      $handler$     $:= handler'$
  else let $(pass', stackCursor', target') = r$ in
    $pass$        $:= pass'$
    $stackCursor$ $:= stackCursor'$
    $target$      $:= target'$
  if $pass \neq undef$ then PUSHREC
  $switch := ExcMech$
```

```
PUSHREC ≡
  if $pass = Leave$ then
    $push(excRecStack, (pass, stackCursor, target))$
  else $push(excRecStack,$
              $(exc, pass, stackCursor, handler))$
```

If the exception reference popped off the *evalStack* is `null`, a `NullReferenceException` is thrown. For a given class c, the macro RAISE(c) is defined as follows.

```
RAISE($c$) ≡
  CREATEANDINITOBJ($c$::.ctor)
    seq
      let $(r, \_) = top(evalStack)$ in
        THROW($r$)
```

First, an exception reference of type c is created and loaded on the *evalStack*. In the next step of our interpreter, the exception reference is thrown by the macro THROW.
To simplify the technical exposition, we make the following assumptions on the *exception classes referred to in our model*:

1. the exception classes are initialized[16];

2. the exception class objects occupy memory blocks of the same size, i.e. the function *sizeOf* has the same value for all the exception classes[17];

The first assumption ensures that the CREATEANDINITOBJ does not proceed with initialization of the exception class. One needs this guarantee for the successful subsequent execution of the THROW macro. The second assumption guarantees that the following never happens: CREATEANDINITOBJ invokes a RAISE(`OutOfMemoryException`) (see below the refinement of CREATEANDINITOBJ). If this happens, then CREATEANDINITOBJ would proceed with the "raising" of `OutOfMemoryException` instead of preparing the *evalStack* for the subsequent execution of THROW.

The ECMA standard states in [1, Partition III,§4.23] that the *Rethrow* instruction is only permitted within the body of a `catch` handler. However, the same instruction is allowed also within a handler region of a `filter` (see `Test` 5 in [3]) even if this does not match the previous statement. It throws the same exception reference that was caught by this handler. Due to the syntactical constraint that *Rethrow* should be used only in a `catch` handler (not of any exception handlers embedded within that `catch` handler), *exc*is always set, i.e. it is not *undef*.

In a `filter` region, there should be exactly one *EndFilter* instruction. This has to be the last instruction in the `filter` region. *EndFilter* takes an integer *val* from the stack that is supposed to be either 0 or 1. In the ECMA standard, 0 and 1 are assimilated with "continue search" and "execute handler", respectively. There is a discrepancy between [1, Partition I,§12.4.2.5] which states *Execution cannot be resumed at the location of the exception, except with a user-filtered handler* and [1, Partition III,§3.34] which states that the only possible return values from the

[16]The *initialized assumption* is justified by the following finding. The exception classes referred to in our model are marked with `beforefieldinit` and have no static fields. Accordingly, their `.cctor` initializers are empty modulo a *Return* instruction.

[17]The *same size assumption* seems reasonable since the exception classes have the same instance fields inherited from the `SystemException` class.

Fig. 12 The execution of $\text{CLR}_{\mathcal{E}}$ instructions

```
$\text{CLR}_E \equiv \textit{execScheme}_E(\text{EXECCLR}_E, \text{SWITCHCLR}, \text{EXCCLR})$

$\textit{execScheme}_E(\text{EXECCLR}, \text{SWITCHCLR}, \text{EXCCLR}) \equiv$
  if $\textit{switch} = \textit{ExcMech}$ then $\text{EXCCLR}$
  else $\textit{execScheme}_C(\text{EXECCLR}, \text{SWITCHCLR})$

$\text{EXECCLR}_E(\textit{instr}) \equiv$
  $\text{EXECCLR}_O(\textit{instr})$
  match $\textit{instr}$
    $\textit{Throw} \rightarrow$ let $(r, \_) = \textit{top}(\textit{evalStack}))$ in $\text{THROW}(r)$
    $\textit{Rethrow} \rightarrow$ $\text{THROW}(\textit{exc})$
    $\textit{EndFilter} \rightarrow$ let $(\textit{val}, \_) = \textit{top}(\textit{evalStack})$ in
                      if $\textit{val} = 1$ then
                        $\text{FOUNDHANDLER}$
                        $\text{RESET}(\textit{stackCursor}, \textit{top}(\textit{frameStack}))$
                      else $\text{GOTONXTHAN}$
                      $\text{POPFRAME}(0, [\,])$
                      $\textit{switch} := \textit{ExcMech}$
    $\textit{EndFinally} \rightarrow$ $\textit{evalStack} := [\,]$
                      $\textit{switch} := \textit{ExcMech}$
    $\textit{Leave}(\textit{pos}) \rightarrow$ $\textit{evalStack} := [\,]$
                      $\text{LOADREC}((\textit{Leave}, (\textit{frame}, 0), \textit{pos}))$
                      $\textit{switch} := \textit{ExcMech}$
    $\text{EXCSENSITIVEINSTR}_E$
```

filter are "exception_continue_search"(0) and "exception_execute_handler"(1). In other words, resumable exceptions are not (yet) supported contradicting Partition I.

If *val* is 1, then the `filter` handler to which *EndFilter* corresponds becomes the *handler* to handle the current exception in the pass *Unwind*. Remember that the `filter` handler is the handler pointed to by the *stackCursor*. The *stackCursor* is reset to be used for the pass *Unwind*: it will point into the topmost frame on *frameStack* which is actually the faulting frame. If *val* is 0, the *stackCursor* is incremented to point to the handler following our `filter` handler. Independent of *val*, the current frame is discarded to reestablish the context of the faulting frame. Note that we do not explicitly pop the *val* from the *evalStack* since anyway the global dynamic function *evalStack* is updated anyway in the next step through POPFRAME to the *evalStack*' of the faulting frame.

EndFinally terminates the execution of the handler region of a `finally`/`fault` handler. It empties the *evalStack* and transfers the control to EXCCLR. A *Leave* instruction empties the *evalStack* and loads on EXCCLR a pass record corresponding to a *Leave* pass.

Remark The reader might ask why the instructions *Throw*, *Rethrow* and *EndFilter* do not set the *evalStack*. The reason is that this set up, i.e. the emptying of the *evalStack*, is supposed to be either a *side-effect* (the case of the *Throw* and *Rethrow* instructions) or ensured for a *correct* CIL (the case of the *EndFilter* instruction). Thus, the *Throw* and *Rethrow* instructions pass the control to EXCCLR which, in a next step, will execute[18] a `catch`/`finally`/`fault` handler region or a `filter` code or propagates the exception in another frame. All these "events" will "clear" the *evalStack*. In case of *EndFilter*, the *evalStack* must contain exactly one item (an `int32` which is popped off by *EndFilter*). Note that this has to be checked by the bytecode verifier and not ensured by the exception handling mechanism.

In addition, $\text{CLR}_{\mathcal{E}}$ extends in the obvious way rules that raise run-time exceptions – see the group of rules $\text{EXCSENSITIVEINSTR}_E$ in Figure 13. A typical representative of the extension

[18]One can formally prove that there is such a "step" in the further run of the EXCCLR.

Fig. 13 The execution of $\text{CLR}_\mathcal{E}$ instructions (*continued*)

```
$\textsc{ExcSensitiveInstr}_E \equiv$
  $Execute(op) \rightarrow$ let $slots = take(evalStack, opNo(op))$ in
                 if $DivByZeroCase(op, slots)$ then
                   $\textsc{Raise}(\texttt{DivideByZeroException})$
                 elseif $OverflowCase(op, slots)$ then
                   $\textsc{Raise}(\texttt{OverflowException})$
                 elseif $InvalidNrCase(op, slots)$
                   $\textsc{Raise}(\texttt{ArithmeticException})$
  $CastClass(c) \rightarrow$ let $(r, \_) = top(evalStack)$ in
                 if $r \neq \texttt{null} \wedge actualTypeOf(r) \not\preceq c$ then
                   $\textsc{Raise}(\texttt{InvalidCastException})$
  $LoadLoc(\_) \rightarrow$ if $\neg zeroInit(meth)$ then
                 $halt :=$ "Verification Exception"
  $Call(\_, \_, c{::}m) \rightarrow$ if $instance(c{::}m) \wedge \neg reqinit(c)$ then
                   let $(\_, [r] \cdot \_) = getVals(evalStack, argNo(c{::}m))$ in
                     if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
  $LoadField(\_, c{::}f) \rightarrow$ if $instance(c{::}f) \wedge \neg reqinit(c)$ then
                   let $(r, \_) = top(evalStack)$ in
                     if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
  $StoreField(\_, c{::}f) \rightarrow$ if $instance(c{::}f) \wedge \neg reqinit(c)$ then
                   let $(\_, [r] \cdot \_) = getVals(evalStack, 2)$ in
                     if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
  $CallVirt(\_, \_, c{::}m) \rightarrow$ let $(\_, [r] \cdot \_) = getVals(evalStack, argNo(c{::}m))$ in
                   let $d{::}m = lookup(actualTypeOf(r), c{::}m)$ in
                     if $\neg reqinit(d)$ then
                       if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
  $\textsc{ArrayInstr}_E$

