Report

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

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
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_I ⊂ CLR_C ⊂ CLR_O ⊂ CLR_E ⊂ CLR_P ⊂ CLR_VC ⊂ CLR_TR ⊂ CLR_MP ⊂ CLR_RM. For each such submodule CLR_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 submodule, i.e. CLR_RM. We omit here bytecode instructions for monitors.

The exceptions submodule CLR_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:

\[
\begin{align*}
  f(s) := t & \quad \text{update } f \text{ at } s \text{ to } t \\
  \text{if } \varphi \text{ then } P \text{ else } Q & \quad \text{if } \varphi \text{ is true, then execute } P, \text{ else } Q \\
  P \ Q & \quad \text{execute } P \text{ and } Q \text{ in parallel} \\
  \text{let } x = t \text{ in } P & \quad \text{assign } t \text{ to } x \text{ and then execute } P \\
  P \ \text{seq} \ Q & \quad \text{execute } P \text{ and then } Q \\
  \text{forall } x \in A \ \text{do } P & \quad \text{execute in parallel } P \text{ for every } x \in A \\
  \text{choose } x \text{ with } \varphi \text{ do } P & \quad \text{choose an } x \text{ satisfying } \varphi, \text{ and execute } P \\
  P \text{ or } Q & \quad \text{execute } P \text{ or } Q
\end{align*}
\]

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_T with instructions required for the compilation of imperative programs of a while language. CLR_T is extended to CLR_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_O which supports instructions for object oriented features. The exceptions are introduced in the machine CLR_E defined in Section 2.5. This section introduces also the complex CLR exception handling mechanism. The machine CLR_P defined in Section 2.6 extends CLR_E by adding instructions for dealing with pointers. The machine CLR_VC defined in Section 2.7 pro-
Fig. 1 The CIL instructions

<table>
<thead>
<tr>
<th>CLR$_C$ instructions</th>
<th>CLR$_R$ instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Instr = \ldots$</td>
<td>$Instr = \ldots$</td>
</tr>
<tr>
<td>Execute($Op$)</td>
<td>LoadArgA($Arg$)</td>
</tr>
<tr>
<td>Const($ConstType$, $Literal$)</td>
<td>LoadLocal($Local$)</td>
</tr>
<tr>
<td>LoadLoc($Local$)</td>
<td>LoadStaticA($Type$, $FRef$)</td>
</tr>
<tr>
<td>StoreLoc($Local$)</td>
<td>LoadFieldA($Type$, $FRef$)</td>
</tr>
<tr>
<td>Branch($Pc$)</td>
<td>LoadInd($LoadIndType$)</td>
</tr>
<tr>
<td>Cond($Op$, $Pc$)</td>
<td>StoreInd($StoreIndType$)</td>
</tr>
<tr>
<td>Dup</td>
<td>InitBlock</td>
</tr>
<tr>
<td>Pop</td>
<td>CopyBlock</td>
</tr>
<tr>
<td>Halt</td>
<td>LocAlloc</td>
</tr>
<tr>
<td></td>
<td>LoadElemA($Class$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLR$_C$ instructions</th>
<th>CLR$_VC$ instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Instr = \ldots$</td>
<td>$Instr = \ldots$</td>
</tr>
<tr>
<td>LoadStatic($Type$, $FRef$)</td>
<td>InitObj($Value.Type$)</td>
</tr>
<tr>
<td>StoreStatic($Type$, $FRef$)</td>
<td>CopyObj($Value.Type$)</td>
</tr>
<tr>
<td>LoadArg($Arg$)</td>
<td>Box($Value.Type$)</td>
</tr>
<tr>
<td>StoreArg($Arg$)</td>
<td>Unbox($Value.Type$)</td>
</tr>
<tr>
<td>Call(TailCall,$Type$, $MRef$)</td>
<td>SizeOf($Value.Type$)</td>
</tr>
<tr>
<td>Return</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLR$_C$ instructions</th>
<th>CLR$_TR$ instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Instr = \ldots$</td>
<td>$Instr = \ldots$</td>
</tr>
<tr>
<td>NewObj($MRef$)</td>
<td>MkRefAny($Type$)</td>
</tr>
<tr>
<td>LoadField($Type$, $FRef$)</td>
<td>RefAnyType</td>
</tr>
<tr>
<td>StoreField($Type$, $FRef$)</td>
<td>RefAnyVal($Type$)</td>
</tr>
<tr>
<td>CallVirt(TailCall,$Type$, $MRef$)</td>
<td></td>
</tr>
<tr>
<td>CastClass($Class$)</td>
<td></td>
</tr>
<tr>
<td>IsInstance($Class$)</td>
<td></td>
</tr>
<tr>
<td>Jmp($MRef$)</td>
<td></td>
</tr>
<tr>
<td>NewArray($Type$)</td>
<td></td>
</tr>
<tr>
<td>LoadLength</td>
<td></td>
</tr>
<tr>
<td>LoadElem($ArrayElemType$)</td>
<td></td>
</tr>
<tr>
<td>StoreElem($ArrayElemType$)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLR$_C$ instructions</th>
<th>CLR$_M$ instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Instr = \ldots$</td>
<td>$Instr = \ldots$</td>
</tr>
<tr>
<td>Throw</td>
<td>LoadFtn($MRef$)</td>
</tr>
<tr>
<td>Rethrow</td>
<td>LoadVirtFtn($MRef$)</td>
</tr>
<tr>
<td>EndFilter</td>
<td>CallI(TailCall,$Sig$)</td>
</tr>
<tr>
<td>EndFinally</td>
<td></td>
</tr>
<tr>
<td>Leave($Pc$)</td>
<td></td>
</tr>
</tbody>
</table>

provides instructions for dealing with value class instances. The typed references represent the novelty introduced by the machine CLR$_TR$ in Section 2.8. The method pointers are handled by the instructions specific to CLR$_M$ which is defined in Section 2.9. Section 3 provides the topmost machine, i.e. CLR$_RM$ 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 \(N\) 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\(_T\) submachine

The CLR\(_T\) 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\(_T\) 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.

\[
\text{ConstType} = \text{int32 | int64 | float32 | float64}
\]

The dynamic state of the CLR\(_T\) consists of a frame containing a program counter *Pc*, local variable addresses *locAddr* and an evaluation stack *evalStack*. The *Pc* runs over the set of natural numbers \(Pc = N\). The *locAddr* carries the addresses of the local variables and not their values. Although the addresses are not needed for CLR\(_T\), they are later addressable with the instructions added in CLR\(_P\). 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:

\[
\begin{align*}
\text{pc} : \text{Pc} & \quad \text{pc} = 0 \\
\text{locAddr} : \text{Map(Local, Adr)} & \quad \text{locAddr}(n) \in \text{Adr}, \forall n \in \text{Local} \\
\text{evalStack} : \text{List(Val \times CLRStackType)} & \quad \text{evalStack} = [] \\
\text{mem} : \text{Map(Adr, Val \cup \{undef\})} & \quad \text{mem}(adr) = \text{undef}, \forall \text{adr} \in \text{Adr}
\end{align*}
\]

**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).

\[
\text{CLRStackType} = \text{int32} | \text{int64} | \text{native int} | F
\]

The valuesOf selects the value component of a list of evalStack slots, i.e. valuesOf applied to \([\text{val}_1, t_1], \ldots, (\text{val}_n, t_n)\] returns \([\text{val}_1, \ldots, \text{val}_n]\). 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, “.”)\(^2\), 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\'·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 CLR\_I. The machine CLR\_I fires the execCLR\_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 division/remainder 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).

\[
\text{ExceptionCase}(op, slots) \iff \text{DivByZeroCase}(op, slots) \lor \text{OverflowCase}(op, slots) \\
\lor \text{InvNrCase}(op, slots)
\]

\(^2\)The “.” denotes the append operation for lists.
The execution of CLR instructions

\[
\text{execCLR}_T(\text{instr}) \equiv \text{match instr}
\]

\[
\text{Execute}(\text{op}) \rightarrow \\
\begin{cases}
\text{let (evalStack', slots) = split(evalStack, opNo(op)) in} \\
\text{if } \neg \text{ExceptionCase(op, slots) then} \\
\text{let (val, t) = (CLRResVal(op, slots), CLRResType(op, typesOf(slots))) in} \\
\text{evalStack := evalStack' : [(val, t)]} \\
\text{pc := pc + 1}
\end{cases}
\]

\[
\text{Const}(t, \text{lit}) \rightarrow \text{evalStack := evalStack · [[lit, CLRTypeOf(t)]]} \\
\text{pc := pc + 1}
\]

\[
\text{LoadLoc}(n) \rightarrow \text{if zeroInit(meth) then} \\
\begin{cases}
\text{let } t = \text{locTypes}(n) \text{ in} \\
\text{evalStack := evalStack · [(memVal(locAdr(n), t), CLRTypeOf(t))]} \\
\text{pc := pc + 1}
\end{cases}
\]

\[
\text{StoreLoc}(n) \rightarrow \text{let (evalStack', [val]) = getVals(evalStack, 1) in} \\
\text{WRITEMEM(locAdr(n), locTypes(n), val)} \\
\text{evalStack := evalStack'} \\
\text{pc := pc + 1}
\]

\[
\text{Branch}(t) \rightarrow \text{pc := t}
\]

\[
\text{Cond}(\text{op, t}) \rightarrow \text{let (evalStack', slots) = split(evalStack, opNo(op)) in} \\
\text{evalStack := evalStack'} \\
\text{pc := if CLRResVal(op, slots) then t else pc + 1}
\]

\[
\text{Dup} \rightarrow \text{let (evalStack', [[val, t]]) = split(evalStack, 1) in} \\
\text{evalStack := evalStack' · [[val, t], [val, t]]} \\
\text{pc := pc + 1}
\]

\[
\text{Pop} \rightarrow \text{pop(evalStack)} \\
\text{pc := pc + 1}
\]

\[
\begin{align*}
\text{DivByZeroCase}(\text{op, slots}) \iff & \quad \text{op } \in \text{DivOp} \land \text{vals}(1) = 0 \\
& \quad \land \quad \text{types}(i) \in \{\text{int32, int64, native int}\}, \ i = 0, 1
\end{align*}
\]

\[
\text{OverflowCase}(\text{op, slots}) \iff \quad \text{op } \in \text{OrfOp} \\
& \quad \land \quad \text{Overflow(CLRResVal(op, slots), CLRResType(op, types))}
\]

\[
\begin{align*}
\text{InvNrCase}(\text{op, slots}) \iff & \quad (\text{op } = \text{cfinite} \land \text{vals}(0) \in \{\text{NaN, +infinity, -infinity}\}) \\
& \quad \lor \quad (\text{op } = \{\text{div, rem}\} \land \text{vals}(0) = \text{min}(\text{types}(0)) \land \text{vals}(1) = -1 \\
& \quad \land \quad \text{types}(i) \in \{\text{int32, int64, native int}\}, \ i = 0, 1)
\end{align*}
\]

The \text{Cond}(t, \text{lit}) instruction pushes the constant \text{lit} of type \text{t} on top of the \text{evalStack}. However, the type of the stack location is determined with \text{CLRTypeOf}. If the \text{zeroInit} flag of current method is set, the \text{LoadLoc} instruction loads the value of a local variable. The value of the local variable \text{n} of the declared type \text{t} is determined using the derived function \text{memVal} : \text{Map(ADR, Val)} applied to the address of \text{n} and \text{t}. The \text{memVal} builds up the value of a given type stored in memory at a certain address. In \text{CLR}_T, \text{memVal}(\text{adr, } t) := \text{mem}(\text{adr})$. The type \text{t} becomes relevant when we refine the universe \text{Val} in \text{CLR}_C. \text{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 \text{WRITEMEM} that also considers the type of the stored value although this is not needed in \text{CLR}_T.

