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 eight submachines, which correspond to eight submodules into which the Common Intermediate Language (CIL) instruction set can be decomposed.

1 Introduction

This paper is one outcome of a larger project [11] 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 [10] for the definite assignment rules). As part of this effort, an ASM[1] model has been developed in [7] to formalize the semantics of C♯. To validate this model, we have refined it and made executable (see [12]) in AsmL [14].

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

[1] A detailed definition of ASMs is available in the AsmBook [8, §2].

to the experiments with the CLR is provided through an executable version implemented in AsmL [14]. Upon completion of the AsmL implementation of the entire CLR model, the full details will be made available in [13].

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 many other differences in [2]. 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*, *pointer handling*, *value class instance handling*. Similarly as in [6], CIL is described as a hierarchy of eight sublanguages, which correspond to eight submodules into which the CLR can be decomposed: $\mathrm{CLR}_{\mathcal{I}} \subset \mathrm{CLR}_{\mathcal{C}} \subset \mathrm{CLR}_{\mathcal{O}} \subset \mathrm{CLR}_{\mathcal{E}} \subset \mathrm{CLR}_{\mathcal{P}} \subset \mathrm{CLR}_{\mathcal{VC}} \subset \mathrm{CLR}_{\mathcal{TR}} \subset \mathrm{CLR}_{\mathcal{MP}}$. For each such submodule $\mathrm{CLR}_{\mathcal{L}}$ we build a submachine CLR_{L} which is a conservative extension of its predecessor. The model for the whole CLR is given by the last submachine, i.e. CLR_{MP}. Due to the space limit, we omit here bytecode instructions for arrays, monitors, exceptions, typed references and method pointers but we include them (as well as skipped details in this paper) in the *full* CLR model defined in [2]. This means, the submodules $\mathrm{CLR}_{\mathcal{E}}$, $\mathrm{CLR}_{\mathcal{TR}}$ and $\mathrm{CLR}_{\mathcal{MP}}$ are skipped in the model defined in this paper[2].

The exceptions submodule $\mathrm{CLR}_{\mathcal{E}}$ including the CLR exception handling mechanism and its analysis is also postponed to a separate paper [3] 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 2.6).

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

[2] The submodules $\mathrm{CLR}_{\mathcal{TR}}$ and $\mathrm{CLR}_{\mathcal{MP}}$ introduce typed references and method pointers, respectively.

Fig. 1. The considered CLR instructions

CLR$_\mathcal{I}$ instructions

$$
\begin{aligned}
\mathit{Instr} = \ & \mathit{Execute}(\mathit{Op}) \\
| \ & \mathit{LoadLoc}(\mathit{Local}) \\
| \ & \mathit{StoreLoc}(\mathit{Local}) \\
| \ & \mathit{Branch}(\mathit{Pc}) \\
| \ & \mathit{Cond}(\mathit{Op},\mathit{Pc}) \\
| \ & \mathit{Dup} \\
| \ & \mathit{Pop}
\end{aligned}
$$

CLR$_\mathcal{P}$ instructions

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

CLR$_\mathcal{C}$ instructions

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

CLR$_\mathcal{VC}$ instructions

$$
\begin{aligned}
\mathit{Instr} = \ & \ldots \\
| \ & \mathit{InitObj}(\mathit{ValueType}) \\
| \ & \mathit{CopyObj}(\mathit{ValueType}) \\
| \ & \mathit{Box}(\mathit{ValueType}) \\
| \ & \mathit{Unbox}(\mathit{ValueType})
\end{aligned}
$$