$\textsc{ArrayInstr}_E \equiv$
  $NewArray(t) \rightarrow$ let $(n, \_) = top(evalStack)$ in
                 if $n < 0$ then $\textsc{Raise}(\texttt{OverflowException})$
                 elseif $SpaceFor([n * sizeOf(t)]) = \emptyset$ then
                   if $SpaceFor([sizeOf(\texttt{OutOfMemoryException})]) \neq \emptyset$ then
                     $\textsc{Raise}(\texttt{OutOfMemoryException})$
                   else $halt :=$ "not enough space to be allocated"
  $LoadLength \rightarrow$ let $(r, \_) = top(evalStack)$ in
                 if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
  $LoadElem(\_) \rightarrow$ let $(\_, [r, i]) = getVals(evalStack, 2)$ in
                 if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
                 elseif $i < 0 \vee i \geq arraySize(r)$ then
                   $\textsc{Raise}(\texttt{IndexOutOfRangeException})$
  $StoreElem(t) \rightarrow$ let $(\_, [r, i, val]) = getVals(evalStack, 3)$ in
                 if $r = \texttt{null}$ then $\textsc{Raise}(\texttt{NullReferenceException})$
                 elseif $(i < 0 \vee i \geq arraySize(r))$ then
                   $\textsc{Raise}(\texttt{IndexOutOfRangeException})$
                 elseif $(t = \texttt{ref} \wedge val \neq \texttt{null} \wedge actualTypeOf(val) \not\preceq elemType(r))$ then
                   $\textsc{Raise}(\texttt{ArrayTypeMismatchException})$
```

is the definition of *Execute*. The extension of this rule raises a `DivideByZeroException`, if *op* is a division operator and the second topmost value on the stack is 0 (see the definition of the predicate *DivByZeroCase* in Section 2.2). An `OverflowException` is thrown if *op* is an operator that performs an overflow check (e.g. `add.ovf`) and the result of the operation cannot be represented in the result type.

CLR vs. JVM The JVM never indicates overflow when executing operations on integral types.

According to the definition of the predicate *InvalidNrCase* in Section 2.2, if *op* is the `ckfinite` operator and the topmost value on the stack is one of `NaN`, `+infinity` or `-infinity`, then an InvalidCastException is thrown. Such an exception is raised also if a division operator is applied to the smallest integer value and 0. The instruction *CastClass* throws an InvalidCastException if the actual (run-time) type of r is neither null nor compatible with the required type r. The attempt to load on the stack the value of a local variable, halts the machine[19] if the current method is not marked with the flag *zeroInit*. A NullReferenceException is thrown if the instructions *Call*, *LoadField*, *StoreField*, *CallVirt* applied to an instance member are called with a null reference.

Macro refinement We refine here the macro CreateAndInitObj used for the *NewObj* instruction in $\text{CLR}_{\mathcal{O}}$:

```
$\text{CreateAndInitObj}(c::\texttt{.ctor}) \equiv$
  if $\neg reqinit(c)$ then
    ...
    if $SpaceFor([sizeOf(c)]) = \emptyset$ then
      if $SpaceFor([sizeOf(\texttt{OutOfMemoryException})]) \neq \emptyset$ then
        Raise(OutOfMemoryException)
      else $halt :=$ “not enough space to be allocated”
```

If the creation of a class instance cannot succeed because there is no sufficient memory to be allocated on the heap, then the system throws an OutOfMemoryException. Note that the machine halts if there is not enough space to be allocated on the heap for an OutOfMemoryException object. In the real CLR this case corresponds to the case when the OutOfMemoryException is reported by the operating system.

Gap ECMA The *Call* instruction can throw a NullReferenceException even if this is not stated in the specification of call (see [1, Partition III,§3.19]). This is because the *Call* can invoke also instance methods and not only static methods.

The rules specific to the array instructions can also throw exceptions. The attempt to create a zero-based, one dimensional array with a negative number of elements produces an OverflowException. If there is insufficient memory to be allocated on the stack for the n array elements of type t, then an OutOfMemoryException is thrown. The same check as in the case of *NewObj* is performed to ensure that there is enough space for an OutOfMemoryException object. The instructions *LoadLength*, *LoadElem* and *StoreElem* throw a NullReferenceException if the target reference is null. If the instructions *LoadElem* and *StoreElem* attempt to access an array element whose index is negative or is greater than the array size, then the system raises an IndexOutOfRangeException. In addition, the *StoreElem* used for object references can throw an ArrayTypeMismatchException if the actual type of the value to be stored into the array element is not compatible with the element type of the array.

Mistake ECMA The standard states in [1, Partition III,§4.7] that *LoadElem* can also throw an ArrayTypeMismatchException. This is probably a mistake since anyway there is no value whose type shall be compared for compatibility with the type embedded in the instruction.

Beside the above rule extensions, we need to refine also the rules *Invoke* and *InitClass* of the switchCLR submachine defined in Section 2.3. Thus, when a method is invoked (including here also a .cctor), a StackOverflowException is thrown by the system if there is not enough memory to be allocated for arguments and local variables (see Fig. 14).

2.6 The $\text{CLR}_{\mathcal{P}}$ submodule

$\text{CLR}_{\mathcal{P}}$ extends $\text{CLR}_{\mathcal{O}}$ with pointer types, i.e. types whose values are memory addresses. $\text{CLR}_{\mathcal{P}}$ provides type-safe operations on pointers (e.g. “read”/“write” a value from/into the address

[19]In this case, the real CLR throws a VerificationException which will stop the execution of the program.

Fig. 14 The SWITCHCLR machine (*continued*)

```
SWITCHCLR ≡ match switch
  Invoke(tail, c::m, args) → ...
                                seq
                                  if $SpaceFor([sizes]) \neq \emptyset$ then
                                    ...
                                  else RAISE(StackOverflowException)
                              ...
  InitClass(c) → if classState(c) = Linked then
                   if $SpaceFor([sizes]) \neq \emptyset$ then
                     ...
                   else RAISE(StackOverflowException)
```

referenced by a pointer) and non-verifiable operations (e.g. initialize a block of memory to a given value, copy data from memory to memory). The main purpose of having pointer types is to permit methods to receive arguments and return values "by reference".

Environment and State The universe *CLRStackType* includes now also the special type & corresponding to *managed pointers*. *LoadIndType* describes the type of a value indirectly loaded from the memory on *evalStack* with *LoadInd*, while *StoreIndType* describes the type of a value indirectly stored into the memory with *StoreInd*.

$CLRStackType = \ldots \mid \&$ $SignedInt =$ `int8` | `int16` | `int32` | `int64`

$Float =$ `float32` | `float64` $UnsignedInt =$ `uint8` | `uint16` | `uint32` | `uint64`

$LoadIndType = SignedInt \mid UnsignedInt \mid Float \mid$ `native int` | `ref`

$StoreIndType = SignedInt \mid Float \mid$ `native int` | `ref`

In Section 2.3 we have introduced the external function *validAdr* to check addresses for validity. An address is invalid if it is `null` or is not in the range of *Adr* or is not "naturally aligned" for the target architecture or is "not mapped" into the process. An address is "naturally aligned" if it is aligned wrt the machine dependent `native int` type. Note that, as a consequence of the definition of *SpaceFor*, the results of all CIL instructions that return addresses (e.g. *LoadLocA* and *LoadArgA*) are valid.

Rules The rules for CLR$_\mathcal{P}$ are defined in Fig. 15 and 17. Assuming that the *zeroInit* flag of the current method is set, *LoadLocA*(n) pushes the address of the local variable indexed with n on the *evalStack*. If the *zeroInit* is not set, then the machine halts. Similarly, *LoadArgA*(n) pushes on the *evalStack* the address of the current method argument indexed with n. In JVM [6], one cannot take the address of local variables and arguments. The address of a static field is loaded using *LoadStaticA*. If the class which declares the static field is not yet initialized, the execution proceeds first with the class initialization. *LoadFieldA* is similar with *LoadStaticA* and can be used for both static and instance fields. If the field is an instance field and the object reference on *evalStack* is `null`, then a `NullReferenceException` is thrown. The *LoadInd* instruction takes the value of the topmost *evalStack* slot, which is supposed to be a pointer (address) and loads the value stored at this address. *StoreInd* takes the values of the two topmost *evalStack* slots, which are supposed to be a pointer (address) and a value. It then stores the value at the address. In both cases, of *LoadInd* and *StoreInd*, the address must be a valid address, otherwise a `NullReferenceException` is thrown.

The address of an array element can be loaded with the *LoadElemA* instruction described in Fig. 16. If the array reference on top of the *evalStack* is `null`, a `NullReferenceException` is thrown. Also if the index is not in the array range, a `IndexOutOfRangeException` is raised. An `ArrayTypeMismatchException` is thrown if the type embedded given in the *LoadElemA* does not match the element type of the array.

Call by-reference mechanism Unlike JVM, the CLR allows to pass to a method arguments

Fig. 15 The execution of CLR$_{\mathcal{P}}$ instructions