\text{WRITEMEM}(\text{adr, } t, \text{val}) \equiv \text{mem}(\text{adr}) := \text{val}

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

CLR_C extends CLR_T 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_C only of the object classes defined by the universe ObjClass (in CLR_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 signatures”. A signature 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_O) may be prefixed by tail whose presence is indicated by a boolean flag – element of TailCall – in the abstract Call instruction.

\[
\begin{align*}
FRef & = \text{Class} \times \text{Field} \quad & \text{Class} & = \text{ObjClass} \quad & \text{TailCall} & = \text{Bool} \\
MRef & = \text{Class} \times \text{Method} \times \text{Sig} \\
\text{Arg} & = \mathbb{N}
\end{align*}
\]

The static function code : \(\text{Map}(\text{MRef}, \text{List}(\text{Instr}))\) 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_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.

\[
\begin{align*}
type & : \text{Map}(\text{FRef}, \text{Type}) \\
\text{statFields} & : \text{Map}(\text{Class}, \text{P}(\text{FRef})) \\
\text{instFields} & : \text{Map}(\text{Class}, \text{P}(\text{FRef})) \\
\text{retType} & : \text{Map}(\text{Sig}, \text{Type}) \\
\text{paramTypes} & : \text{Map}(\text{Sig}, \text{List}(\text{Type})) \\
\text{classNm} & : \text{Map}(\text{MRef}, \text{Class}) \\
\text{methNm} & : \text{Map}(\text{MRef}, \text{Meth}) \\
\text{locTypes} & : \text{Map}(\text{MRef}, \text{List}(\text{Type})) \\
\text{zeroInit} & : \text{Map}(\text{MRef}, \text{Bool})
\end{align*}
\]

In CLR_C, the procedural abstraction is added in the form of static methods. Unlike in CLR_T where we had a single method frame, in CLR_C we have a stack frameStack of call frames described by Frame.

\[
\text{Frame} = \text{Pc} \times \text{Map}(\text{Local}, \text{Adr}) \times \text{Map}(\text{Arg}, \text{Adr}) \times \mathcal{P}(\text{Adr}) \times \text{List}(\text{Val} \times \text{CLRStackType}) \times \text{MRef}
\]

A frame in CLR_C is enriched with more information than in CLR_T. The frame components are, in order, the following: a program counter pc, local variables addresses locAdr, arguments addresses argAdr : \(\text{Map}(\text{Arg}, \text{Adr})\), 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_C, they become addressable.

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

7
in CLR. 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:

\[
\text{locAdr} = \emptyset \quad \text{argAdr} = \emptyset \quad \text{StackAdr} = \emptyset \quad \text{meth} = \text{Object::entrypoint}
\]

\[
\text{frameStack} = [\]
\]

**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:

\[
\text{Call}(\text{False}, \text{void}, \text{CIL\_entrypoint})
\]

\[
\text{Halt}
\]

We denote by `frame` the currently executed frame:

\[
\text{frame} = (\text{pc}, \text{locAdr}, \text{argAdr}, \text{StackAdr}, \text{evalStack}, \text{meth})
\]

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:

\[
\text{Switch} = \text{Noswitch} | \text{Invoke(TailCall, MRef, List(Val))} | \text{InitClass(Class)}
\]

\[
\text{Result(List(Val × CLRStackType))}
\]

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 CLR. 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 CLR). 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 CLR) of that class, or the first invocation of any static, instance or virtual method (see CLR) 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 beforefieldinit : Map(Class, 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:

\[
\begin{align*}
\text{ClassState} &= \text{Linked} | \text{Initialized} \\
\text{classState} : = \text{Map(Class, ClassState)} \\
\text{classState}(c) &= \text{Linked}, \forall c \in \text{Class} \setminus \{\text{Object, } <\text{Module}>\} \\
\text{classState}(\text{Object}) &= \text{classState}(<\text{Module}>) = \text{Initialized}
\end{align*}
\]

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 initialized(c) holds for a class c if classState(c) = 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:

\[
\text{reqinit(c)} \iff \neg\text{initialized(c)} \land \neg\text{beforefieldinit(c)}
\]

Rules The ASM rules for CLR_C are defined in Fig. 3. The machine CLR_C executes the macro execScheme_C which is parameterized by the machines execCLR_C and switchCLR. The macro 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 switchCLR. Otherwise, as a consequence of the beforefieldinit semantics, either a beforefieldinit class is initialized through INITIALIZECLASS or execCLR 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. INITIALIZECLASS arbitrarily chooses a beforefieldinit class that is not yet initialized. It then passes the control to the switchCLR to initialize the class.

\[\text{INITIALIZECLASS} \equiv \text{choose } c \in \text{Class with } \neg\text{initialized}(c) \land \text{beforefieldinit}(c) \text{ do}
\]

\[\text{switch} := \text{InitClass}(c)\]

The following explanations are for the execCLR 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 LoadStatic(t, c::f) is the declared type of the field reference c::f. The same explanation applies for StoreStatic(t, c::f). In case of Call(\_, t, c::m), t denotes the return type of the considered method reference. LoadStatic(t, c::f) pushes on the evalStack the value of the field f stored at the address 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: argTypes : Map(MRef, List(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:

\[\text{argTypes}(c::m) = \text{paramTypes}(c::m)\]

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 LoadArg(n). The instruction StoreArg(n) writes the value of the topmost evalStack slot into the address of argument n.

\(^4\)The execScheme_C is redefined to execScheme_E in CLR_E while introducing exceptions. If an exception is thrown, the execScheme_E passes the control to the exception handling mechanism. Thus, it prevents the execution of the execCLR machines.

\(^5\)This definition is refined in CLR_C and CLR_VC.
The switch machine

The rules of the submachine switchCLR are presented in Fig. 4. The rule Invoke(tail, c::m, 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.
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`.

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 × MRef, Val ∪ { 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.

---

6The macro `DeAllocMem` is defined in the paragraph on memory management.
SetFrame(c::m, args) ≡

\[ pc := 0 \]
\[ evalStack := [] \]
\[ meth := c::m \]

MakeArgLoc(argTypes(c::m), locTypes(c::m), args · zero(locTypes(c::m), c::m))

\[ zero(t, c::m) = \text{if} \ zeroInit(c::m) \text{ then} \text{defVal}(t) \text{ else} \text{undef} \]

**Remark** The tail calls (in particular Jmp in CLR_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.

InitStatFields(c) ≡ \forall f \in \text{statFields}(c) \ do
\[ \text{WriteMem}(\text{globals}(f), \text{type}(f), \text{defVal}(\text{type}(f))) \]

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 : \mathcal{M}(\text{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 : \mathcal{M}(\text{Adr}, \text{Byte} \cup \{\text{undef}\}) and the definitions of memVal and WriteMem are refined as follows:

\[ \text{memVal}(adr, t) = [\text{mem}(adr + i) | i \in [0..\text{sizeOf}(t) - 1]] \]

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

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:

\[ \text{length(encode(val))} = \text{sizeOf}(t) \text{ and } \text{decode}(t, \text{encode}(val)) = val \]

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 \texttt{validAdr : Map(Adr, 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 length(sizes):

\[
\text{SpaceFor(sizes)} = \{ \text{adr}_i^{n-1} \in \text{Adr}^n \mid \text{validAdr}(\text{adr}_i) \ \forall i = 0, n - 1 \text{ and } \\
\bigcup_{i=0}^{n-1} [\text{adr}_i, \text{adr}_i + \text{sizes}(i)] \subseteq \text{Adr} \setminus (\text{StackAdr} \cup \text{HeapAdr}) \}
\]

If an attempt to allocate on the stack fails, a \texttt{StackOverflowException} is thrown (see CLR\textsubscript{C}). Similarly, if one needs to allocate on the heap, then an \texttt{OutOfMemoryException} is raised.

The following explanations are for the macro \texttt{MAKEARGLOC}, which we used when setting up the frame for a method call. \texttt{MAKEARGLOC} is applied to three lists: two lists of types \texttt{types\textsubscript{1}} and \texttt{types\textsubscript{2}} and a list of values \texttt{vals}. It assumes that the sum of the first two lists’ lengths is equal with the length of \texttt{vals}. The macro resets first the maps \texttt{locAdr} and \texttt{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 \texttt{types\textsubscript{1}} and \texttt{types\textsubscript{2}} and writes the values \texttt{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 \texttt{StackAdr}. The \texttt{switch} is set to \texttt{Noswitch} only if there is sufficient place to be allocated on the stack\footnote{To avoid a conflict in the update of \texttt{switch}, we set it in the \texttt{MAKEARGLOC}. Note that, in case there is not sufficient space for the allocation, then the \texttt{switch} will point to the exception handling mechanism – see the refinement of \texttt{MAKEARGLOC} in CLR\textsubscript{E}.}.

\texttt{MAKEARGLOC(types\textsubscript{1}, types\textsubscript{2}, vals) \equiv}

\[
\begin{align*}
\text{locAdr} & := \emptyset \\
\text{argAdr} & := \emptyset \\
\text{seq} & := \text{choose} \ (\text{adr})^{n+m-1} \in \text{SpaceFor(sizes)} \\
\text{forall } i = 0, m + n - 1 & \text{ do} \\
\text{if } i < m & \text{ then } \text{argAdr}(i) := \text{adr}_i \\
\text{else } \text{locAdr}(i - m) & := \text{adr}_i \\
\text{WRITEMEM} & (\text{adr}_i, \text{types}(i), \text{vals}(i)) \\
\text{StackAdr} & := \text{StackAdr} \cup \bigcup_{i=0}^{n-1} [\text{adr}_i, \text{adr}_i + \text{sizes}(i)] \\
\text{where } \text{types} & = \text{types\textsubscript{1}} \cdot \text{types\textsubscript{2}} \text{ and } \text{sizes} = \text{sizeOf}^\ast (\text{types}) \\
\text{and } m & = \text{length(}\text{types\textsubscript{1}}\text{)} \text{ and } n = \text{length(}\text{types\textsubscript{2}}\text{)}
\end{align*}
\]

The macro \texttt{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\textsubscript{P}).

\texttt{DEALLOCMEM(A) \equiv}

\[
\begin{align*}
\text{StackAdr} & := A \\
\text{forall } \text{adr in StackAdr \setminus A } & \text{do mem(adr)} := \text{undef}
\end{align*}
\]

2.4 The CLR\textsubscript{O} submodule

CLR\textsubscript{O} extends CLR\textsubscript{C} by object-oriented features like objects creation and initialization, instance fields and methods (including instance constructors) and type casts. CLR\textsubscript{O} includes also an optimization of the tail calls described in CLR\textsubscript{C}.\footnote{The function \texttt{sizeOf\textsuperscript{\ast}} is refined in CLR\textsubscript{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 $CLRTypOf$ maps every object type to the special type $O$.

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

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

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

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

$ArrayElemType = SignedInt \mid UnsignedInt \mid FloatType \mid native \mid ref$

Beside static fields, in CLR 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.

$fieldOffset : \text{Map(Class} \times \text{FRef}, \mathbb{N})$

$fieldAdr : \text{Map(ObjRef} \times \text{FRef}, \text{Adr})$

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 $elemType$.