CLR$_\mathcal{O}$ instructions

$$
\begin{aligned}
\mathit{Instr} = \ & \ldots \\
| \ & \mathit{NewObj}(\mathit{MRef}) \\
| \ & \mathit{LoadField}(\mathit{Type},\mathit{FRef}) \\
| \ & \mathit{StoreField}(\mathit{Type},\mathit{FRef}) \\
| \ & \mathit{CallVirt}(\mathit{TailCall},\mathit{Type},\mathit{MRef}) \\
| \ & \mathit{CastClass}(\mathit{Class}) \\
| \ & \mathit{IsInstance}(\mathit{Class}) \\
| \ & \mathit{Jmp}(\mathit{MRef})
\end{aligned}
$$

CLR$_\mathcal{I}$ with instructions required for the compilation of imperative programs of a *while* language. CLR$_\mathcal{I}$ is extended to CLR$_\mathcal{C}$ in Section 2.3 by including instructions used for compilation of static features of classes. Section 2.4 defines an object-based machine CLR$_\mathcal{O}$ which supports instructions for object oriented features. The machine CLR$_\mathcal{P}$ defined in Section 2.5 extends CLR$_\mathcal{O}$ by adding instructions for dealing with pointers. The topmost machine CLR$_\mathcal{VC}$ defined in Section 2.6 provides instructions for dealing with value class instances. Section 3 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*. 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. The last two universes are synonyms for the universe $\mathbb{N}$ of natural numbers. *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. *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 3). However, we specify all the instructions independent on their verifiability status.

Naming conventions We summarize the major naming conventions used in 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, f a field name (or sometimes a field reference) and m a method name.

2.2 The $\mathrm{CLR}_\mathcal{I}$ submodule

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

Environment and State The list of bytecode instructions of the current method is maintained in $code : List(Instr)$. $\mathrm{CLR}_\mathcal{I}$ is dealing only with a single method whose local variable types are given by the list *locTypes*. The universe *DivOp* consists of the division operators `div`, `div.un`, `rem`, `rem.un`. Some operators perform overflow checks; examples are `add.ovf.un` and `sub.ovf.un`. We denote by *OvfOp* the universe of these operators. Upon these definitions, the following set relations hold: $DivOp \subset Op$ and $OvfOp \subset Op$. The dynamic state of the $\mathrm{CLR}_\mathcal{I}$ consists of a frame containing a program counter *pc*, local variable addresses *locAdr* and an evaluation stack *evalStack*. The *pc* runs over the universe *Pc*. The *locAdr* carries the addresses of the local variables and not their values. Although the addresses are not needed for $\mathrm{CLR}_\mathcal{I}$, they are later addressable with the instructions added in $\mathrm{CLR}_\mathcal{P}$. The universe of addresses *Adr* is the interval $Adr = 0 \mathinner{..} maxAdr - 1$ where the non-negative integer *maxAdr* depends on the target architecture. The *mem* is used to store values into the memory locations. The second column of the following declaration defines the initial values of the dynamic functions in the first column:

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

Typed evaluation stack The *evalStack* is specified as a list of typed values. This is a crucial difference wrt the untyped JVM operand stack. The latter is made of uniform 32-bits wide 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*[3]. 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.

$CLRStackType$ = `int32` | `int64` | `native int` | `F`

The *CLRStackType*s serve for the following purposes:

– to define the semantic function *CLRResVal* – it computes the result of all the operators (see the *Execute* rule below);
– 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 *valid* CIL and also specific verification conditions (the formalization of the bytecode verifier is future work).

The *valuesOf* selects the value component of a list of *evalStack* slots, i.e. *valuesOf* applied to $[(val_1, t_1), \ldots, (val_n, t_n)]$ returns $[val_1, \ldots, val_n]$. Similarly, *typesOf* selects the type component of a list of *evalStack* slots. The *CLRTypeOf* extends "upwards" a type to a *CLRStackType* (see [2] for details). Beside the usual list operations (e.g. *push*, *pop*, *take*, *length*, "·")[4], we use different operations for dealing with the *evalStack*: $split(evalStack,n)$ splits off the last n slots of the *evalStack*. More exactly, $split(evalStack,n)$ is the pair $(evalStack', ns)$ of two lists where $evalStack' \cdot ns = evalStack$ and $length(ns) = n$. Similarly, $getVals(evalStack,n)$ splits off the last n slots of the *evalStack* by retrieving *only* their values.