```
CLR$_P$ $\equiv$ $execScheme_E$(EXECCLR$_P$, SWITCHCLR, EXCCLR)

EXECCLR$_P$($instr$) $\equiv$
  EXECCLR$_E$($instr$)
  match $instr$
    $LoadLocA(n)$ $\to$ if $zeroInit(meth)$ then
                           $evalStack := evalStack \cdot [(locAdr(n), \&)]$
                           $pc := pc + 1$
                         else $halt$ := "Verification Exception"
    $LoadArgA(n)$ $\to$ $evalStack := evalStack \cdot [(argAdr(n), \&)]$
                         $pc := pc + 1$
    $LoadStaticA(\_, c{::}f)$ $\to$ if $initialized(c)$ then
                                    $evalStack := evalStack \cdot [(globals(c{::}f), \&)]$
                                    $pc := pc + 1$
                                  else $switch := InitClass(c)$
    $LoadFieldA(\_, c{::}f)$ $\to$ if $initialized(c)$ then
                                   let $(evalStack', [x]) = getVals(evalStack, 1)$ in
                                     if $static(c{::}f)$ then $evalStack := evalStack' \cdot [(globals(c{::}f), \&)]$
                                                              $pc := pc + 1$
                                     elseif $x \neq$ null then $evalStack := evalStack' \cdot [(fieldAdr(x, c{::}f), \&)]$
                                                               $pc := pc + 1$
                                     else RAISE(NullReferenceException)
                                 else $switch := InitClass(c)$
    $LoadInd(t)$ $\to$ let $(evalStack', [adr]) = getVals(evalStack, 1)$ in
                         if $validAdr(adr)$ then
                           $evalStack := evalStack' \cdot [(memVal(adr, t), CLRTypeOf(t))]$
                           $pc := pc + 1$
                         else RAISE(NullReferenceException)
    $StoreInd(t)$ $\to$ let $(evalStack', [adr, val]) = getVals(evalStack, 2)$ in
                          if $validAdr(adr)$ then
                            WRITEMEM($adr, t, val$)
                            $evalStack := evalStack'$
                            $pc := pc + 1$
                          else RAISE(NullReferenceException)
    ARRAYINSTR$_P$
    NONVERIFIABLE$_P$
```

Fig. 16 The execution of CLR$_{\mathcal{P}}$ instructions (*continued*)

```
ARRAYINSTR$_P$ $\equiv$
  $LoadElemA(c)$ $\to$ let $(evalStack', [r, i]) = getVals(evalStack, 2)$ in
                          if $r =$ null then RAISE(NullReferenceException)
                          elseif $(i < 0 \vee i \geq arraySize(r))$ then RAISE(IndexOutOfRangeException)
                          elseif $c \neq elemType(r)$ then RAISE(ArrayTypeMismatchException)
                          else
                            $evalStack := evalStack' \cdot [(elemAdr(r, i), \&)]$
                            $pc := pc + 1$
```

by-reference (the equivalent of the C# `ref` or Pascal `var` parameters). This is realized by passing (*by-value*) the address of the *by-reference* argument. Consequently, any assignment to the corresponding parameter actually modifies the corresponding caller's variable. The instructions in Fig. 15 offer support for computing variable addresses.

The following explanations apply to the non-verifiable CLR$_{\mathcal{P}}$ instructions in Fig. 17. The *InitBlock* instruction writes a given value of type `unsigned int8` in all the addresses of a block of memory. It takes the values of the three topmost *evalStack* slots, which are supposed to be,

Fig. 17 The execution of non-verifiable CLR$_{\mathcal{P}}$ instructions

```
NonVerifiable_P ≡
  $InitBlock \rightarrow$ let $(evalStack', [adr, val, size]) = getVals(evalStack, 3)$ in
        if $validAdr(adr)$ then
          forall $i = 0, size - 1$ do $mem(adr + i) := val$
          $evalStack := evalStack'$
          $pc := pc + 1$
        else Raise(NullReferenceException)
  $CopyBlock \rightarrow$ let $(evalStack', [dest\_adr, src\_adr, size]) = getVals(evalStack, 3)$ in
        if $validAdr(dest\_adr) \wedge validAdr(src\_adr)$ then
          if $\neg overlap(dest\_adr, src\_adr, size)$ then
            forall $i = 0, size - 1$ do $mem(dest\_adr + i) := mem(src\_adr + i)$
            $evalStack := evalStack'$
            $pc := pc + 1$
          else Report($UndefinedBehavior$)
        else Raise(NullReferenceException)
  $LocAlloc \rightarrow$ let $(evalStack', [size]) = getVals(evalStack, 1)$ in
        if $SpaceFor([size]) \neq \emptyset$ then
          choose $(adr) \in SpaceFor([size])$ do
            $evalStack := evalStack' \cdot [(adr, \&)]$
            if $zeroInit(meth)$ then
              forall $i = 0, size - 1$ do $mem(adr + i) := 0$
            $StackAdr := StackAdr \cup [adr, adr + size)$
          $pc := pc + 1$
        else Raise(StackOverflowException)
```

in order, a "pointer" (the block's first address), a value and the "size" of the block. It writes the value into a number of addresses given by the block's "size" starting with the address given by the "pointer". If the address is not valid, then a `NullReferenceException` is thrown. *CopyBlock* copies data from memory to memory. It takes the values of the three topmost *evalStack* slots, which are supposed to be, in order, a "destination" address, a "source" address and a "size" of a block of addresses. It then copies a number of bytes given by the block's "size" from the "source" address to the "destination" address. The "destination" and "source" addresses must be valid addresses and the "destination" and "source" areas shall not overlap. If they overlap, the behavior is undefined. If one of the addresses is not valid, then a `NullReferenceException` is thrown. The predicate *overlap* decides whether two blocks of addresses overlap:

$$overlap(adr_1, adr_2, size) \Leftrightarrow (adr_2 + size - 1 \geq adr_1) \wedge (adr_1 + size - 1 \geq adr_2)$$

The *LocAlloc* instruction is used for the compilation of a C♯ `stackalloc` statement (see [7] for details). It allocates space on the stack, in the so-called *local memory pool* (see [1, Partition I,§12.3.2.4] for details). It takes the value of the topmost *evalStack* slot, which represents the "size" of the block to be allocated. If there is sufficient space for a block of the given "size", the starting address of the arbitrary chosen block is loaded on the *evalStack*. The value 0 is stored in all the addresses of the block, only if the *zeroInit* flag of the current method is set. If there is not sufficient space to be allocated, then a `StackOverflowException` is thrown.

2.7 The CLR$_{\mathcal{VC}}$ submodule

CLR$_{\mathcal{VC}}$ extends CLR$_{\mathcal{P}}$ by value classes. Value classes are value types (in contrast to reference types) whose values (also known as "unboxed objects") are represented as *mappings* assigning values to the fields of the value class. A value class instance is usually allocated on the stack in contrast with the object class instances which are allocated on the heap. However, one can allocate a value type instance also on the heap but only within (e.g. as a field of) a boxed object. The value classes support the compilation of the C♯ structs. CLR$_{\mathcal{VC}}$ comes with a refinement of the *NewObj* instruction and also with new operations such as "boxing" and "unboxing".

Environment and State *ValueClass* is the universe of value class names. The universes *CLRStackType*, *Class*, *LoadIndType* and *StoreIndType* are refined to include also value classes.

$CLRStackType = \ldots \mid ValueClass$ $\qquad LoadIndType = \ldots \mid ValueClass$
$Class = \ldots \mid ValueClass$ $\qquad StoreIndType = \ldots \mid ValueClass$

The *fieldAdr* is refined to be applicable also to pointers referring to value class instances: $fieldAdr : Map((ObjRef \cup Adr) \times FRef, Adr)$. Thus, $fieldAdr(adr, vc::f)$ is the address of the instance field $vc::f$ of a value class instance stored at the address *adr*. Since the objects of a value type can be viewed as mappings which associate values to each instance field, we need to refine the definitions of *memVal* and WriteMem given in Section 2.2.