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

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

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

$elemType : \text{Map(ObjRef}, \text{Type})$

$elemType(r) = t$, where $\text{actualTypeOf}(r) = Array(t)$

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 $CallVirt$.

$lookup : \text{Map(Type} \times \text{MRef}, \text{MRef})$ 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 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 $\text{actualTypeOf} : \text{Map}(\text{ObjRef}, \text{Type})$ records the actual type of an object on the heap.

Due to the object references, $\text{mem}$ is refined as $\text{mem} : \text{Map(Adr}, \text{Byte} \cup \text{ObjRef} \cup \{\text{undef}\})$ and the $\text{memVal}$ and $\text{WRITEMEM}$ are refined as follows:

$\text{memVal}(adr, t) = \begin{cases} \text{if } t \in \text{ObjType} \text{ then } \text{mem}(adr) \\ \text{else } \text{mem}(adr + i) \mid i \in [0..\text{sizeOf}(t) - 1] \end{cases}$

$\text{WRITEMEM}(adr, t, val) \equiv \begin{cases} \text{if } t \in \text{ObjType} \text{ then } \text{mem}(adr) := val \\ \text{else forall } i \in [0..\text{sizeOf}(t) - 1] \text{ do } \text{mem}(adr + i) := val(i) \end{cases}$
The function \( sizeOf^* \) which was used in the definition of \( \text{MAKEARGLOC} \) in Section 2.3 is refined to take into account object references. Thus, \( sizeOf^*(t) = sizeOf(\text{native int}) \) if \( t \in \text{ObjType} \), and \( sizeOf^*(t) = sizeOf(t) \) otherwise.

**Rules** Fig. 5, 6 and 8 define the rules for \( \text{CLR}_O \). The rule \( \text{NewObj} \) is applicable only for object classes\(^{11}\) and is defined through the macro \( \text{CREATEANDINITOBJ} \).

\[
\text{CREATEANDINITOBJ}(c::\text{ctor}) \equiv \\
\text{if } \neg \text{reqinit}(c) \text{ then} \\
\quad \text{if } \text{SpaceFor}([sizeOf(c)]) \neq \emptyset \text{ then} \\
\quad \quad \text{let } (\text{evalStack}', \text{vals}) = \text{getVals(}\text{evalStack}, \text{paramNo}(c::\text{ctor})) \text{ in} \\
\quad \quad \quad \text{let } r = \text{new}(\text{ObjRef}, c) \text{ in} \\
\quad \quad \quad \text{evalStack} := \text{evalStack}' \cdot [(r, O)] \\
\quad \quad \quad \text{actualTypeOf}(r) := c \\
\quad \quad \quad \text{forall } f \in \text{instFields}(c) \text{ do} \\
\quad \quad \quad \quad \text{WRITEMEM(}\text{fieldAddr}(r, f), \text{type}(c::f), \text{defVal}(\text{type}(f))) \\
\quad \quad \quad \text{switch} := \text{Invoke}(\text{False, c::ctor, } [r] \cdot \text{vals}) \\
\quad \quad \text{else } \text{switch} := \text{InitClass}(c)
\]

\(^{11}\)In Section 2.7 it is defined a \( \text{NewObj} \) rule for value classes.
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:

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

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 CLRVC, 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 CLRVC). 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 CLR₀ instructions (continued)

ARRAYINSTR₀ ≡

\[
\text{NewArray}(t) \rightarrow \text{let } (evalStack', [n]) = \text{getVals}(evalStack, 1) \text{ in}
\]

if \( n \geq 0 \) then

if \( \text{SpaceFor}([n \times \text{sizeOf}(t)]) \neq \emptyset \) then

let \( r = \text{new}(\text{ObjRef}, t, n) \) in

actualTypeOf(\( r \)) := Array(\( t \))

arraySize(\( r \)) := \( n \)

evalStack := evalStack' · [(r, O)]

\[ \text{pc} := \text{pc} + 1 \]

\[
\text{LoadLength} \rightarrow \text{let } (evalStack', [r]) = \text{getVals}(evalStack, 1) \text{ in}
\]

if \( r \neq \text{null} \) then

\[ \text{evalStack} := \text{evalStack'} \cdot [(\text{arraySize}(r), \text{native int})] \]

\[ \text{pc} := \text{pc} + 1 \]

\[
\text{LoadElem}(t) \rightarrow \text{let } (evalStack', [r, i]) = \text{getVals}(evalStack, 2) \text{ in}
\]

if \( r \neq \text{null} \land (i \geq 0 \land i < \text{arraySize}(r)) \) then

\[ \text{evalStack} := \text{evalStack'} \cdot [(\text{memVal}(\text{elemAdr}(r, i), \text{elemType}(r)), \text{CLRTypetOf}(t))] \]

\[ \text{pc} := \text{pc} + 1 \]

\[
\text{StoreElem}(t) \rightarrow \text{let } (evalStack', [r, i, val]) = \text{getVals}(evalStack, 3) \text{ in}
\]

if \( r \neq \text{null} \land (i \geq 0 \land i < \text{arraySize}(r)) \)

\( \land (t \neq \text{ref} \lor \text{val} = \text{null} \lor \text{actualTypeOf}(\text{val}) \preceq \text{elemType}(r)) \) then

\[ \text{WriteMem}(\text{elemAdr}(r, i), \text{elemType}(r), \text{val}) \]

\[ \text{evalStack} := \text{evalStack'} \]

\[ \text{pc} := \text{pc} + 1 \]

Remark LoadField, StoreField and CallVirt (but also LoadFieldA in CLRₚ) 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 CLRₚ). This is another difference wrt JVM [6].

The Call rule that we define in CLR₀ can be fired only for instance methods (for statics the Call rule from CLR₂ 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 CLR₀ as follows: for every instance method reference \( c::m \) where \( c \) is an object class:

\[
\text{argTypes}(c::m) = [e] \cdot \text{paramTypes}(c::m)
\]

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 CLR₂). InstanceOf 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 InstanceOF 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:
The elements addresses are recorded by \textit{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.

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

The length of an array can be obtained using the \texttt{LoadLength} instruction. The \texttt{LoadElem} loads on the stack an array element given by the index and array reference on the \texttt{evalStack}. If one needs to write the value of an array element, then we use the \texttt{StoreElem} instruction. The store operation succeeds if the array reference is not \texttt{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 \texttt{Jmp} is not verifiable, we still model its semantics (see Fig. 8). If the method is declared by a class which requires initialization, \texttt{Jmp} initializes the class first. \texttt{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 \texttt{Jmp} is fired. The memory allocated on the stack for the current frame is reclaimed through \texttt{DeAllocMem}.

\section{The CLR$_E$ submachine}

CLR$_E$ extends CLR$_O$ by instructions dealing with exception handling. CLR$_E$ 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, \texttt{Unwind} and \texttt{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 transferred 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.

\[
\text{Switch} = \ldots | \text{ExcMech}
\]

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 occurred 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 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:

\[\text{Exc} = \ldots \]

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.
The predicates $isInTry$, $isInHandler$ and $isInFilter$

\[
\begin{align*}
  isInTry(pos, h) & \iff \text{tryStart}(h) \leq pos < \text{tryStart}(h) + \text{tryLength}(h) \\
  isInHandler(pos, h) & \iff \text{handlerStart}(h) \leq pos < \text{handlerStart}(h) + \text{handlerLength}(h) \\
  isInFilter(pos, h) & \iff \text{filterStart}(h) \leq pos < \text{handlerStart}(h)
\end{align*}
\]

Any 7-tuple of the above form describes a handler of kind $clauseKind$ which “protects” the region\(^{13}\) that starts at $\text{tryStart}$ and has the length $\text{tryLength}$, handles the exception in an area of instructions that starts at $\text{handlerStart}$ and has the length $\text{handlerLength}$ – 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 $\text{filterStart}$. In case of a $filter$ handler, the handler region starting at $\text{handlerStart}$ immediately follows the $filter$ region – consequently we have $\text{filterStart} < \text{handlerStart}$. We often refer to the sequence of instructions between $\text{filterStart}$ and $\text{handlerStart} - 1$ 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 $pos \in Pc$ and a handler $h \in Exc$. Note that if the predicate $isInFilter$ is true, then $\text{filterStart}$ 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.
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.

\[
\text{ExcRec} = \text{ExcRec}(exc: \text{ObjRef}, pass: \{\text{StackWalk, Unwind}\}, stackCursor: \text{Frame} \times \mathbb{N}, handler: \text{Frame} \times \mathbb{N})
\]

\[
\text{LeaveRec} = \text{LeaveRec}(pass: \{\text{Leave}\}, stackCursor: \text{Frame} \times \mathbb{N}, target: \text{Pc})
\]

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.
In the initial state of excCLR, there is no pass to be executed, i.e. \( \text{pass} = \text{undef} \).

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 \( \text{stackCursor} \), then excCLR will try to find (when \( \text{StackWalking} \)) or to execute (when \( \text{Unwinding} \)) or to leave (when \( \text{Leave} \)) the corresponding handler; otherwise it will continue its work in the invoker frame or end its \( \text{Leave} \) pass at the \( \text{target} \).

### 2.5.3 StackWalk pass

During a \( \text{StackWalk} \) pass, excCLR starts in the current \( \text{frame} \) 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 \( \text{excHA} \) of the corresponding method \( m \).

\[
\text{existsHanWithinFrame}((\_, \_, \_, \_, \_, \_, m), n) \iff n < \text{length}(\text{excHA}(m))
\]

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

\[
\text{SearchInvFrame}(f) \equiv \\
\text{let } \_ \cdot [f', f] \cdot \_ = \text{frameStack} \cdot [\text{frame}] \text{ in} \\
\text{RESET}(\text{stackCursor}, f')
\]

There are three groups of possible handlers \( h \) excCLR is looking for in a given frame during its \( \text{StackWalk} \):

- a **catch** handler whose \( \text{try} \) block protects the program counter \( \text{pc} \) of the frame pointed at by \( \text{stackCursor} \) and whose \( \text{type} \) is a supertype of the exception type;

\[
\text{matchCatch}(\text{pos}, \text{t}, h) \iff \text{isInTry}(\text{pos}, h) \land \text{clauseKind}(h) = \text{catch} \land \text{t} \preceq \text{type}(h)
\]

- a **filter** handler whose \( \text{try} \) block protects the \( \text{pc} \) of the frame pointed at by \( \text{stackCursor} \);

\[
\text{matchFilter}(\text{pos}, h) \iff \text{isInTry}(\text{pos}, h) \land \text{clauseKind}(h) = \text{filter}
\]

- a **filter** handler whose \( \text{filter} \) region contains \( \text{pc} \) of the frame pointed at by \( \text{stackCursor} \).

This corresponds to an outer exception and will be described in more detail below.

The order of the \( \text{if} \) clauses in the \( \text{let} \) statement from the rule \( \text{StackWalk} \) is not important. This is justified by the following property:

**Disjointness 2** For every type \( t \), the predicates \( \text{matchCatch}^t \), \( \text{matchFilter} \) and \( \text{isInFilter} \) are pairwise disjoint\(^{14}\).

\(^{14}\)By \( \text{matchCatch}^t \) we understand the predicate defined by the set \( \{ (\text{pos}, h) \mid \text{matchCatch}(\text{pos}, t, h) \} \).
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 \texttt{stackCursor}, namely \texttt{hanWithinFrame}((\ldots, m), n) = \texttt{excHA}(m)(n), is not of any of the above types, the \texttt{stackCursor} is incremented to point to the next handler in the \texttt{excHA}:

$$
\text{GoToNxtHan} \equiv \texttt{stackCursor} := (f, n + 1) \\
\textbf{where} \quad \texttt{stackCursor} = (f, n)
$$

The Ordering assumption stated in Section 2.5.2 and the lexical nesting constraints stated in \cite[Partition I, §12.4.2.7]{ECMA-262} ensure that, if the \texttt{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 \texttt{stackCursor}) of any of the above types.