Rules The ASM rules in Fig. 2 describe the dynamic semantics of $\mathrm{CLR}_{\mathcal{I}}$. The machine CLR_I fires the $\textsc{exec}\mathrm{CLR}_I$ rules for the current instruction $code(pc)$. $Execute(op)$ takes the topmost $opNo(op)$ values of the *evalStack* where $opNo(op)$ returns the number of operands of the operator op. If there is no exception case, the result of the semantic function $CLRResVal(op,slots)$ is loaded on the *evalStack*. The result type *CLRResType* of all the operators is defined by the ECMA standard [1, Partition III,§1.5] and is a function of the operator and of the operands' types. The cases when an exception is thrown are: **(a)** division by zero for operators of integral types; **(b)** operations that perform an overflow check and whose results cannot be represented in the result type; **(c)** values that are not "normal" numbers are checked for finiteness or `div`ision/`rem`ainder operations are executed for a minimal value of an integral type and -1. In the following formalizations, *vals* stands for $valuesOf(slots)$ and *types* for $typesOf(slots)$.

[3] Concerning the definition of *CLRStackType*: `native int` stands for `int32` or `int64` whichever is more convenient for the target architecture; floating-point numbers are represented using an internal floating-point type `F`.

[4] The "·" denotes the *append* operation for lists.

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

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

$$
\begin{array}{l}
\textsc{execCLR}_I(instr) \equiv \mathbf{match}\ instr \\
\quad Execute(op) \rightarrow \\
\quad\quad \mathbf{let}\ (evalStack', slots) = split(evalStack, opNo(op))\ \mathbf{in} \\
\quad\quad\quad \mathbf{if}\ \neg ExceptionCase(op, slots)\ \mathbf{then} \\
\quad\quad\quad\quad \mathbf{let}\ (val, t) = (CLRResVal(op, slots), CLRResType(op, typesOf(slots)))\ \mathbf{in} \\
\quad\quad\quad\quad\quad evalStack := evalStack' \cdot [(val, t)] \\
\quad\quad\quad\quad\quad pc := pc + 1 \\
\quad LoadLoc(n) \rightarrow\ \mathbf{let}\ t = locTypes(n)\ \mathbf{in} \\
\quad\quad\quad evalStack := evalStack \cdot [(memVal(locAdr(n), t), CLRTypeOf(t))] \\
\quad\quad\quad pc := pc + 1 \\
\quad StoreLoc(n) \rightarrow\ \mathbf{let}\ (evalStack', [val]) = getVals(evalStack, 1)\ \mathbf{in} \\
\quad\quad\quad \textsc{WriteMem}(locAdr(n), locTypes(n), val) \\
\quad\quad\quad evalStack := evalStack' \\
\quad\quad\quad pc := pc + 1 \\
\quad Branch(t) \rightarrow\ pc := t \\
\quad Cond(op, t) \rightarrow\ \mathbf{let}\ (evalStack', slots) = split(evalStack, opNo(op))\ \mathbf{in} \\
\quad\quad\quad evalStack := evalStack' \\
\quad\quad\quad pc := \mathbf{if}\ CLRResVal(op, slots)\ \mathbf{then}\ t\ \mathbf{else}\ pc + 1 \\
\quad Dup \rightarrow\ \mathbf{let}\ (evalStack' \cdot [(val, t)]) = split(evalStack, 1)\ \mathbf{in} \\
\quad\quad\quad evalStack := evalStack' \cdot [(val, t), (val, t)] \\
\quad\quad\quad pc := pc + 1 \\
\quad Pop \rightarrow\ pop(evalStack) \\
\quad\quad\quad pc := pc + 1
\end{array}
$$