```
$memVal(adr, t) =$ if $t \in ObjType$ then $mem(adr)$
                   elseif $t \in ValueClass$ then
                      $\{f \mapsto memVal(fieldAdr(adr, f), type(f)) \mid f \in instFields(t)\}$
                   else $[mem(adr + i) \mid i \in [0..sizeOf(t) - 1]]$
```

```
$\textsc{WriteMem}(adr, t, val) \equiv$ if $t \in ObjType$ then $mem(adr) := val$
                              elseif $t \in ValueClass$ then
                                 forall $f \in instFields(t)$ do
                                    $\textsc{WriteMem}(fieldAdr(adr, f), type(f), val(f))$
                              else forall $i \in [0..sizeOf(t) - 1]$ do $mem(adr + i) := val(i)$
```

The instances of value classes can be "boxed" in the heap and addressed then by heap references. A boxed object on the heap embeds the actual type as well as a list of instance field addresses. When "unboxing" a "boxed" object, one needs its address on the heap. This is determined with $addressOf : Map(ObjRef, Adr)$ applied to the corresponding boxed object reference.

The instructions *LoadToken* and *ArgList* load handles on the *evalStack*. The handles are nothing else than instances of specific value types. However, they can be viewed as *constants* with respect to a given token (in case of *LoadToken*) or a given method (in case of *ArgList*). Because of this aspect, we do not model here these instructions.

Rules Fig. 18 describes instructions for value types (including the value classes) and how is applied the instruction *NewObj* to create a value class instance. Such an instance is usually allocated as an argument or local variable and then initialized with *InitObj*. Unlike instances of object classes, they are allocated on the stack (by means of the *stackalloc* macro). The *stackalloc* chooses a block of unallocated memory addresses whose length is given by the value type's size. The field addresses are computed using the field offsets. Since the fields can be of value class types, one needs to compute also the addresses of the their corresponding instance fields. This is performed by the recursively defined macro AllocFields.

```
let $adr = stackalloc(vc)$ in $P \equiv$
   choose $(adr) \in SpaceFor([n])$ do
      $StackAdr := StackAdr \cup [adr, adr + n)$
      $\textsc{AllocFields}(adr, vc)$
   seq $P$
   where $n = sizeOf(vc)$
```

```
$\textsc{AllocFields}(adr, t) \equiv$
   if $t \in ValueClass$ then
      forall $f \in instFields(t)$ do
         let $a = adr + fieldOffSet(t, f)$ in
            $fieldAdr(adr, f) := a$
            $\textsc{AllocFields}(a, type(f))$
```

One aspect that differentiates the value classes from the object classes is concerning the instance methods invocation, in particular constructors invocation. The argument 0 of instance methods defined by a value class *vc* is of the pointer type *vc*&. Therefore the definition of *argTypes* is refined as follows: for every instance method reference $vc::m$ defined by the value class *vc*

$argTypes(vc::m) = [vc\&] \cdot paramTypes(vc::m)$

CLR implementation mistake The ECMA standard states in [1, Partition I,§8.9.5], that if a type is not marked with `beforefieldinit`, then its type initializer is executed if, in particular,

Fig. 18 The execution of CLR$_{\mathcal{VC}}$ instructions

```
$\text{CLR}_{VC} \equiv \mathit{execScheme}_E(\text{EXECCLR}_{VC}, \text{SWITCHCLR}, \text{EXCCLR})$

$\text{EXECCLR}_{VC}(\mathit{instr}) \equiv$
  $\text{EXECCLR}_P(\mathit{instr})$
  match $\mathit{instr}$
    $\mathit{NewObj}(vc\text{::}\texttt{.ctor}) \rightarrow$ if $vc \in \mathit{ValueClass}$
        if $\neg \mathit{reqinit}(vc)$ then
          if $\mathit{SpaceFor}([\mathit{sizeOf}(vc)]) \neq \emptyset$ then
            let $(\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{paramNo}(vc\text{::}\texttt{.ctor}))$ in
              let $\mathit{adr} = \mathit{stackalloc}(vc)$ in
                $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{adr}, vc), vc)]$
                forall $f \in \mathit{instFields}(vc)$ do
                  $\text{WRITEMEM}(\mathit{fieldAdr}(\mathit{adr}, f), \mathit{type}(f), \mathit{defVal}(\mathit{type}(f)))$
                $\mathit{switch} := \mathit{Invoke}(\mathit{False}, vc\text{::}\texttt{.ctor}, [\mathit{adr}] \cdot \mathit{vals})$
          else $\text{RAISE}(\texttt{OutOfMemoryException})$
        else $\mathit{switch} := \mathit{InitClass}(vc)$
    $\mathit{InitObj}(vt) \rightarrow$ let $(\mathit{evalStack}', [\mathit{adr}]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
        $\text{WRITEMEM}(\mathit{adr}, vt, \mathit{defVal}(vt))$
        $\mathit{evalStack} := \mathit{evalStack}'$
        $pc := pc + 1$
    $\mathit{CopyObj}(vt) \rightarrow$ let $(\mathit{evalStack}', [\mathit{dest\_adr}, \mathit{src\_adr}]) = \mathit{getVals}(\mathit{evalStack}, 2)$ in
        if $\mathit{validAdr}(\mathit{dest\_adr}) \wedge \mathit{validAdr}(\mathit{src\_adr})$ then
          $\text{WRITEMEM}(\mathit{src\_adr}, vt, \mathit{memVal}(\mathit{dest\_adr}, vt))$
          $\mathit{evalStack} := \mathit{evalStack}'$
          $pc := pc + 1$
        else $\text{RAISE}(\texttt{NullReferenceException})$
    $\mathit{Box}(vt) \rightarrow$ if $\mathit{SpaceFor}([\mathit{sizeOf}(vt)]) \neq \emptyset$ then
        let $(\mathit{evalStack}', [\mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
          let $r = \mathit{new}(\mathit{ObjRef})$ and $\mathit{adr} = \mathit{heapalloc}(vt)$ in
            $\mathit{actualTypeOf}(r) := vt$
            $\mathit{addressOf}(r) := \mathit{adr}$
            $\text{WRITEMEM}(\mathit{adr}, vt, \mathit{val})$
            $\mathit{evalStack} := \mathit{evalStack}' \cdot [(r, O)]$
            $pc := pc + 1$
      else $\text{RAISE}(\texttt{OutOfMemoryException})$
    $\mathit{Unbox}(vt) \rightarrow$ let $(\mathit{evalStack}', [r]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
        if $\mathit{actualTypeOf}(r) = vt$ then
          $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{addressOf}(r), \&)]$
          $pc := pc + 1$
        else $\text{RAISE}(\texttt{InvalidCastException})$
    $\mathit{SizeOf}(vt) \rightarrow$ $\mathit{evalStack} := \mathit{evalStack} \cdot [(\mathit{sizeOf}(vt), \texttt{int32})]$
        $pc := pc + 1$
```

an access to an instance field of that type occurs. However, this is not performed in the following case. Suppose that v is a local variable of a method m and its type is a value type P that is not marked with `beforefieldinit`. If our method has the "zero init" flag set, then v is automatically zero initialized upon m's entry. If we access an instance field of v either by *LoadField* or *StoreField*, then P's initializer is surprisingly not executed (contradicting the above ECMA statement). The reason might be that, upon m's entry, v is zero initialized and consequently also all its instance fields.

The instruction *InitObj* initializes an instance of a value type. It takes the value of the topmost *evalStack* slot, which is supposed to be a pointer to a value type instance. Then, it initializes all the instance fields of the instance to the default value of the proper type by means of the recursively defined macro WRITEMEM. The *CopyObj* instruction copies an instance of a value type. It takes two pointers from the *evalStack* and copies the value type object stored at the address given by the pointer to the address given by the second pointer. However, if one pointer refers to an invalid address, a `NullReferenceException` is thrown.

The *Box* instruction turns a value type instance into a heap-allocated object "by copying", while *Unbox* performs the inverse coercion. *Box* checks first if there sufficient memory to make the conversion. Then, it takes a value type instance from the *evalStack*, it creates an object reference and allocates on the heap through the *heapalloc*, a block of memory of length given by the value type's size.