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

$$
\text{FoundHandler} \equiv \\
\text{pass} := \texttt{Unwind} \\
\text{handler} := \texttt{stackCursor} \\
\text{Reset}(s, f) \equiv s := (f, 0)
$$

If the handler is a \texttt{filter}, then by means of \texttt{ExecFilter} its \texttt{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 \cite{ECMA-262}. The currently to be executed frame becomes the frame for executing the \texttt{filter} region. The faulting exception frame is pushed on the \texttt{frameStack}. The current frame points now to the method, local variables and arguments of the frame in which \texttt{stackCursor} is, it has the exception reference on the evaluation stack \texttt{evalStack} and the program counter \texttt{pc} set to the beginning \texttt{filterStart} of the \texttt{filter} region. The \texttt{switch} is set to \textit{Noswitch} in order to pass the control to the normal machine \texttt{execCLR_E}.

$$
\text{ExecFilter}(h) \equiv \\
\texttt{pc} := \texttt{filterStart}(h) \\
\texttt{evalStack} := [\texttt{exc}] \\
\texttt{locAdr} := \texttt{locAdr'} \\
\texttt{argAdr} := \texttt{argAdr'} \\
\texttt{meth} := \texttt{meth'} \\
\text{push(frameStack, frame)} \\
\texttt{switch} := \textit{Noswitch} \\
\textbf{where} \quad \texttt{stackCursor} = ((\ldots, \texttt{locAdr'}, \texttt{argAdr'}, \ldots, \texttt{meth'}), \ldots)
$$

Exceptions in \texttt{filter} region? It is not documented in the ECMA standard what happens if an (inner) exception is thrown while executing the \texttt{filter} region during the \texttt{StackWalk} pass of an outer exception. The following cases are to be considered:
Fig. 10 The exception handling machine excCLR

\[
\text{excCLR} \equiv \\
\text{match pass} \\
\quad \text{StackWalk} \rightarrow \text{if existsHanWithinFrame(stackCursor) then} \\
\quad \quad \text{let } h = \text{hanWithinFrame(stackCursor) in} \\
\quad \quad \quad \text{if matchCatch(pos, actualTypeOf(exc), h) then} \\
\quad \quad \quad \quad \text{FoundHandler} \\
\quad \quad \quad \quad \text{Reset(stackCursor, frame)} \\
\quad \quad \quad \text{elseif matchFilter(pos, h) then ExecFilter(h)} \\
\quad \quad \quad \text{elseif isInFilter(pos, h) then ExitInnerExc} \\
\quad \quad \text{else GoToNxtHan} \\
\quad \text{else SearchInvFrame(f)} \\
\quad \text{where stackCursor} = (f, -) \text{ and } f = (\text{pos, \ldots, \ldots})
\]

\[
\text{Unwind} \rightarrow \text{if existsHanWithinFrame(stackCursor) then} \\
\quad \text{let } h = \text{hanWithinFrame(stackCursor) in} \\
\quad \quad \text{if matchTargetHan(handler, stackCursor) then} \\
\quad \quad \quad \text{ExecHan(h)} \\
\quad \quad \text{elseif matchFinFault(pc, h) then} \\
\quad \quad \quad \text{ExecHan(h)} \\
\quad \quad \quad \text{GoToNxtHan} \\
\quad \text{elseif isInHandler(pc, h) then} \\
\quad \quad \quad \text{AbortPrevPassRec} \\
\quad \quad \text{GoToNxtHan} \\
\quad \text{elseif isInFilter(pc, h) then} \\
\quad \quad \quad \text{ContinueOuterExc} \\
\quad \text{else GoToNxtHan} \\
\quad \text{else if meth = c::cctor then classState(c) := Exc(exc)} \\
\quad \quad \text{PopFrame(0, [])} \\
\quad \text{SearchInvFrame(frame)}
\]

\[
\text{Leave} \rightarrow \text{if existsHanWithinFrame(stackCursor) then} \\
\quad \text{let } h = \text{hanWithinFrame(stackCursor) in} \\
\quad \quad \text{if isFinFromTo(h, pc, target) then ExecHan(h)} \\
\quad \quad \text{if isRealHanFromTo(h, pc, target) then AbortPrevPassRec} \\
\quad \text{GoToNxtHan} \\
\quad \text{else} \\
\quad \quad \text{pc := target} \\
\quad \text{PopRec} \\
\quad \text{switch := Noswitch}
\]

- 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 occurred 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]).

\[
\text{ExitInnerExc} \equiv \\
\text{pass} := \text{Unwind} \\
\text{Reset}(\text{stackCursor}, \text{frame})
\]

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 Unwinding, 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.

\[
\text{matchTargetHan}((f1,n1),(f2,n2)) \Leftrightarrow \text{sameFrame}(f1,f2) \land n1 = n2
\]

\[
\text{sameFrame}(f1,f2) \Leftrightarrow \text{pr}_i(f1) = \text{pr}_i(f2), \forall i \in \{2,3,6\}
\]

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

\[
\text{matchFinFault}(pos,h) \Leftrightarrow \text{isInTry}(pos,h) \land \text{clauseKind}(h) \in \{\text{finally, fault}\}
\]

- 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.
\[
\text{EXEC\textsc{Han}(h) \equiv} \\
\quad pc := \text{handlerStart}(h) \\
\quad \text{evalStack :=} \\
\quad \quad \text{if clauseKind}(h) \in \{\text{catch, filter}\} \text{ then} \\
\quad \quad \quad [\text{exc}] \\
\quad \quad \text{else} \\
\quad \quad \quad [] \\
\quad \text{switch := Noswitch}
\]

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

Let us assume that the handler pointed to by \texttt{stackCursor} is an arbitrary handler whose handler region contains \texttt{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 \texttt{filter} code:

- if the exception is taken care of in the handler region, i.e. it is successfully handled by a \texttt{catch/filter} handler or it is discarded (because it occurred in a \texttt{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 \texttt{excCLR} is aborted through \texttt{AbortPrevPassRec};

\[
\text{AbortPrevPassRec} \equiv \text{pop(excRecStack)}
\]

- the exception is propagated further, i.e. the \texttt{Unwind} pass continues via \texttt{GoToNxtHan} (see Fig. 10) which sets the \texttt{stackCursor} to the next handler in \texttt{excHA}.

The execution of a handler region can occur only when \texttt{excCLR} runs in the \texttt{Unwind} and \texttt{Leave} passes: in \texttt{Unwind} handler regions of any kind are executed while in \texttt{Leave} only \texttt{finally} handler regions are executed. If the raised exception occurred while \texttt{excCLR} runs an \texttt{Unwind} pass for handling an outer exception, the \texttt{Unwind} pass of the outer exception is stopped and the corresponding pass record is popped from \texttt{excRecStack} (see Tests 1, 3 and 4 in [3]). If the exception has been thrown while \texttt{excCLR} runs a \texttt{Leave} pass for executing \texttt{finally} handlers on the way from a \texttt{Leave} instruction to its target, then this pass is stopped and its associated pass record is popped off \texttt{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 \texttt{stackCursor} is a \texttt{filter} handler whose \texttt{filter} region contains \texttt{pc}, then the current (inner) exception is aborted and the \texttt{filter} considered as not providing a handler for the outer exception. So there is no way to exit a \texttt{filter} region with
an exception. This ensures that the frame built by EXECFILTER for executing a \texttt{filter} region is used only for this purpose. The handling of the outer exception is continued through \texttt{CONTINUEOUTEREXC} (see Fig. 10) which pops the frame built for executing the \texttt{filter} region, pops from the \texttt{excRecStack} the pass record corresponding to the inner exception and reestablishes the pass context of the outer exception, but with the \texttt{stackCursor} pointing to the handler following the just inspected \texttt{filter} handler. The updates of the \texttt{stackCursor} in \texttt{PopRec} and \texttt{GotoNxtHan} are done sequentially such that the update in \texttt{GotoNxtHan} overwrites the update in \texttt{PopRec}.

\begin{verbatim}
CONTINUEOUTEREXC ≡
  PopFrame(0, [])
  PopRec seq GotoNxtHan
\end{verbatim}

\begin{verbatim}
PopRec ≡
  if excRecStack = [] then
    SetRecUndef
    switch := Noswitch
  else let (excRecStack', [r]) = split(excRecStack, 1) in
    if r ∈ ExcRec then
      let (exc', pass', stackCursor', handler') = r in
      exc := exc'
      pass := pass'
      stackCursor := stackCursor'
      handler := handler'
    if r ∈ LeaveRec then
      let (pass', stackCursor', target') = r in
      pass := pass'
      stackCursor := stackCursor'
      target := target'
    execRecStack := execRecStack'

SetRecUndef ≡
  exc := undef
  pass := undef
  stackCursor := undef
  target := undef
  handler := undef
\end{verbatim}

If the handler pointed to by the \texttt{stackCursor} is not of any of the above types, the \texttt{stackCursor} is incremented to point to the next handler in the \texttt{exxHA}.

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

**Exceptions in class initializers?** If an exception occurs in a type initializer \texttt{.cctor}, then the type shall be marked as being in an erroneous state and a \texttt{TypeInitializationException} is thrown. This means that an exception can and will escape the body of an initializer only by the specific exception \texttt{TypeInitializationException}. This is not specified by the ECMA
The meaning is that, if a class is in a state $\text{Exc}(r)$, then the class initializer has been exited with a $\text{TypeInitializationException}$ reference $r$.

The $\text{InitClass}$ 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 $\text{LoadRec}$ 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 $\text{Exc}(\_)$ state shall throw the same initially thrown exception reference, we need to refine the predicate $\text{reqinit}$ as follows:

$$\text{reqinit}(c) \Leftrightarrow (\neg \text{initialized}(c) \lor \neg \text{beforefieldinit}(c)) \lor \text{classState}(c) = \text{Exc}(\_).$$

When a field is accessed, the predicate $\text{initialized}$ is checked. If the corresponding class is in an $\text{Exc}(\_)$ state, then the predicate $\text{initialized}$ is False. Consequently, the switch is set to $\text{InitClass}$ 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).

---

15In the real CLR implementation, the exception thrown in .cctor is embedded as an inner exception in the $\text{TypeInitializationException}$. We do not model this aspect here.