$$
\begin{array}{l}
ExceptionCase(op, slots) \Leftrightarrow DivByZeroCase(op, slots) \vee OverflowCase(op, slots) \\
\quad\quad\quad \vee\ InvNrCase(op, slots)
\end{array}
$$

$$
\begin{array}{l}
DivByZeroCase(op, slots) \Leftrightarrow op \in DivOp \wedge vals(1) = 0 \\
\quad\quad\quad \wedge\ types(i) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\},\ i = 0, 1
\end{array}
$$

$$
\begin{array}{l}
OverflowCase(op, slots) \Leftrightarrow op \in OvfOp \\
\quad\quad\quad \wedge\ Overflow(CLRResVal(op, slots), CLRResType(op, types))
\end{array}
$$

$$
\begin{array}{l}
InvNrCase(op, slots) \Leftrightarrow (op = \texttt{ckfinite} \wedge vals(0) \in \{\texttt{NaN}, \texttt{+infinity}, \texttt{-infinity}\}) \\
\quad\quad\quad \vee\ (op \in \{\texttt{div}, \texttt{rem}\} \wedge vals(0) = min(types(0)) \wedge vals(1) = -1 \\
\quad\quad\quad\quad \wedge\ types(i) \in \{\texttt{int32}, \texttt{int64}, \texttt{native int}\},\ i = 0, 1)
\end{array}
$$

LoadLoc loads the value of a local variable. The value of the local variable n of the declared type t is determined using the derived function $memVal : Map(Adr, Val)$ applied to the address of n and t. The $memVal$ builds up the value of a given type stored in memory at a certain address. In $\mathrm{CLR}_{\mathcal{I}}$, $memVal(adr, t) := mem(adr)$. The type t becomes relevant when we refine the universe Val in $\mathrm{CLR}_{\mathcal{C}}$. *StoreLoc* writes the value of the topmost slot into the memory at the local variable's address. The "write in memory" is defined through the WriteMem that also considers the type of the stored value although this is not needed in $\mathrm{CLR}_{\mathcal{I}}$.

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

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

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

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

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

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

The set of static and instance fields of a class is obtained by applying to the class name the functions *statFields* and *instFields*, respectively. 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 $\mathrm{CLR}_{\mathcal{O}}$). The function *type* applied to a field reference returns the declared type of the field. The functions *paramTypes* and *retType* select the list of parameter types and the return type of a (method) signature, respectively. The function *paramNo* is derived from *paramTypes* and assigns to a signature the length of the parameter types list. We actually use *paramTypes*, *paramNo* and *retType* applied to method references to get the parameter types, parameters number and return type embedded into the corresponding signature. The *locTypes* assigns to every method reference the list of its local variable types ("locals signature"). The *locNo* is defined as the length of the *locTypes*. 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 if the local variables should be automatically initialized to zero by the CLR.

$type : Map(FRef, Type)$
$statFields : Map(Class, \mathcal{P}(FRef))$
$instFields : Map(Class, \mathcal{P}(FRef))$

$retType : Map(Sig, Type)$
$paramTypes : Map(Sig, List(Type))$
$locTypes : Map(MRef, List(Type))$
$zeroInit : Map(MRef, Bool)$

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

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

A frame in $\mathrm{CLR}_{\mathcal{C}}$ is enriched with more information than in $\mathrm{CLR}_{\mathcal{I}}$. The frame components are, in order, the following: a program counter *pc*, local variables addresses *locAdr*, arguments addresses $argAdr : Map(Arg, Adr)$, 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 $\mathrm{CLR}_{\mathcal{C}}$, they become addressable in $\mathrm{CLR}_{\mathcal{P}}$. Unlike CLR, JVM does not have separate instructions for method arguments. We model in a simple manner also the memory allocation (see the end of this section where we provide an abstract specification of the memory management). Upon exiting a method, the addresses allocated for the method's *evalStack* have to be deallocated. Therefore, it is crucial to "remember" the set of stack-allocated addresses for the invoker frame. We extend the stipulations for the initial state as follows:

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

We assume that the program "entrypoint" is a method declared by `Object` which calls the method marked with `entrypoint` in the bytecode (see [6, §10.1] and [2] for justifications of this assumption). We denote by *frame* the currently executed frame: $frame = (pc, locAdr, argAdr, StackAdr, evalStack, meth)$. Note that we separate the current *frame* from the stack of frames, i.e. we do not include *frame* in the *frameStack*.