```
let $r = new(ObjRef)$ in $P \equiv$
  import $r$ do
    $ObjRef(r) := True$
  seq $P$
```

```
let $adr = heapalloc(vt)$ in $P \equiv$
  choose $(adr) \in SpaceFor([n])$ in
    $HeapAdr := HeapAdr \cup [adr, adr + n)$
    $\textsc{AllocFields}(adr, vt)$
  seq $P$
  where $n = sizeOf(vt)$
```

Note that *Box* copies the data from the value type instance into the newly allocated object. The *Unbox* instruction takes an object reference to a boxed object from the *evalStack* and extracts the value type instance from it. However, the value pushed on the *evalStack* is a pointer representing the address (given by *addressOf*) of the value type instance that is present inside of the boxed object.

The *SizeOf* instruction loads on the *evalStack* the size of a given value type.

2.8 The CLR$_{\mathcal{TR}}$ submachine

CLR$_{\mathcal{TR}}$ extends CLR$_{\mathcal{VC}}$ with typed references. There is a special kind of method parameters called "typed reference parameters". The typed references support languages like Visual Basic that require by-reference passing of unboxed data to methods that are not statically restricted as to the type of data they accept. Therefore, these languages require a way of passing both the address of the argument together with its the static type.

Environment and State The typed references can be seen as opaque descriptors described by the universe *TypedRef*.

The function *typedRefAdr* : *Map*(*TypedRef*, *Adr*) returns the address embedded in a typed reference. The type transmitted with a typed reference is recorded by the function *typedRefType* : *Map*(*TypedRef*, *Type*). Since the instruction *RefAnyType* needs handles associated to types, we assume that the function *handleOf* : *Map*(*Type*, *Val*) returns the `RuntimeTypeHandle` object associated to a given type.

The universe *Val* should contain also the universe `typedref` which is the mapping of the library `System.TypedReference`. The later is derived from `ValueType`, which is derived in turn from `Object`. However, the runtime disallows any conversion of the type to `Object` or any `ValueType`. Otherwise, this would have circumvented the restriction that managed pointers cannot be members within value classes and classes — an ability that would greatly hurt the performance of garbage collection.

Rules Fig. 19 defines the instructions that support the typed reference handling. A typed reference is created using the *MkRefAny* instruction. The new typed reference embeds the pointer on top of the *evalStack* and also the type given in the instruction. We use the following macro to yield a fresh typed reference:

```
let $tr = new(TypedRef)$ in $P \equiv$
  import $tr$ do $TypedRef(tr) := True$
  seq $P$
```

The *RefAnyType* instruction expects on top of the *evalStack* a typed reference.

Mistake ECMA. *RefAnyType* instruction pushes on the *evalStack* not the metadata token of the type of a `typedref` as the ECMA standard states in [1, Partition III,§4.21], but an instance

Fig. 19 The execution of $\mathrm{CLR}_{\mathcal{TR}}$ instructions

```
$\mathrm{CLR}_{TR} \equiv \mathit{execScheme}_E(\textsc{execCLR}_{TR}, \textsc{switchCLR}, \textsc{excCLR})$

$\textsc{execCLR}_{TR}(\mathit{instr}) \equiv$
  $\textsc{execCLR}_{VC}(\mathit{instr})$
  match $\mathit{instr}$
    $\mathit{MkRefAny}(t) \rightarrow$ let $\mathit{tr} = \mathit{new}(\mathit{TypedRef})$ in
        let $(\mathit{evalStack}', [\mathit{adr}]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
          $\mathit{typedRefAdr}(\mathit{tr}) := \mathit{adr}$
          $\mathit{typedRefType}(\mathit{tr}) := t$
          $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{tr}, \texttt{TypedReference})]$
          $\mathit{pc} := \mathit{pc} + 1$

    $\mathit{RefAnyType} \rightarrow$ let $(\mathit{evalStack}', [\mathit{tr}]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
        let $t = \mathit{typedRefType}(\mathit{tr})$ in
          $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{handleOf}(t), \texttt{RuntimeTypeHandle})]$
          $\mathit{pc} := \mathit{pc} + 1$

    $\mathit{RefAnyVal}(t) \rightarrow$ let $(\mathit{evalStack}', [\mathit{tr}]) = \mathit{getVals}(\mathit{evalStack}, 1)$ in
        if $t \neq \mathit{typedRefType}(\mathit{tr})$ then $\textsc{Raise}(\texttt{InvalidCastException})$
        else
          $\mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{typedRefAdr}(\mathit{tr}), \&)]$
          $\mathit{pc} := \mathit{pc} + 1$
```

of the `RuntimeTypeHandle` value type corresponding to the type described by the token. A token on the evaluation stack would be useless: no instruction pops a token off the *evalStack*.

The *RefAnyVal* instruction retrieves the address embedded in the typed reference on top of the *evalStack*. However, an `InvalidCastException` is thrown if the type given in the *RefAnyVal* instruction is not identical to the type stored in the typed reference.

2.9 The $\mathrm{CLR}_{\mathcal{MP}}$ submodule

$\mathrm{CLR}_{\mathcal{MP}}$ extends $\mathrm{CLR}_{\mathcal{TR}}$ by method pointers. The CLR instructions handling method pointers support the compilation of the C++ method pointers and C♯ delegates.

Environment and State method pointers are a special type of pointers but they are treated fundamentally different than the rest of the pointers. A method pointer is a pointer to a function entry point. It reliably identifies the function assigned to the pointer. Thus, if one knows the method entry point, one can determine the function signature, the calling convention, local signature.

Therefore, one can see a method pointer as a function identifier. This identifier can be regarded as pointing to a real address or directly to the corresponding fully qualified function name. Since the use of an address representing a method pointer is anyway limited, we represent the method pointers as elements of the universe *FunctionPtr* that point to the corresponding function references. For a method pointer, the function $\mathit{methodOf} : \mathit{Map}(\mathit{FunctionPtr}, \mathit{MRef})$ returns the corresponding function reference.

The delegates are the object-oriented type-safe equivalent of function pointers. Delegates are created by defining a subclass of the class `MulticastDelegate` which in turn is derived from `Delegate`. A delegate instance is then simply an instance of this object class and is created with the *NewObj* instruction introduced in $\mathrm{CLR}_{\mathcal{O}}$. However, there are some constraints on the definition of such a "delegate class": it should have a method `Invoke` with appropriate parameters, and each instance of a delegate forwards calls to its `Invoke` method to a compatible static or instance method (represented as a method pointer) on a particular object. The object and method to which it delegates are known as the *target object* and the *target method* and are

Fig. 20 The execution of CLR$_{\mathcal{MP}}$ instructions

```
$\mathrm{CLR}_{MP} \equiv execScheme_E(\textsc{execCLR}_{MP}, \textsc{switchCLR}, \textsc{excCLR})$