---

Fig. 11 The switchCLR machine (continued)

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

\[
\text{isFinFromTo}(h, \text{pos}_1, \text{pos}_2) \iff \\
\text{isInTry}(\text{pos}_1, h) \land \text{clauseKind}(h) = \text{finally} \land \neg\text{isInTry}(\text{pos}_2, h) \land \neg\text{isInHandler}(\text{pos}_2, h) \\
\text{isRealHanFromTo}(h, \text{pos}_1, \text{pos}_2) \iff \\
\text{clauseKind}(h) \in \{\text{catch, filter}\} \land \text{isInHandler}(\text{pos}_1, h) \land \neg\text{isInHandler}(\text{pos}_2, h)
\]

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

### 2.5.6 The rules of CLR$E$

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

The rules of \text{execCLR}_E in Fig. 12 specify the effect of the CIL instructions related to exceptions. Each of these rules transfers the control to \text{execCLR}. The \text{Throw} instruction is defined through the macro \text{Throw}:

\[
\text{throw}(r) \equiv \\
\quad \text{if } r \neq \text{null} \text{ then} \\
\quad \quad \text{LoadRec}((r, \text{StackWalk}, (\text{frame}, 0), \text{undef})) \\
\quad \text{else} \; \text{Raise}(	ext{NullReferenceException})
\]

The topmost element of the \text{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 \text{execCLR}: the \text{stackCursor} is initialized by the current \text{frame} and 0. If the exception mechanism is already working in a pass, i.e. \text{pass} \neq \text{undef}, then the current pass record is pushed on the \text{excRecStack}.

\[
\text{LoadRec}(r) \equiv \\
\quad \text{if } r \in \text{ExcPass} \text{ then} \\
\quad \quad \text{let } (\text{exc}', \text{pass}', \text{stackCursor}', \text{handler}') = r \text{ in} \\
\quad \quad \quad \text{exc} := \text{exc}' \\
\quad \quad \quad \text{pass} := \text{pass}' \\
\quad \quad \quad \text{stackCursor} := \text{stackCursor}' \\
\quad \quad \quad \text{handler} := \text{handler}' \\
\quad \text{else let } (\text{pass}', \text{stackCursor}', \text{target}') = r \text{ in} \\
\quad \quad \quad \text{pass} := \text{pass}' \\
\quad \quad \quad \text{stackCursor} := \text{stackCursor}' \\
\quad \quad \quad \text{target} := \text{target}' \\
\quad \text{if } \text{pass} \neq \text{undef} \text{ then} \; \text{PushRec} \\
\quad \quad \text{switch} := \text{ExcMech}
\]

29
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\(^{\text{16}}\);
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\(^{\text{17}}\);

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), excis 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

\(^{\text{16}}\)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 .ctor initializers are empty modulo a Return instruction.

\(^{\text{17}}\)The same size assumption seems reasonable since the exception classes have the same instance fields inherited from the SystemException class.
filter are “exception_continue_search”(0) and “exception_execute_handler”(1). In other words, resumable exceptions are not (yet) supported contradicting Partition I.

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

\( \text{EndFinally} \) terminates the execution of the \( \text{handler} \) region of a \( \text{finally/fault} \) \( \text{handler} \). It empties the \( \text{evalStack} \) and transfers the control to \( \text{excCLR} \). A \( \text{Leave} \) instruction empties the \( \text{evalStack} \) and loads on \( \text{excCLR} \) a pass record corresponding to a \( \text{Leave} \) pass.

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

In addition, \( \text{CLR}_E \) 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 \( \text{excCLR} \).
Fig. 13 The execution of CLR\textsubscript{E} instructions (continued)

\[
\text{ExcS\textsuperscript{5}ensitiveInstr}\textsubscript{E} \equiv \\
\text{Execute}(op) \rightarrow \text{let slots} = \text{take(evalStack, opNo(op)) in} \\
\quad \text{if DivByZeroCase(op, slots) then} \\
\quad \quad \text{RAISE(DivideByZeroException)} \\
\quad \text{elseif OverflowCase(op, slots) then} \\
\quad \quad \text{RAISE(OverflowException)} \\
\quad \text{elseif InvalidNrCase(op, slots) then} \\
\quad \quad \text{RAISE(InvalidCastException)} \\
\text{CastClass(c)} \rightarrow \text{let } (r, \_\_) = \text{top(evalStack) in} \\
\quad \text{if } r \neq \text{null} \land \text{actualTypeOf(r) } \not\leq c \text{ then} \\
\quad \quad \text{RAISE(InvalidCastException)} \\
\text{LoadLen}o\text{c}(\_\_) \rightarrow \text{if } \neg \text{ZeroInit(meth) then} \\
\quad \text{halt := "Verification Exception"} \\
\text{Call}(\_\_, c::m) \rightarrow \text{if instance(c::m) } \land \neg \text{reqinit(c) then} \\
\quad \text{let } (\_\_, [r] \_\_) = \text{getVals(evalStack, argNo(c::m)) in} \\
\quad \quad \text{if } r = \text{null then RAISE(NullReferenceException)} \\
\text{StoreField}(\_\_, c::f) \rightarrow \text{if instance(c::f) } \land \neg \text{reqinit(c) then} \\
\quad \text{let } (\_\_, [r] \_\_) = \text{getVals(evalStack, 2) in} \\
\quad \quad \text{if } r = \text{null then RAISE(NullReferenceException)} \\
\text{CallVirt}(\_\_, c::m) \rightarrow \text{let } (\_\_, [r] \_\_) = \text{getVals(evalStack, argNo(c::m)) in} \\
\quad \text{let } ::m = \text{lookup(actualTypeOf(r), c::m)} in \\
\quad \quad \text{if } \neg \text{reqinit(d) then} \\
\quad \quad \quad \text{if } r = \text{null then RAISE(NullReferenceException)} \\
\text{ArrayInstr}\textsubscript{E} \equiv \\
\text{NewArray}(t) \rightarrow \text{let } (n, \_) = \text{top(evalStack) in} \\
\quad \text{if } n < 0 \text{ then RAISE(OverflowException)} \\
\quad \text{elseif SpaceFor([n * sizeOf(t)]) } = \emptyset \text{ then} \\
\quad \quad \text{RAISE(OutOfMemoryException)} \\
\quad \text{else halt := "not enough space to be allocated"} \\
\text{LoadLength} \rightarrow \text{let } (r, \_\_) = \text{top(evalStack) in} \\
\quad \text{if } r = \text{null then RAISE(NullReferenceException)} \\
\text{LoadElem}(\_\_) \rightarrow \text{let } (\_\_, [r, i]) = \text{getVals(evalStack, 2) in} \\
\quad \text{if } r = \text{null then RAISE(NullReferenceException)} \\
\quad \quad \text{elseif } i < 0 \lor i \geq \text{arraySize(r)} \text{ then} \\
\quad \quad \quad \text{RAISE(IndexOutOfRangeException)} \\
\text{StoreElem}(t) \rightarrow \text{let } (\_\_, [r, i], val) = \text{getVals(evalStack, 3) in} \\
\quad \text{if } r = \text{null then RAISE(NullReferenceException)} \\
\quad \quad \text{elseif } i < 0 \lor i \geq \text{arraySize(r)} \text{ then} \\
\quad \quad \quad \text{RAISE(IndexOutOfRangeException)} \\
\quad \text{elseif } t = \text{ref } \land \text{val } \neq \text{null } \land \text{actualTypeOf(val) } \not\leq \text{elemType(r)} \text{ then} \\
\quad \quad \quad \text{RAISE(ArrayTypeMismatchException)}
\]

is the definition of \text{Execute}. The extension of this rule raises a \text{DivideByZeroException}, if \text{op} is a division operator and the second topmost value on the stack is 0 (see the definition of the predicate \text{DivByZeroCase} in Section 2.2). An \text{OverflowException} is thrown if \text{op} is an operator that performs an overflow check (e.g. \text{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 \text{InvalidNrCase} in Section 2.2, if \text{op} is the \text{ckfinite} operator and the topmost value on the stack is one of NaN, +\text{infinity} or −\text{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 CLR\( _O \):

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

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 submodule 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 CLR\( _P \) submodule

CLR\( _P \) extends CLR\( _O \) with pointer types, i.e. types whose values are memory addresses. CLR\( _P \) 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)

\[
\text{switchCLR} \equiv \text{match switch}
\]

\[
\text{Invoke}(\text{tail}, \text{c}::m, \text{args}) \rightarrow \ldots
\]

seq
if SpaceFor([sizes]) \neq \emptyset then
\ldots
else Raise(\text{StackOverflowException})
\ldots

InitClass(c) \rightarrow if classState(c) = \text{Linked} then
if SpaceFor([sizes]) \neq \emptyset then
\ldots
else Raise(\text{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 \( \text{CLRStackType} \) includes now also the special type &
corresponding to managed pointers. \( \text{LoadIndType} \) describes the type of a value indirectly loaded
from the memory on \( \text{evalStack} \) with \( \text{LoadInd} \), while \( \text{StoreIndType} \) describes the type of a value
indirectly stored into the memory with \( \text{StoreInd} \).

\[
\text{CLRStackType} = \ldots | \& \quad \text{SignedInt} = \text{int8} | \text{int16} | \text{int32} | \text{int64}
\]

\[
\text{UnsignedInt} = \text{uint8} | \text{uint16} | \text{uint32} | \text{uint64}
\]

\[
\text{LoadIndType} = \text{SignedInt} | \text{UnsignedInt} | \text{Float} | \text{native int} | \text{ref}
\]

\[
\text{StoreIndType} = \text{SignedInt} | \text{Float} | \text{native int} | \text{ref}
\]

In Section 2.3 we have introduced the external function \( \text{validAdr} \) to check addresses for
validity. An address is invalid if it is \text{null} or is not in the range of \( \text{Adr} \) 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 \text{native int} type. Note that, as a consequence
of the definition of \( \text{SpaceFor} \), the results of all CIL instructions that return addresses (e.g.
\( \text{LoadLocA} \) and \( \text{LoadArgA} \)) are valid.

**Rules** The rules for \( \text{CLR}P \) are defined in Fig. 15 and 17. Assuming that the \text{zeroInit}
flag of the current method is set, \( \text{LoadLocA}(n) \) pushes the address of the local variable indexed with \( n \)
on the \( \text{evalStack} \). If the \text{zeroInit} is not set, then the machine halts. Similarly, \( \text{LoadArgA}(n) \) pushes
on the \( \text{evalStack} \) 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 \( \text{LoadStaticA} \). If the class which declares the static field is not yet initialized, the execution
proceeds first with the class initialization. \( \text{LoadFieldA} \) is similar with \( \text{LoadStaticA} \) and can be
used for both static and instance fields. If the field is an instance field and the object reference on
\( \text{evalStack} \) is \text{null}, a \text{NullReferenceException} is thrown. Also if the index is not in the array range, a
\text{IndexOutOfRangeException} is raised. An \text{ArrayTypeMismatchException} is thrown if the type embedded given in the
\( \text{LoadElemA} \) does not match the element type of the array.