We consider the approach from [6] to separate the methods transfer and the execution of method bodies. We introduce a switch machine SWITCHCLR that is responsible for the methods transfer. For more details, we refer the interested reader to [2,6]. The universe *Switch* defines the states of this machine:

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

The current state of the submachine SWITCHCLR is specified by the dynamic function $switch : Switch$ whose initial value is *Noswitch*.

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

Class initialization The ECMA standard imposes several rules in [1, Partition I,§8.9.5] concerning the class initialization (for both object classes and value classes as we will see in $\mathrm{CLR}_{\mathcal{VC}}$). The class initialization implies the execution of its initializer. A class may have or may not have an initializer. In the latter case, the CLR creates one which usually contains assignments to the static fields (it might be also "empty", i.e. with a body consisting of a *Return* instruction only) and marks the class with the attribute `beforefieldinit`. If a class is not marked with `beforefieldinit`, then the initializer is executed at the first access to any static or instance field (see $\mathrm{CLR}_{\mathcal{O}}$) of that class, or the first invocation of any static, instance or virtual method (see $\mathrm{CLR}_{\mathcal{O}}$) of that class. If the class is marked with `beforefieldinit`, the invocation of a static method (declared by the class) does not trigger the initialization. In such a case, the initialization is triggered only by a static field access. Let us denote by $beforefieldinit : Map(Class, Bool)$ 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*:

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

The predicate *initialized* is derived from *classState*: $initialized(c)$ 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`:

$$reqinit(c) \Leftrightarrow \neg initialized(c) \wedge \neg beforefieldinit(c)$$

Rules The ASM rules for $\mathrm{CLR}_{\mathcal{C}}$ are defined in Fig. 3. The machine CLR_{C} executes the macro *execScheme* which is parameterized by the machines $\textsc{exec}\mathrm{CLR}_{C}$ and $\textsc{switch}$CLR. The macro *execScheme* is defined as follows[5]. If *switch* is set, i.e. it has a value other than *Noswitch*, then the control is passed to the machine $\textsc{switch}$CLR. Otherwise, as a consequence of the `beforefieldinit` semantics, either a `beforefieldinit` class is initialized through InitializeClass or $\textsc{exec}$CLR fires a rule for the current instruction. We use the abbreviation "**or**" as defined in [8, §2.2.5] for the special case of non-deterministic choice among two rules. InitializeClass arbitrarily chooses a `beforefieldinit` class that is

[5] The *execScheme* is redefined in $\mathrm{CLR}_{\mathcal{E}}$ in [2] while introducing exceptions. If an exception is thrown, the *execScheme* passes the control to the exception handling mechanism. Thus, it prevents the execution of the $\textsc{exec}$CLR machines. Consequently, we do not have to modify the $\textsc{exec}$CLR machines introduced in this paper.

not yet initialized. It then passes the control to the SWITCHCLR to initialize the class.

$$\text{INITIALIZECLASS} \equiv \textbf{choose } c \in \mathit{Class} \textbf{ with } \neg \mathit{initialized}(c) \wedge \mathit{beforefieldinit}(c) \textbf{ do} \\ \quad \mathit{switch} := \mathit{InitClass}(c)$$