$\textsc{execCLR}_{MP}(instr) \equiv$
  $\textsc{execCLR}_{TR}(instr)$
  match $instr$
    $LoadFtn(c{::}m) \rightarrow$ let $fp = new(FunctionPtr, c{::}m)$ in
                        $evalStack := evalStack \cdot [(fp, \texttt{native int})]$
                        $pc := pc + 1$
    $LoadVirtFtn(c{::}m) \rightarrow$ let $(evalStack', [r]) = getVals(evalStack, 1)$ in
                          if $r \neq \texttt{null}$ then
                            let $fp = new(FunctionPtr, lookup(actualTypeOf(r), c{::}m))$ in
                              $evalStack := evalStack' \cdot [(fp, \texttt{native int})]$
                              $pc := pc + 1$
                          else $\textsc{Raise}(\texttt{NullReferenceException})$
    $\textsc{NonVerifiable}_{MP}$
```

chosen when the delegate object is created.
Mistake ECMA. In [1, Partition II,§13.6], it is stated that the `Invoke` method a delegate type is supposed to define shall be virtual and have the same signature (return type, parameter types, calling convention, and modifiers) as the target method. On the other hand, since the `Invoke` is virtual, its calling convention necessarily contains the `instance` attribute (see [1, Partition II,§14.3]). Therefore, if the calling conventions of the `Invoke` and target method match, then would be hopeless to create delegate instances that refer to a static method. This is obviously not true. Therefore, one should not be stated that the calling conventions have to match.
Rules Fig. 20 shows the rules for the CLR$_{\mathcal{MP}}$ submachine. The *LoadFtn* instruction pushes on the *evalStack* a pointer to the given method as an unmanaged pointer (type `native int`). The method embedded in the *LoadFtn* can be a static or an instance method. Note that the instruction *LoadFtn* used with a static method, does not trigger the initialization of the class that defines the method. A method pointer is created through the following macro:

```
let $fp = new(FunctionPtr, c{::}m)$ in $P \equiv$
  import $fp$ do
    $FunctionPtr(fp) := True$
    $methodOf(fp)\ \ := c{::}m$
  seq $P$
```

A method pointer for a virtual method can be obtained with the instruction *LoadVirtFtn*. This instruction pops from the *evalStack* an object reference. It then loads on the *evalStack* a pointer to the virtual method determined by the actual type of the object reference and the given method.
Gap ECMA The ECMA standard specifies in [1, Partition III,§4.17] that the instruction *LoadVirtFtn* cannot throw an exception. When the function pointer is determined, *LoadVirtFtn* has to know the actual type of the object reference on the *evalStack*. If this is `null`, one cannot determine the virtual method whose pointer is needed. This aspect justifies why a `NullReferenceException` is thrown. This behavior is however not specified in the ECMA standard.

The method pointers on the *evalStack* can be used with the non-verifiable instruction *CallI* described in Fig. 21 or to construct delegates with *NewObj*. A *CallI* instruction calls a method pointer with a corresponding number of arguments. The method pointer as well as the arguments whose number is determined by the signature present in the *CallI* instruction, shall be present on the *evalStack*. Note that, in case of an instance method, the list of arguments includes also

Fig. 21 The execution of non-verifiable CLR$_{\mathcal{MP}}$ instructions

```
$\textsc{NonVerifiable}_{MP} \equiv$
  $CallI(tail, sig) \rightarrow$ let $(evalStack', args \cdot [fp]) = getVals(evalStack, 1 + argNo(sig))$ in
                                  let $c{::}m = methodOf(fp)$ in
                                    if $\neg reqinit(c)$ then
                                      $switch := Invoke(tail, c{::}m, args)$
                                    else $switch := InitClass(c)$
```

the `this` argument. The *CallI* instruction can have a `tail` prefix. If the method pointer corresponds to a method declared by a class that needs to be initialized, then the class is initialized. Surprisingly, no `NullReferenceException` is thrown if *CallI* is invoked with a `null` object and a pointer to an instance method. Our experiments showed that the `NullReferenceException` is eventually thrown, when the object reference is accessed. This behavior is also not specified in the ECMA documentation. As a side remark, one can observe that, when invoking methods in the verifiable code, it is ensured that the `this` pointer is non-`null` while in the valid code, the `this` pointer can also be `null` (see the *CallI* instruction).

Remark In the ASM model for C♯ defined in [7], the destination method is determined at time the delegate object is invoked. This does not conform with the semantics of the CIL code corresponding to a C♯ delegate object invocation. Correct is that the destination method is determined when the delegate object is created (with the *NewObj* instruction). However, the two semantics are equivalent for the purposes of the model in [7].

3 The CLR$_{\mathcal{RM}}$ submodule

The submodule CLR$_{\mathcal{RM}}$ extends CLR$_{\mathcal{MP}}$ with specifications for runtime managed functions. Such methods are "managed by the runtime", i.e. their implementation is provided by the runtime and not by the user code (the method body has no bytecode instructions). Examples are the `Invoke` method and the constructor `.ctor` that every delegate class is supposed to declare. In our model, the runtime managed methods are specified using ASM rules. As examples, we describe in Fig. 22 the two mentioned examples. There are no instructions introduced by CLR$_{\mathcal{RM}}$.

Rules To extend the submachine CLR$_{MP}$ with runtime managed methods, we refine the $execScheme_E$ defined in CLR$_{\mathcal{E}}$ as showed in Fig. 22. The instance constructor `.ctor` of a delegate class shall take exactly two parameters. The first parameter is supposed to be of type `Object` and the second parameters of type `IntPtr`[20]. The first argument shall be the instance of the class that defined the target method while the second argument shall be a method pointer to the called method. However, the constructor expects as the "argument 0" also a delegate instance whose fields `_target` and `_methodPtr` are set to the target object and target method pointer, respectively. If the target object is `null`, then the exception `NullReferenceException` is thrown.

The method `Invoke` defined by a delegate class shall have the same return type and parameter types as the target method. The target object can be `null` only in case the target method is `static`. The reason is that, before the creation of the delegate instance, the `.ctor` constructor checks the target object against `null`. Therefore, the list of arguments which is passed to the target method contains the target object only if there exists one, i.e. the field `_target` is not `null`. If the class which declares the target method requires initialization, then it is initialized.

[20]The library type `System.IntPtr` corresponds to the CLR `native int` type.

Fig. 22 The implementation via ASM rules of runtime managed functions

```
$\text{CLR}_{RM} \equiv execScheme_{RM}(\text{EXECCLR}_{RM}, \text{SWITCHCLR}, \text{EXCCLR})$

$execScheme_{RM}(\text{EXECRUNMAN}, \text{SWITCHCLR}, \text{EXCCLR}) \equiv$
  **if** $switch = ExcMech$ **then** $\text{EXCCLR}$
  **elseif** $switch \neq Noswitch$ **then** $\text{SWITCHCLR}$
  **else** $\text{INITIALIZECLASS}$ **or if** $isRunMan(meth)$ **then** $\text{EXECRUNMAN}$
                                   **else** $\text{EXECCLR}_{MP}(code(pc))$

$execScheme_{E}(\text{EXECCLR}, \text{SWITCHCLR}, \text{EXCCLR}) \equiv$
  **if** $switch = ExcMech$ **then** $\text{EXCCLR}$
  **else** $execScheme_{C}(\text{EXECCLR}, \text{SWITCHCLR})$