**Call by-reference mechanism** Unlike JVM, the CLR allows to pass to a method arguments
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$_P$ 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,
The value classes support the compilation of the C# `struct` (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$_{	ext{VC}}$ submodule

CLR$_{	ext{VC}}$ extends CLR$_{	ext{P}}$ 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$_{	ext{VC}}$ 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 = . . . | ValueClass LoadIndType = . . . | ValueClass Class = . . . | ValueClass StoreIndType = . . . | ValueClass

The fieldAdr is refined to be applicable also to pointers referring to value class instances: fieldAdr : Map(ObjRef ∪ Adr) × 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.

\[
\text{memVal}(adr, t) = \begin{cases} 
\text{mem}(adr) & \text{if } t \in \text{ObjType} \\
\text{memVal(fieldAdr}(adr, f), \text{type}(f)) & \text{if } f \in \text{instFields}(t) \\
\text{mem}(adr + i) & \text{if } i \in [0..\text{sizeOf}(t) - 1] 
\end{cases}
\]

\[
\text{WRITEMEM}(adr, t, val) = \begin{cases} 
\text{mem}(adr) := val & \text{if } t \in \text{ObjType} \\
\text{WRITEMEM}(fieldAdr(adr, f), \text{type}(f), val(f)) & \text{if } f \in \text{instFields}(t) \\
\text{mem}(adr + i) := val(i) & \text{if } i \in [0..\text{sizeOf}(t) - 1]
\end{cases}
\]

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.

\[
\text{let } adr = \text{stackalloc}(vc) \text{ in } P \equiv
\begin{cases} 
\text{choose } (adr) \in \text{SpaceFor}([n]) & \text{do} \\
\text{StackAdr} := \text{StackAdr} \cup [adr, adr + n] \\
\text{ALLOCFIELDS}(adr, vc) \\
\text{seq } P \\
\text{where } n = \text{sizeOf}(vc)
\end{cases}
\]

\[
\text{ALLOCFIELDS}(adr, t) = \begin{cases} 
\text{if } t \in \text{ValueClass} & \text{do} \\
\text{forall } f \in \text{instFields}(t) & \text{do} \\
\text{WRITEMEM}(fieldAdr(adr, f), \text{type}(f), val(f)) \\
\text{let } a = adr + \text{fieldOffSet}(t, f) & \text{in} \\
\text{fieldAdr}(adr, f) := a \\
\text{ALLOCFIELDS}(a, \text{type}(f))
\end{cases}
\]

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

\[
\text{argTypes}(vc::m) = [vc&] \cdot \text{paramTypes}(vc::m)
\]

**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

\[ \text{CLR}_{VC} \equiv \text{execScheme}_{E} (\text{execCLR}_{VC}, \text{switchCLR}, \text{excCLR}) \]

\[ \text{execCLR}_{VC}(\text{instr}) \equiv \text{execCLR}_{P}(\text{instr}) \]

\[ \begin{align*}
\text{match } & \text{ instr} \\
\text{NewObj}(vc::.ctor) & \rightarrow \text{ if } vc \in \text{ValueClass} \text{ then } \\
& \quad \text{ if } \neg \text{reqinit}(vc) \text{ then } \\
& \qquad \text{ if } \text{SpaceFor}([\text{sizeof}(vc)]) \neq \emptyset \text{ then } \\
& \qquad \quad \text{ let } (\text{evalStack}', \text{vals}) = \text{getVals}($evalStack$, \text{paramNo}(vc::.ctor)) \text{ in } \\
& \qquad \quad \text{ let } adr = \text{stackalloc}(vc) \text{ in } \\
& \qquad \quad \quad \text{ evalStack} := \text{evalStack}' \cdot [(\text{memVal}(adr, vc), vc)] \\
& \qquad \quad \quad \text{forall } f \in \text{instFields}(vc) \text{ do } \\
& \qquad \quad \quad \quad \text{ WRITEMEM(fieldAddr(adr, f), type(f), defVal(type(f))) } \\
& \qquad \quad \quad \text{ switch } := \text{Invoke(False, vc::.ctor, [adr] \cdot vals)} \\
& \quad \quad \text{ else } \text{RAISE(OutOfMemoryException)} \\
& \quad \text{ else } \text{switch} := \text{InitClass}(vc) \\
\text{InitObj}(vt) & \rightarrow \text{ let } (\text{evalStack}', [adr]) = \text{getVals}(\text{evalStack}, 1) \text{ in } \\
& \quad \text{ WRITEMEM(adr, vt, defVal(vt)) } \\
& \quad \text{ evalStack} := \text{evalStack}' \\
& \quad pc := pc + 1 \\
\text{CopyObj}(vt) & \rightarrow \text{ let } (\text{evalStack}', [\text{dest}_adr, \text{src}_adr]) = \text{getVals}(\text{evalStack}, 2) \text{ in } \\
& \quad \text{ if } \text{validAddr(\text{dest}_adr)} \land \text{validAddr(\text{src}_adr)} \text{ then } \\
& \quad \quad \text{ WRITEMEM(\text{src}_adr, vt, memVal(\text{dest}_adr, vt)) } \\
& \quad \quad \text{ evalStack} := \text{evalStack}' \\
& \quad pc := pc + 1 \\
& \quad \text{ else } \text{RAISE(NullReferenceException)} \\
\text{Box}(vt) & \rightarrow \text{ if } \text{SpaceFor}([\text{sizeof}(vt)]) \neq \emptyset \text{ then } \\
& \quad \text{ let } (\text{evalStack}', [val]) = \text{getVals}(\text{evalStack}, 1) \text{ in } \\
& \quad \text{ let } r = \text{new(ObjRef)} \text{ and } \text{adr} = \text{heapalloc}(vt) \text{ in } \\
& \quad \text{ actualTypeOf(r)} := vt \\
& \quad \text{ addressOf(r)} := \text{adr} \\
& \quad \text{ WRITEMEM(adr, vt, val) } \\
& \quad \text{ evalStack} := \text{evalStack}' \cdot [(r, O)] \\
& \quad pc := pc + 1 \\
& \quad \text{ else } \text{RAISE(OutOfMemoryException)} \\
\text{Unbox}(vt) & \rightarrow \text{ let } (\text{evalStack}', [r]) = \text{getVals}(\text{evalStack}, 1) \text{ in } \\
& \quad \text{ if } \text{actualTypeOf(r)} = \text{vt} \text{ then } \\
& \quad \quad \text{ evalStack} := \text{evalStack}' \cdot [(\text{addressOf}(r), &) ] \\
& \quad pc := pc + 1 \\
& \quad \text{ else } \text{RAISE(InvalidCastException)} \\
\text{SizeOf}(vt) & \rightarrow \text{ evalStack} := \text{evalStack} \cdot [\text{sizeof}(vt), \text{int32}] \\
& \quad pc := pc + 1
\end{align*} \]