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 $\mathit{LoadStatic}(t, c{::}f)$ is the declared type of the field reference $c{::}f$. The same explanation applies for $\mathit{StoreStatic}(t, c{::}f)$. In case of $\mathit{Call}(\_, t, c{::}m)$, t denotes the return type of the considered method reference. $\mathit{LoadStatic}(t,c{::}f)$ pushes on the *evalStack* the value of the field f stored at the address $\mathit{globals}(c{::}f)$. The value of the topmost *evalStack* slot is stored by *StoreStatic* into the address of $c{::}f$. To formalize *LoadArg* and *StoreArg*, we need to determine the types of the current method arguments: $\mathit{argTypes} : \mathit{Map}(\mathit{MRef}, \mathit{List}(\mathit{Type}))$ yields for a method reference the list of argument types. We denote by *argNo* the length of *argTypes*. For every static method reference $c{::}m$, *argTypes* is defined[6] as follows:

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

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

The value of the argument n is loaded on the *evalStack* through $\mathit{LoadArg}(n)$. The instruction $\mathit{StoreArg}(n)$ writes the value of the topmost *evalStack* slot into the address of argument n. For calling a method, a *Call* takes the necessary number of arguments from the *evalStack* and transfers the control to SWITCHCLR. It forwards the boolean information concerning a possible tail call through the switch value *Invoke* passed to SWITCHCLR. The *Return* takes from the *evalStack* zero or one value depending on the return type of the current method and transfers the control to SWITCHCLR together with the returned value (if any).

The switch machine The rules of SWITCHCLR are presented in Fig. 4. The rule $\mathit{Invoke}(\mathit{tail}, c{::}m, \mathit{args})$ handles the context transfer from the current method to the method $c{::}m$. The CLR supports tail calls (this is a crucial difference wrt JVM) since there are .NET languages where the recursion is the only way to express repetition. If the prefix `tail` is attached to a call instruction, then the caller's stack frame is discarded prior making the call. So, when *tail* is *True*, the current frame is not pushed on *frameStack* and the stack addresses allocated for the current frame are deallocated. Otherwise, the current frame is saved on *frameStack*. The frame for invoking $c{::}m$ with the list of arguments *args* becomes the current frame. When setting up this frame with SETFRAME memory is allocated on the stack for arguments and local variables through MAKEARGLOC (see the paragraph on the abstract memory management at the end of this section). The macro MAKEARGLOC writes the values

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

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

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

$$
\begin{array}{l}
\mathit{execScheme}(\textsc{exec}\mathrm{CLR}, \textsc{switch}\mathrm{CLR}) \equiv \\
\quad \textbf{if } \mathit{switch} \neq \mathit{Noswitch} \textbf{ then } \textsc{switch}\mathrm{CLR} \\
\quad \textbf{else } \textsc{InitializeClass} \textbf{ or } \textsc{exec}\mathrm{CLR}(\mathit{code}(\mathit{pc}))
\end{array}
$$

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

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 $\mathit{defVal} : \mathit{Map}(\mathit{Type}, \mathit{Val})$ assigns to every type its "zero", i.e. its default value. The function $\mathit{zero} : \mathit{Map}(\mathit{Type} \times \mathit{MRef}, \mathit{Val} \cup \{\mathit{undef}\})$ 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.