$\text{EXECCLR}_{RM} \equiv$
  **if** $methNm(meth) =$ .ctor **then**
    **let** $del = memVal(argAdr(0), classNm(meth))$ **in**
      **let** $r = memVal(argAdr(1),$ object$)$ **in**
        **let** $fp = memVal(argAdr(2),$ native int$)$ **in**
          **if** $\neg static(methodOf(fp)) \wedge r =$ null **then**
            $\text{RAISE}($NullReferenceException$)$
          **else** $\text{WRITEMEM}(fieldAdr(del,$ Delegate::_target$),$ object$, r)$
               $\text{WRITEMEM}(fieldAdr(del,$ Delegate::_methodPtr$),$ native int$, fp)$
  **elseif** $methNm(meth) =$ Invoke **then**
    **let** $del = memVal(argAdr(0), classNm(meth))$ **in**
      **let** $r = memVal(fieldAdr(del,$ Delegate::_target$),$ object$)$ **in**
        **let** $fp = memVal(fieldAdr(del,$ Delegate::_methodPtr$),$ native int$)$ **in**
          **let** $slots = [memVal(argAdr(i, argTypes(i))) \mid i = 1, argNo(meth)]$ **in**
            **let** $args =$ **if** $r \neq$ null **then** $[r] \cdot slots$ **else** $slots$ **in**
              **let** $c = classNm(methodOf(fp))$ **in**
                **if** $\neg reqinit(c)$ **then**
                  $switch := Invoke(False, methodOf(fp), args)$
                **else** $switch := InitClass(c)$
```

4 Conclusion and Future Work

We have provided a modular definition of the CLR virtual machine in terms of ASM model. The abstract model takes the form of an abstract interpreter for a hierarchy of nine stepwise refined CLR program layers. We assume that the inputs of the interpreter are CIL bytecode programs successfully loaded and linked (i.e. prepared and verified to satisfy the required link-time constraints). As a next step of our project, we propose ourselves to relax the assumption that the CIL code is verified by accompanying the execution machine with a run-time checking machine. This *defensive* machine is going to serve for proving the *soundness* and *completeness* of the CLR bytecode verifier.

References

[1] Common Language Infrastructure (CLI), Standard ECMA–335, Second Edition. Web pages at `http://www.ecma-international.org/publications/`.

[2] Nicu G. Fruja and Egon Börger. Analysis of the .NET CLR Exception Handling Mechanism. *3rd .NET Technologies Conference*, 2005.

[3] Nicu G. Fruja. Experiments with CLR. Example programs to determine the meaning of CLR features not specified by the ECMA standard. Available at http://www.inf.ethz.ch/personal/fruja/publications/clrexctests.pdf

[4] Andrew D. Gordon and Don Syme. Typing a Multi-Language Intermediate Code. Microsoft Technical Report MSR-TR-2000-106, 2000.

[5] K. J. Gough. Stacking them up: a Comparison of Virtual Machines. ACM International Conference Proceeding Series. pag. 55–61. IEEE Computer Society, Washington, DC, USA, 2001.

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

[7] E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C♯. To appear in *Journal Theoretical Computer Science*, 2005.

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

[9] J. Gough. Compiling for the .NET. Common Language Runtime (CLR). Prentice Hall, 2002.

[10] Nicu G. Fruja. Specification and Implementation Problems for C♯. Proceedings of the Workshop on Abstract State Machines (*ASM'04*), Germany, 2004.

[11] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C♯. *Journal of Object Technology*, vol. 3, no. 9, 2004.

[12] Nicu G. Fruja. Type Safety in C♯ and .NET CLR. PhD Thesis in preparation.

[13] Horatiu V. Jula and Nicu G. Fruja. An Executable Specification of C♯. Submitted to *ASM'05*, 2005.

[14] C. Marrocco. An Executable Specification of the .NET CLR. Diploma Thesis guided by Nicu G. Fruja, ETH Zürich, in preparation.

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

5 Appendix

5.1 The Basic Instructions

IL Bytecode instruction	Abstract instruction
`add`	*Execute*(`add`)
`add.ovf`	*Execute*(`add.ovf`)
`add.ovf.un`	*Execute*(`add.ovf.un`)
`and`	*Execute*(`and`)
`arglist`	*ArgList*
`beq` t	*Cond*(`eq`,t)
`beq.s` t	*Cond*(`eq`,t)
`bge` t	*Cond*(`ge`,t)
`bge.s` t	*Cond*(`ge`,t)
`bge.un` t	*Cond*(`ge.un`,t)
`bge.un.s` t	*Cond*(`ge.un`,t)
`bgt` t	*Cond*(`gt`,t)
`bgt.s` t	*Cond*(`gt`,t)
`bgt.un` t	*Cond*(`gt.un`,t)
`bgt.un.s` t	*Cond*(`gt.un`,t)
`ble` t	*Cond*(`le`,t)
`ble.s` t	*Cond*(`le`,t)
`ble.un` t	*Cond*(`le.un`,t)
`ble.un.s` t	*Cond*(`le.un`,t)
`blt` t	*Cond*(`lt`,t)
`blt.s` t	*Cond*(`lt`,t)
`blt.un` t	*Cond*(`lt.un`,t)
`blt.un.s` t	*Cond*(`lt.un`,t)
`bne.un` t	*Cond*(`ne.un`,t)
`bne.un.s` t	*Cond*(`ne.un`,t)
`br` t	*Branch*(t)

IL Bytecode instruction	Abstract instruction
`br.s` *t*	*Branch*(*t*)
`break`	not described
`brfalse` *t*	*Cond*(`ifzero`,*t*)
`brfalse.s` *t*	*Cond*(`ifzero`,*t*)
`brnull` *t*	*Cond*(`ifnull`,*t*)
`brnull.s` *t*	*Cond*(`ifnull`,*t*)
`brzero` *t*	*Cond*(`ifzero`,*t*)
`brzero.s` *t*	*Cond*(`ifzero`,*t*)
`brtrue` *t*	*Cond*(`ifnonzero`,*t*)
`brtrue.s` *t*	*Cond*(`ifnonzero`,*t*)
`brinst` *t*	*Cond*(`ifnonnull`,*t*)
`brinst.s` *t*	*Cond*(`ifnonnull`,*t*)
`call` *mref*/*rt*	*Call*(`false`,*rt*,*mref*)
`tail.call` *mref*/*rt*	*Call*(`true`,*rt*,*mref*)
`calli` *sig*	*CallI*(`false`,*sig*)
`tail.calli` *sig*	*CallI*(`true`,*sig*)
`ceq`	*Execute*(`ceq`)
`cgt`	*Execute*(`cgt`)
`ckfinite`	*Execute*(`ckfinite`)
`cgt.un`	*Execute*(`cgt.un`)
`clt`	*Execute*(`clt`)
`clt.un`	*Execute*(`clt.un`)
`conv.i1`	*Execute*(`conv.i1`)
`conv.i2`	*Execute*(`conv.i2`)
`conv.i4`	*Execute*(`conv.i4`)
`conv.i8`	*Execute*(`conv.i8`)
`conv.r4`	*Execute*(`conv.r4`)
`conv.r8`	*Execute*(`conv.r8`)
`conv.u1`	*Execute*(`conv.u1`)
`conv.u2`	*Execute*(`conv.u2`)

IL Bytecode instruction	Abstract instruction
`conv.u4`	*Execute*(`conv.u4`)
`conv.u8`	*Execute*(`conv.u8`)