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

The instruction \texttt{InitObj} initializes an instance of a value type. It takes the value of the topmost \texttt{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 \texttt{WRITEMEM}. The \texttt{CopyObj} instruction copies an instance of a value type. It takes two pointers from the \texttt{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 ≡
import r do
  ObjRef(r) := True
seq P
```

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\(_T_R\) submachine

CLR\(_T_R\) extends CLR\(_T_VC\) 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 ≡
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 CLR\textsubscript{T R} instructions

\[
\text{CLR}_{T R} \equiv \text{execScheme}_E(\text{execCLR}_{T R}, \text{switchCLR}, \text{excCLR})
\]

\[
\begin{align*}
\text{execCLR}_{T R}(\text{instr}) & \equiv \\
\text{execCLR}_{V C}(\text{instr})
\end{align*}
\]

\[
\begin{aligned}
\text{match } \text{instr} & \\
\text{MKRefAny}(t) & \rightarrow \text{let } \text{tr} = \text{new}(\text{TypedRef}) \text{ in} \\
& \quad \text{let } (\text{evalStack}', [\text{adr}]) = \text{getVals(}\text{evalStack}, 1) \text{ in} \\
& \quad \text{typedRefAdr(}\text{tr}) := \text{adr} \\
& \quad \text{typedRefType}(\text{tr}) := t \\
& \quad \text{evalStack} := \text{evalStack}' \cdot [(\text{tr}, \text{TypedReference})] \\
& \quad \text{pc} := \text{pc} + 1
\end{aligned}
\]

\[
\begin{aligned}
\text{RefAnyType} & \rightarrow \text{let } (\text{evalStack}', [\text{tr}]) = \text{getVals(}\text{evalStack}, 1) \text{ in} \\
& \quad \text{let } t = \text{typedRefType(}\text{tr}) \text{ in} \\
& \quad \text{evalStack} := \text{evalStack}' \cdot [(\text{handleOf(}\text{t}), \text{RuntimeTypeHandle})] \\
& \quad \text{pc} := \text{pc} + 1
\end{aligned}
\]

\[
\begin{aligned}
\text{RefAnyVal} & \rightarrow \text{let } (\text{evalStack}', [\text{tr}]) = \text{getVals(}\text{evalStack}, 1) \text{ in} \\
& \quad \text{if } t \neq \text{typedRefType(}\text{tr}) \text{ then } \text{Raise(}\text{InvalidCastException}) \\
& \quad \text{else} \\
& \quad \quad \text{evalStack} := \text{evalStack}' \cdot [(\text{typedRefAdr(}\text{tr}), \&)] \\
& \quad \quad \text{pc} := \text{pc} + 1
\end{aligned}
\]

of the \text{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 \text{evalStack}.

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

2.9 The CLR\textsubscript{MP} submodule

\text{CLR}_{\text{MP}} extends \text{CLR}_{T R} by method pointers. The CLR instructions handling method pointers support the compilation of the C++ method pointers and C\# delegates.

\textbf{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 \text{FunctionPtr} that point to the corresponding function references. For a method pointer, the function \text{methodOf} : \text{Map(}\text{FunctionPtr}, \text{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 \text{MulticastDelegate} which in turn is derived from \text{Delegate}. A delegate instance is then simply an instance of this object class and is created with the \text{NewObj} instruction introduced in CLR\textsubscript{O}. However, there are some constraints on the definition of such a “delegate class”: it should have a method \text{Invoke} with appropriate parameters, and each instance of a delegate forwards calls to its \text{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 \text{target object} and the \text{target method} and are
Fig. 20 The execution of CLR\(_{MP}\) instructions

\[
\text{CLR}_{MP} \equiv \text{execScheme_E}(\text{EXECCLR}_{MP}, \text{SWITCHCLR}, \text{EXECCLR})
\]

\[
\text{EXECCLR}_{MP}(\text{instr}) \equiv \\
\text{EXECCLR}_{TN}(\text{instr})
\]

\[
\text{match} \text{ instr} \\
\text{LoadFtn}(c::m) \rightarrow \text{ let } fp = \text{new}(\text{FunctionPtr}, c::m) \text{ in } \\
\text{ evalStack} := \text{evalStack} \cdot [(fp, \text{native int})] \\
\text{ pc } := \text{pc} + 1
\]

\[
\text{LoadVirtFtn}(c::m) \rightarrow \text{ let } (\text{evalStack}', [r]) = \text{getVals}(\text{evalStack}, 1) \text{ in } \\
\text{ if } r \neq \text{null} \text{ then } \\
\text{ let } fp = \text{new}(\text{FunctionPtr}, \text{lookup}(\text{actualTypeOf}(r), c::m)) \text{ in } \\
\text{ evalStack} := \text{evalStack}' \cdot [(fp, \text{native int})] \\
\text{ pc } := \text{pc} + 1 \\
\text{ else } \text{Raise}(\text{NullReferenceException})
\]

\[
\text{NONVERIFIABLE}_{MP}
\]

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\(_{MP}\) 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:

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

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$_{MP}$ instructions

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

the \textit{this} argument. The \textit{CallI} instruction can have a \texttt{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 \texttt{NullReferenceException} is thrown if \textit{CallI} is invoked with a \texttt{null} object and a pointer to an instance method. Our experiments showed that the \texttt{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 \texttt{this} pointer is non-\texttt{null} while in the valid code, the \texttt{this} pointer can also be \texttt{null} (see the \textit{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 \texttt{NewObj} instruction). However, the two semantics are equivalent for the purposes of the model in [7].

3 The CLR$_{RM}$ submodule

The submodule CLR$_{RM}$ extends CLR$_{MP}$ 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 \texttt{Invoke} method and the constructor \texttt{.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$_{RM}$.

Rules To extend the submachine CLR$_{MP}$ with runtime managed methods, we refine the \texttt{execScheme$_E$} defined in CLR$_E$ as showed in Fig. 22. The instance constructor \texttt{.ctor} of a delegate class shall take exactly two parameters. The first parameter is supposed to be of type \texttt{Object} and the second parameters of type \texttt{IntPtr}. 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 \_\texttt{target} and \_\texttt{methodPtr} are set to the target object and target method pointer, respectively. If the target object is \texttt{null}, then the exception \texttt{NullReferenceException} is thrown.

The method \texttt{Invoke} defined by a delegate class shall have the same return type and parameter types as the target method. The target object can be \texttt{null} only in case the target method is \texttt{static}. The reason is that, before the creation of the delegate instance, the \texttt{.ctor} constructor checks the target object against \texttt{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 \_\texttt{target} is not \texttt{null}. If the class which declares the target method requires initialization, then it is initialized.

\footnote{20}{The library type \texttt{System.IntPtr} corresponds to the CLR \texttt{native int} type.}
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


## 5 Appendix

### 5.1 The Basic Instructions

<table>
<thead>
<tr>
<th>IL Bytecode instruction</th>
<th>Abstract instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td><code>Execute(add)</code></td>
</tr>
<tr>
<td><code>add.ovf</code></td>
<td><code>Execute(add.ovf)</code></td>
</tr>
<tr>
<td><code>add.ovf.un</code></td>
<td><code>Execute(add.ovf.un)</code></td>
</tr>
<tr>
<td><code>and</code></td>
<td><code>Execute(and)</code></td>
</tr>
<tr>
<td><code>arglist</code></td>
<td><code>ArgList</code></td>
</tr>
<tr>
<td><code>beq t</code></td>
<td><code>Cond(eq,t)</code></td>
</tr>
<tr>
<td><code>beq.s t</code></td>
<td><code>Cond(eq,s,t)</code></td>
</tr>
<tr>
<td><code>bge t</code></td>
<td><code>Cond(ge,t)</code></td>
</tr>
<tr>
<td><code>bge.s t</code></td>
<td><code>Cond(ge,s,t)</code></td>
</tr>
<tr>
<td><code>bge.un t</code></td>
<td><code>Cond(ge.un,t)</code></td>
</tr>
<tr>
<td><code>bge.un.s t</code></td>
<td><code>Cond(ge.un,s,t)</code></td>
</tr>
<tr>
<td><code>bgt t</code></td>
<td><code>Cond(gt,t)</code></td>
</tr>
<tr>
<td><code>bgt.s t</code></td>
<td><code>Cond(gt,s,t)</code></td>
</tr>
<tr>
<td><code>bgt.un t</code></td>
<td><code>Cond(gt.un,t)</code></td>
</tr>
<tr>
<td><code>bgt.un.s t</code></td>
<td><code>Cond(gt.un,s,t)</code></td>
</tr>
<tr>
<td><code>ble t</code></td>
<td><code>Cond(le,t)</code></td>
</tr>
<tr>
<td><code>ble.s t</code></td>
<td><code>Cond(le,s,t)</code></td>
</tr>
<tr>
<td><code>ble.un t</code></td>
<td><code>Cond(le.un,t)</code></td>
</tr>
<tr>
<td><code>ble.un.s t</code></td>
<td><code>Cond(le.un,s,t)</code></td>
</tr>
<tr>
<td><code>blt t</code></td>
<td><code>Cond(lt,t)</code></td>
</tr>
<tr>
<td><code>blt.s t</code></td>
<td><code>Cond(lt,s,t)</code></td>
</tr>
<tr>
<td><code>blt.un t</code></td>
<td><code>Cond(lt.un,t)</code></td>
</tr>
<tr>
<td><code>blt.un.s t</code></td>
<td><code>Cond(lt.un,s,t)</code></td>
</tr>
<tr>
<td><code>bne.un t</code></td>
<td><code>Cond(ne.un,t)</code></td>
</tr>
<tr>
<td><code>bne.un.s t</code></td>
<td><code>Cond(ne.un,s,t)</code></td>
</tr>
<tr>
<td><code>br t</code></td>
<td><code>Branch(t)</code></td>
</tr>
<tr>
<td>IL Bytecode instruction</td>
<td>Abstract instruction</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>br.s t</td>
<td>Branch(t)</td>
</tr>
<tr>
<td>break</td>
<td>not described</td>
</tr>
<tr>
<td>brfalse t</td>
<td>Cond(ifzero, t)</td>
</tr>
<tr>
<td>brfalse.s t</td>
<td>Cond(ifzero, t)</td>
</tr>
<tr>
<td>brnull t</td>
<td>Cond(ifnull, t)</td>
</tr>
<tr>
<td>brnull.s t</td>
<td>Cond(ifnull, t)</td>
</tr>
<tr>
<td>brzero t</td>
<td>Cond(ifzero, t)</td>
</tr>
<tr>
<td>brzero.s t</td>
<td>Cond(ifzero, t)</td>
</tr>
<tr>
<td>brtrue t</td>
<td>Cond(ifnonzero, t)</td>
</tr>
<tr>
<td>brtrue.s t</td>
<td>Cond(ifnonzero, t)</td>
</tr>
<tr>
<td>brinst t</td>
<td>Cond(ifnonnull, t)</td>
</tr>
<tr>
<td>brinst.s t</td>
<td>Cond(ifnonnull, t)</td>
</tr>
<tr>
<td>call mref/rt</td>
<td>Call(false, rt, mref)</td>
</tr>
<tr>
<td>tail.call mref/rt</td>
<td>Call(true, rt, mref)</td>
</tr>
<tr>
<td>calli sig</td>
<td>CallI(false, sig)</td>
</tr>
<tr>
<td>tail.calli sig</td>
<td>CallI(true, sig)</td>
</tr>
<tr>
<td>ceq</td>
<td>Execute(ceq)</td>
</tr>
<tr>
<td>cgt</td>
<td>Execute(cgt)</td>
</tr>
<tr>
<td>ckfinite</td>
<td>Execute(ckfinite)</td>
</tr>
<tr>
<td>cgt.un</td>
<td>Execute(cgt.un)</td>
</tr>
<tr>
<td>clt</td>
<td>Execute(clt)</td>
</tr>
<tr>
<td>clt.un</td>
<td>Execute(clt.un)</td>
</tr>
<tr>
<td>conv.i1</td>
<td>Execute(conv.i1)</td>
</tr>
<tr>
<td>conv.i2</td>
<td>Execute(conv.i2)</td>
</tr>
<tr>
<td>conv.i4</td>
<td>Execute(conv.i4)</td>
</tr>
<tr>
<td>conv.i8</td>
<td>Execute(conv.i8)</td>
</tr>
<tr>
<td>conv.r4</td>
<td>Execute(conv.r4)</td>
</tr>
<tr>
<td>conv.r8</td>
<td>Execute(conv.r8)</td>
</tr>
<tr>
<td>conv.u1</td>
<td>Execute(conv.u1)</td>
</tr>
<tr>
<td>conv.u2</td>
<td>Execute(conv.u2)</td>
</tr>
<tr>
<td>IL Bytecode instruction</td>
<td>Abstract instruction</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>conv.u4</td>
<td>Execute(conv.u4)</td>
</tr>
<tr>
<td>conv.u8</td>
<td>Execute(conv.u8)</td>
</tr>
<tr>
<td>conv.i</td>
<td>Execute(conv.i)</td>
</tr>
<tr>
<td>conv.u</td>
<td>Execute(conv.u)</td>
</tr>
<tr>
<td>conv.r.un</td>
<td>Execute(conv.r.un)</td>
</tr>
<tr>
<td>conv.ovf.i1</td>
<td>Execute(conv.ovf.i1)</td>
</tr>
<tr>
<td>conv.ovf.i2</td>
<td>Execute(conv.ovf.i2)</td>
</tr>
<tr>
<td>conv.ovf.i4</td>
<td>Execute(conv.ovf.i4)</td>
</tr>
<tr>
<td>conv.ovf.i8</td>
<td>Execute(conv.ovf.i8)</td>
</tr>
<tr>
<td>conv.ovf.u1</td>
<td>Execute(conv.ovf.u1)</td>
</tr>
<tr>
<td>conv.ovf.u2</td>
<td>Execute(conv.ovf.u2)</td>
</tr>
<tr>
<td>conv.ovf.u4</td>
<td>Execute(conv.ovf.u4)</td>
</tr>
<tr>
<td>conv.ovf.u8</td>
<td>Execute(conv.ovf.u8)</td>
</tr>
<tr>
<td>conv.ovf.i</td>
<td>Execute(conv.ovf.i)</td>
</tr>
<tr>
<td>conv.ovf.u</td>
<td>Execute(conv.ovf.u)</td>
</tr>
<tr>
<td>conv.ovf.i1.un</td>
<td>Execute(conv.ovf.i1.un)</td>
</tr>
<tr>
<td>conv.ovf.i2.un</td>
<td>Execute(conv.ovf.i2.un)</td>
</tr>
<tr>
<td>conv.ovf.i4.un</td>
<td>Execute(conv.ovf.i4.un)</td>
</tr>
<tr>
<td>conv.ovf.i8.un</td>
<td>Execute(conv.ovf.i8.un)</td>
</tr>
<tr>
<td>conv.ovf.u1.un</td>
<td>Execute(conv.ovf.u1.un)</td>
</tr>
<tr>
<td>conv.ovf.u2.un</td>
<td>Execute(conv.ovf.u2.un)</td>
</tr>
<tr>
<td>conv.ovf.u4.un</td>
<td>Execute(conv.ovf.u4.un)</td>
</tr>
<tr>
<td>conv.ovf.u8.un</td>
<td>Execute(conv.ovf.u8.un)</td>
</tr>
<tr>
<td>conv.ovf.i.un</td>
<td>Execute(conv.ovf.i.un)</td>
</tr>
<tr>
<td>conv.ovf.u.un</td>
<td>Execute(conv.ovf.u.un)</td>
</tr>
<tr>
<td>cpblk</td>
<td>CopyBlock</td>
</tr>
<tr>
<td>div</td>
<td>Execute(div)</td>
</tr>
<tr>
<td>div.un</td>
<td>Execute(div.un)</td>
</tr>
<tr>
<td>dup</td>
<td>Dup</td>
</tr>
<tr>
<td>endfilter</td>
<td>EndFilter</td>
</tr>
<tr>
<td>IL Bytecode instruction</td>
<td>Abstract instruction</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>endfault</td>
<td>EndFinally</td>
</tr>
<tr>
<td>endfinally</td>
<td>EndFinally</td>
</tr>
<tr>
<td>initblk</td>
<td>InitBlock</td>
</tr>
<tr>
<td>jmp meth</td>
<td>Jmp(meth)</td>
</tr>
<tr>
<td>ldarg num</td>
<td>LoadArg(num)</td>
</tr>
<tr>
<td>ldarg.s num</td>
<td>LoadArg(num)</td>
</tr>
<tr>
<td>ldarg.0</td>
<td>LoadArg(0)</td>
</tr>
<tr>
<td>ldarg.1</td>
<td>LoadArg(1)</td>
</tr>
<tr>
<td>ldarg.2</td>
<td>LoadArg(2)</td>
</tr>
<tr>
<td>ldarg.3</td>
<td>LoadArg(3)</td>
</tr>
<tr>
<td>ldarga argNum</td>
<td>LoadArgA(argNum)</td>
</tr>
<tr>
<td>ldarga.s argNum</td>
<td>LoadArgA(argNum)</td>
</tr>
<tr>
<td>ldc.i4 num</td>
<td>Const(int32,num)</td>
</tr>
<tr>
<td>ldc.i4.s num</td>
<td>Const(int32,num)</td>
</tr>
<tr>
<td>ldc.i4.m1</td>
<td>Const(int32,-1)</td>
</tr>
<tr>
<td>ldc.i4.m1</td>
<td>Const(int32,-1)</td>
</tr>
<tr>
<td>ldc.i4.0</td>
<td>Const(int32,0)</td>
</tr>
<tr>
<td>ldc.i4.1</td>
<td>Const(int32,1)</td>
</tr>
<tr>
<td>ldc.i4.2</td>
<td>Const(int32,2)</td>
</tr>
<tr>
<td>ldc.i4.3</td>
<td>Const(int32,3)</td>
</tr>
<tr>
<td>ldc.i4.4</td>
<td>Const(int32,4)</td>
</tr>
<tr>
<td>ldc.i4.5</td>
<td>Const(int32,5)</td>
</tr>
<tr>
<td>ldc.i4.6</td>
<td>Const(int32,6)</td>
</tr>
<tr>
<td>ldc.i4.7</td>
<td>Const(int32,7)</td>
</tr>
<tr>
<td>ldc.i4.8</td>
<td>Const(int32,8)</td>
</tr>
<tr>
<td>ldftn mref</td>
<td>LoadFtn(mref)</td>
</tr>
<tr>
<td>ldind.i1</td>
<td>LoadInd(int8)</td>
</tr>
<tr>
<td>IL Bytecode instruction</td>
<td>Abstract instruction</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>ldind.i2</td>
<td>LoadInd(int16)</td>
</tr>
<tr>
<td>ldind.i4</td>
<td>LoadInd(int32)</td>
</tr>
<tr>
<td>ldind.i8</td>
<td>LoadInd(int64)</td>
</tr>
<tr>
<td>ldind.u1</td>
<td>LoadInd(uint8)</td>
</tr>
<tr>
<td>ldind.u2</td>
<td>LoadInd(uint16)</td>
</tr>
<tr>
<td>ldind.u4</td>
<td>LoadInd(uint32)</td>
</tr>
<tr>
<td>ldind.r4</td>
<td>LoadInd(float32)</td>
</tr>
<tr>
<td>ldind.u8</td>
<td>LoadInd(uint64)</td>
</tr>
<tr>
<td>ldind.r8</td>
<td>LoadInd(float64)</td>
</tr>
<tr>
<td>ldind.i</td>
<td>LoadInd(native int)</td>
</tr>
<tr>
<td>ldind.ref</td>
<td>LoadInd(object ref)</td>
</tr>
<tr>
<td>ldloc indx</td>
<td>LoadLoc indx</td>
</tr>
<tr>
<td>ldloc.s indx</td>
<td>LoadLoc indx</td>
</tr>
<tr>
<td>ldloc.0</td>
<td>LoadLoc 0</td>
</tr>
<tr>
<td>ldloc.1</td>
<td>LoadLoc 1</td>
</tr>
<tr>
<td>ldloc.2</td>
<td>LoadLoc 2</td>
</tr>
<tr>
<td>ldloc.3</td>
<td>LoadLoc 3</td>
</tr>
<tr>
<td>ldloca indx</td>
<td>LoadLocA indx</td>
</tr>
<tr>
<td>ldloca.s indx</td>
<td>LoadLocA indx</td>
</tr>
<tr>
<td>ldnull</td>
<td>Const(Null,null)</td>
</tr>
<tr>
<td>leave t</td>
<td>Leave(t)</td>
</tr>
<tr>
<td>leave.s t</td>
<td>Leave(t)</td>
</tr>
<tr>
<td>localloc</td>
<td>LocAlloc</td>
</tr>
<tr>
<td>mul</td>
<td>Execute(mul)</td>
</tr>
<tr>
<td>mul.ovf</td>
<td>Execute(mul.ovf)</td>
</tr>
<tr>
<td>mul.ovf.un</td>
<td>Execute(mul.ovf.un)</td>
</tr>
<tr>
<td>neg</td>
<td>Execute(neg)</td>
</tr>
<tr>
<td>nop</td>
<td>not described</td>
</tr>
<tr>
<td>not</td>
<td>Execute(not)</td>
</tr>
<tr>
<td>or</td>
<td>Execute(or)</td>
</tr>
<tr>
<td>IL Bytecode instruction</td>
<td>Abstract instruction</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>pop</td>
<td>Pop</td>
</tr>
<tr>
<td>rem</td>
<td>Execute(rem)</td>
</tr>
<tr>
<td>rem.un</td>
<td>Execute(rem.un)</td>
</tr>
<tr>
<td>ret</td>
<td>Return</td>
</tr>
<tr>
<td>shl</td>
<td>Execute(shl)</td>
</tr>
<tr>
<td>shr</td>
<td>Execute(shr)</td>
</tr>
<tr>
<td>shr.un</td>
<td>Execute(shr.un)</td>
</tr>
<tr>
<td>starg num</td>
<td>StoreArg(num)</td>
</tr>
<tr>
<td>starg.s num</td>
<td>StoreArg(num)</td>
</tr>
<tr>
<td>stind.i1</td>
<td>StoreInd(int8)</td>
</tr>
<tr>
<td>stind.i2</td>
<td>StoreInd(int16)</td>
</tr>
<tr>
<td>stind.i4</td>
<td>StoreInd(int32)</td>
</tr>
<tr>
<td>stind.i8</td>
<td>StoreInd(int64)</td>
</tr>
<tr>
<td>stind.r4</td>
<td>StoreInd(float32)</td>
</tr>
<tr>
<td>stind.r8</td>
<td>StoreInd(float64)</td>
</tr>
<tr>
<td>stind.i</td>
<td>StoreInd(native int)</td>
</tr>
<tr>
<td>stind.ref</td>
<td>StoreInd(object ref)</td>
</tr>
<tr>
<td>stloc indx</td>
<td>StoreLoc(indx)</td>
</tr>
<tr>
<td>stloc.s indx</td>
<td>StoreLoc(indx)</td>
</tr>
<tr>
<td>stloc.0</td>
<td>StoreLoc(0)</td>
</tr>
<tr>
<td>stloc.1</td>
<td>StoreLoc(1)</td>
</tr>
<tr>
<td>stloc.2</td>
<td>StoreLoc(2)</td>
</tr>
<tr>
<td>stloc.3</td>
<td>StoreLoc(3)</td>
</tr>
<tr>
<td>sub</td>
<td>Execute(sub)</td>
</tr>
<tr>
<td>sub.ovf</td>
<td>Execute(sub.ovf)</td>
</tr>
<tr>
<td>sub.ovf.un</td>
<td>Execute(sub.ovf.un)</td>
</tr>
<tr>
<td>switch</td>
<td>not supported</td>
</tr>
<tr>
<td>xor</td>
<td>Execute(xor)</td>
</tr>
</tbody>
</table>
### 5.2 Object Model Instructions

<table>
<thead>
<tr>
<th>IL Bytecode instruction</th>
<th>Abstract instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>box</strong> valTypeTok</td>
<td>Box(valTypeTok)</td>
</tr>
<tr>
<td><strong>callvirt</strong> mref/rt</td>
<td>CallVirt(false, rt, mref)</td>
</tr>
<tr>
<td><strong>tail.callvirt</strong> mref/rt</td>
<td>CallVirt(true, rt, mref)</td>
</tr>
<tr>
<td><strong>castclass</strong> class</td>
<td>CastClass(class)</td>
</tr>
<tr>
<td><strong>cpobj</strong> classTok</td>
<td>CopyObj(classTok)</td>
</tr>
<tr>
<td><strong>initobj</strong> classTok</td>
<td>InitObj(classTok)</td>
</tr>
<tr>
<td><strong>isinst</strong> class</td>
<td>IsInstance(class)</td>
</tr>
<tr>
<td>ldelem.i1</td>
<td>LoadElem(int8)</td>
</tr>
<tr>
<td>ldelem.i2</td>
<td>LoadElem(int16)</td>
</tr>
<tr>
<td>ldelem.i4</td>
<td>LoadElem(int32)</td>
</tr>
<tr>
<td>ldelem.i8</td>
<td>LoadElem(int64)</td>
</tr>
<tr>
<td>ldelem.u1</td>
<td>LoadElem(uint8)</td>
</tr>
<tr>
<td>ldelem.u2</td>
<td>LoadElem(uint16)</td>
</tr>
<tr>
<td>ldelem.u4</td>
<td>LoadElem(uint32)</td>
</tr>
<tr>
<td>ldelem.u8</td>
<td>LoadElem(uint64)</td>
</tr>
<tr>
<td>ldelem.r4</td>
<td>LoadElem(float32)</td>
</tr>
<tr>
<td>ldelem.r8</td>
<td>LoadElem(float64)</td>
</tr>
<tr>
<td>ldelem.i</td>
<td>LoadElem(native int)</td>
</tr>
<tr>
<td>ldelem.ref</td>
<td>LoadElem(object ref)</td>
</tr>
<tr>
<td>ldelema class</td>
<td>LoadElemA(class)</td>
</tr>
<tr>
<td>ldfld fref/t</td>
<td>LoadField(t, fref)</td>
</tr>
<tr>
<td>ldflda fref/t</td>
<td>LoadFieldA(t, fref)</td>
</tr>
<tr>
<td>ldldelem</td>
<td>LoadLength</td>
</tr>
<tr>
<td>ldoobj classTok</td>
<td>LoadInd(classTok)</td>
</tr>
<tr>
<td>ldsfld fref/t</td>
<td>LoadStatic(t, fref)</td>
</tr>
<tr>
<td>ldsflda fref</td>
<td>LoadStaticA(t, fref)</td>
</tr>
<tr>
<td>ldststr string</td>
<td>Const(string, string)</td>
</tr>
<tr>
<td>ldtoken token</td>
<td>LoadToken(token)</td>
</tr>
<tr>
<td>IL Bytecode instruction</td>
<td>Abstract instruction</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>ldvirtftn meth</td>
<td>LoadVirtFtn(meth)</td>
</tr>
<tr>
<td>mkrefany class</td>
<td>MkRefAny(class)</td>
</tr>
<tr>
<td>newarr etype</td>
<td>NewArray(etype)</td>
</tr>
<tr>
<td>newobj ctor</td>
<td>NewObj(ctor)</td>
</tr>
<tr>
<td>refanytype</td>
<td>RefAnyType</td>
</tr>
<tr>
<td>refanyval type</td>
<td>RefAnyVal(type)</td>
</tr>
<tr>
<td>rethrow</td>
<td>Rethrow</td>
</tr>
<tr>
<td>sizeof valueType</td>
<td>SizeOf(valueType)</td>
</tr>
<tr>
<td>stelem.i1</td>
<td>StoreElem(int8)</td>
</tr>
<tr>
<td>stelem.i2</td>
<td>StoreElem(int16)</td>
</tr>
<tr>
<td>stelem.i4</td>
<td>StoreElem(int32)</td>
</tr>
<tr>
<td>stelem.i8</td>
<td>StoreElem(int64)</td>
</tr>
<tr>
<td>stelem.r4</td>
<td>StoreElem(float32)</td>
</tr>
<tr>
<td>stelem.r8</td>
<td>StoreElem(float64)</td>
</tr>
<tr>
<td>stelem.i</td>
<td>StoreElem(native int)</td>
</tr>
<tr>
<td>stelem.ref</td>
<td>StoreElem(object ref)</td>
</tr>
<tr>
<td>stfld fref/t</td>
<td>StoreField(t,fref)</td>
</tr>
<tr>
<td>stobj classTok</td>
<td>StoreInd(classTok)</td>
</tr>
<tr>
<td>stsfld fref/t</td>
<td>StoreField(t,fref)</td>
</tr>
<tr>
<td>throw</td>
<td>Throw</td>
</tr>
<tr>
<td>unbox valuetype</td>
<td>Unbox(valuetype)</td>
</tr>
</tbody>
</table>