`conv.i`	*Execute*(`conv.i`)
`conv.u`	*Execute*(`conv.u`)
`conv.r.un`	*Execute*(`conv.r.un`)
`conv.ovf.i1`	*Execute*(`conv.ovf.i1`)
`conv.ovf.i2`	*Execute*(`conv.ovf.i2`)
`conv.ovf.i4`	*Execute*(`conv.ovf.i4`)
`conv.ovf.i8`	*Execute*(`conv.ovf.i8`)
`conv.ovf.u1`	*Execute*(`conv.ovf.u1`)
`conv.ovf.u2`	*Execute*(`conv.ovf.u2`)
`conv.ovf.u4`	*Execute*(`conv.ovf.u4`)
`conv.ovf.u8`	*Execute*(`conv.ovf.u8`)
`conv.ovf.i`	*Execute*(`conv.ovf.i`)
`conv.ovf.u`	*Execute*(`conv.ovf.u`)
`conv.ovf.i1.un`	*Execute*(`conv.ovf.i1.un`)
`conv.ovf.i2.un`	*Execute*(`conv.ovf.i2.un`)
`conv.ovf.i4.un`	*Execute*(`conv.ovf.i4.un`)
`conv.ovf.i8.un`	*Execute*(`conv.ovf.i8.un`)
`conv.ovf.u1.un`	*Execute*(`conv.ovf.u1.un`)
`conv.ovf.u2.un`	*Execute*(`conv.ovf.u2.un`)
`conv.ovf.u4.un`	*Execute*(`conv.ovf.u4.un`)
`conv.ovf.u8.un`	*Execute*(`conv.ovf.u8.un`)
`conv.ovf.i.un`	*Execute*(`conv.ovf.i.un`)
`conv.ovf.u.un`	*Execute*(`conv.ovf.u.un`)
`cpblk`	*CopyBlock*
`div`	*Execute*(`div`)
`div.un`	*Execute*(`div.un`)
`dup`	*Dup*
`endfilter`	*EndFilter*

IL Bytecode instruction	Abstract instruction
`endfault`	*EndFinallt*
`endfinally`	*EndFinally*
`initblk`	*InitBlock*
`jmp` *meth*	*Jmp*(*meth*)
`ldarg` *num*	*LoadArg*(*num*)
`ldarg.s` *num*	*LoadArg*(*num*)
`ldarg.0`	*LoadArg*(0)
`ldarg.1`	*LoadArg*(1)
`ldarg.2`	*LoadArg*(2)
`ldarg.3`	*LoadArg*(3)
`ldarga` *argNum*	*LoadArgA*(*argNum*)
`ldarga.s` *argNum*	*LoadArgA*(*argNum*)
`ldc.i4` *num*	*Const*(`int32`,*num*)
`ldc.i8` *num*	*Const*(`int64`,*num*)
`ldc.r4` *num*	*Const*(`float32`,*num*)
`ldc.r8` *num*	*Const*(`float64`,*num*)
`ldc.i4.0`	*Const*(`int32`,0)
`ldc.i4.1`	*Const*(`int32`,1)
`ldc.i4.2`	*Const*(`int32`,2)
`ldc.i4.3`	*Const*(`int32`,3)
`ldc.i4.4`	*Const*(`int32`,4)
`ldc.i4.5`	*Const*(`int32`,5)
`ldc.i4.6`	*Const*(`int32`,6)
`ldc.i4.7`	*Const*(`int32`,7)
`ldc.i4.8`	*Const*(`int32`,8)
`ldc.i4.m1`	*Const*(`int32`,-1)
`ldc.i4.M1`	*Const*(`int32`,-1)
`ldc.i4.s` *num*	*Const*(`int32`,*num*)
`ldftn` *mref*	*LoadFtn*(*mref*)
`ldind.i1`	*LoadInd*(`int8`)

IL Bytecode instruction	Abstract instruction
`ldind.i2`	*LoadInd*(`int16`)
`ldind.i4`	*LoadInd*(`int32`)
`ldind.i8`	*LoadInd*(`int64`)
`ldind.u1`	*LoadInd*(`uint8`)
`ldind.u2`	*LoadInd*(`uint16`)
`ldind.u4`	*LoadInd*(`uint32`)
`ldind.r4`	*LoadInd*(`float32`)
`ldind.u8`	*LoadInd*(`uint64`)
`ldind.r8`	*LoadInd*(`float64`)
`ldind.i`	*LoadInd*(`native int`)
`ldind.ref`	*LoadInd*(`object ref`)
`ldloc` *indx*	*LoadLoc*(*indx*)
`ldloc.s` *indx*	*LoadLoc*(*indx*)
`ldloc.0`	*LoadLoc*(0)
`ldloc.1`	*LoadLoc*(1)
`ldloc.2`	*LoadLoc*(2)
`ldloc.3`	*LoadLoc*(3)
`ldloca` *indx*	*LoadLocA*(*indx*)
`ldloca.s` *indx*	*LoadLocA*(*indx*)
`ldnull`	*Const*(*Null*,`null`)
`leave` *t*	*Leave*(*t*)
`leave.s` *t*	*Leave*(*t*)
`localloc`	*LocAlloc*
`mul`	*Execute*(`mul`)
`mul.ovf`	*Execute*(`mul.ovf`)
`mul.ovf.un`	*Execute*(`mul.ovf.un`)
`neg`	*Execute*(`neg`)
`nop`	not described
`not`	*Execute*(`not`)
`or`	*Execute*(`or`)

IL Bytecode instruction	Abstract instruction
`pop`	*Pop*
`rem`	*Execute*(`rem`)
`rem.un`	*Execute*(`rem.un`)
`ret`	*Return*
`shl`	*Execute*(`shl`)
`shr`	*Execute*(`shr`)
`shr.un`	*Execute*(`shr.un`)
`starg` *num*	*StoreArg*(*num*)
`starg.s` *num*	*StoreArg*(*num*)
`stind.i1`	*StoreInd*(`int8`)
`stind.i2`	*StoreInd*(`int16`)
`stind.i4`	*StoreInd*(`int32`)
`stind.i8`	*StoreInd*(`int64`)
`stind.r4`	*StoreInd*(`float32`)
`stind.r8`	*StoreInd*(`float64`)
`stind.i`	*StoreInd*(`native int`)
`stind.ref`	*StoreInd*(`object ref`)
`stloc` *indx*	*StoreLoc*(*indx*)
`stloc.s` *indx*	*StoreLoc*(*indx*)
`stloc.0`	*StoreLoc*(0)
`stloc.1`	*StoreLoc*(1)
`stloc.2`	*StoreLoc*(2)
`stloc.3`	*StoreLoc*(3)
`sub`	*Execute*(`sub`)
`sub.ovf`	*Execute*(`sub.ovf`)
`sub.ovf.un`	*Execute*(`sub.ovf.un`)
`switch`	not supported
`xor`	*Execute*(`xor`)

5.2 Object Model Instructions

IL Bytecode instruction	Abstract instruction
`box` *valTypeTok*	*Box*(*valTypeTok*)
`callvirt` *mref*/*rt*	*CallVirt*(`false`,*rt*,*mref*)
`tail.callvirt` *mref*/*rt*	*CallVirt*(`true`,*rt*,*mref*)
`castclass` *class*	*CastClass*(*class*)
`cpobj` *classTok*	*CopyObj*(*classTok*)
`initobj` *classTok*	*InitObj*(*classTok*)
`isinst` *class*	*IsInstance*(*class*)
`ldelem.i1`	*LoadElem*(`int8`)
`ldelem.i2`	*LoadElem*(`int16`)
`ldelem.i4`	*LoadElem*(`int32`)
`ldelem.i8`	*LoadElem*(`int64`)
`ldelem.u1`	*LoadElem*(`uint8`)
`ldelem.u2`	*LoadElem*(`uint16`)
`ldelem.u4`	*LoadElem*(`uint32`)
`ldelem.u8`	*LoadElem*(`uint64`)
`ldelem.r4`	*LoadElem*(`float32`)
`ldelem.r8`	*LoadElem*(`float64`)
`ldelem.i`	*LoadElem*(`native int`)
`ldelem.ref`	*LoadElem*(`object ref`)
`ldelema` *class*	*LoadElemA*(*class*)
`ldfld` *fref*/*t*	*LoadField*(*t*,*fref*)
`ldflda` *fref*/*t*	*LoadFieldA*(*t*,*fref*)
`ldlen`	*LoadLength*
`ldobj` *classTok*	*LoadInd*(*classTok*)
`ldsfld` *fref*/*t*	*LoadStatic*(*t*,*fref*)
`ldsflda` *fref*	*LoadStaticA*(*t*,*fref*)
`ldstr` *string*	*Const*(`string`,*string*)
`ldtoken` *token*	*LoadToken*(*token*)

IL Bytecode instruction	Abstract instruction
`ldvirtftn` *meth*	*LoadVirtFtn*(*meth*)
`mkrefany` *class*	*MkRefAny*(*class*)
`newarr` *etype*	*NewArray*(*etype*)
`newobj` *ctor*	*NewObj*(*ctor*)
`refanytype`	*RefAnyType*
`refanyval` *type*	*RefAnyVal*(*type*)
`rethrow`	*Rethrow*
`sizeof` *valueType*	*SizeOf*(*valueType*)
`stelem.i1`	*StoreElem*(`int8`)
`stelem.i2`	*StoreElem*(`int16`)
`stelem.i4`	*StoreElem*(`int32`)
`stelem.i8`	*StoreElem*(`int64`)
`stelem.r4`	*StoreElem*(`float32`)
`stelem.r8`	*StoreElem*(`float64`)
`stelem.i`	*StoreElem*(`native int`)
`stelem.ref`	*StoreElem*(`object ref`)
`stfld` *fref*/*t*	*StoreField*(*t*,*fref*)
`stobj` *classTok*	*StoreInd*(*classTok*)
`stsfld` *fref*/*t*	*StoreField*(*t*,*fref*)
`throw`	*Throw*
`unbox` *valuetype*	*Unbox*(*valuetype*)