$$\textsc{SetFrame}(c{::}m, \mathit{args}) \equiv$$

Fig. 4. The SWITCHCLR machine

$$
\begin{array}{l}
\textsc{switchCLR} \equiv \textbf{match } \mathit{switch} \\
\quad \mathit{Invoke}(\mathit{tail}, c{::}m, \mathit{args}) \rightarrow \begin{array}[t]{l} \textbf{if } \mathit{tail} \textbf{ then } \textsc{DeAllocMem}(\mathit{StackAdr}') \\ \textbf{else } \mathit{push}(\mathit{frameStack}, \mathit{frame}) \\ \textsc{SetFrame}(c{::}m, \mathit{args}) \\ \mathit{switch} := \mathit{Noswitch} \\ \textbf{where } (\_, \_, \_, \mathit{StackAdr}', \_, \_) = \mathit{top}(\mathit{frameStack}) \end{array} \\
\quad \mathit{InitClass}(c) \rightarrow \begin{array}[t]{l} \textbf{if } \mathit{classState}(c) = \mathit{Linked} \textbf{ then} \\ \quad \mathit{classState}(c) := \mathit{Initialized} \\ \quad \textsc{InitStatFields}(c) \\ \quad \textsc{SetFrame}(c{::}\texttt{.cctor}, [\,]) \\ \quad \mathit{push}(\mathit{frameStack}, \mathit{frame}) \\ \quad \mathit{switch} := \mathit{Noswitch} \end{array} \\
\quad \mathit{Result}(\mathit{slots}) \rightarrow \begin{array}[t]{l} \textbf{if } \mathit{methNm}(\mathit{meth}) = \texttt{.cctor} \textbf{ then } \textsc{PopFrame}(0, [\,]) \\ \textbf{else } \textsc{PopFrame}(1, \mathit{slots}) \\ \mathit{switch} := \mathit{Noswitch} \end{array}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{pc} := 0 \\
\quad \mathit{evalStack} := [\,] \\
\quad \mathit{meth} := c{::}m \\
\quad \textsc{MakeArgLoc}(\mathit{argTypes}(c{::}m), \mathit{locTypes}(c{::}m), \mathit{args} \cdot \mathit{zero}(\mathit{locTypes}(c{::}m), c{::}m))
\end{array}
$$

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

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`.

$$
\begin{array}{l}
\textsc{InitStatFields}(c) \equiv \textbf{forall } f \in \mathit{statFields}(c) \textbf{ do} \\
\qquad\qquad \textsc{WriteMem}(\mathit{globals}(f), \mathit{type}(f), \mathit{defVal}(\mathit{type}(f)))
\end{array}
$$

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. If the 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 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[7].

$$
\begin{array}{l}
\textsc{PopFrame}(k, \mathit{slots}) \equiv \\
\quad \textbf{let } (\mathit{frameStack}', [(\mathit{pc}', \mathit{locAdr}', \mathit{argAdr}', \mathit{StackAdr}', \mathit{evalStack}', \mathit{meth}')]) \\
\qquad = \mathit{split}(\mathit{frameStack}, 1) \textbf{ in}
\end{array}
$$

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

$$\begin{array}{ll}
pc & := pc' + k \\
locAdr & := locAdr' \\
argAdr & := argAdr' \\
evalStack & := evalStack' \cdot slots \\
meth & := meth' \\
frameStack & := frameStack' \\
\multicolumn{2}{l}{\textsc{DeAllocMem}(StackAdr')}
\end{array}$$

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.

Abstract memory management We explain the details on how the addresses – elements of *Adr* – are allocated. We stick our formalization to the two kinds of allocations – stack and heap allocation – without considering a garbage collector. However, we provide a simple notion of stack de-allocation. We consider two dynamic functions that keep track of the addresses allocated on the stack and on the heap: $StackAdr : \mathcal{P}(Adr)$ and $HeapAdr : \mathcal{P}(Adr)$. Accordingly to their definitions, in the initial state of CLR_C, the *StackAdr* and the *HeapAdr* are $\emptyset$.

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

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

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

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

We use the partial functions *encode* and *decode* to determine the sequence of bytes associated to a value of a simple value type[8] or of a pointer type and to obtain the value associated to a sequence of bytes. For a value *val* of a type t, the functions *encode* and *decode* satisfy the following equations:

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

We are now able to determine when there is enough space in *Adr* to be allocated for a list of values[9]. 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 on the target architecture, we assume the external function $validAdr : Map(Adr, Bool)$

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

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

decides whether an address is valid. One can find more details on this function in Section 2.5. In the definition below, n stands for $length(sizes)$:

$$SpaceFor(sizes) = \big\{(adr_i)_{i=0}^{n-1} \in Adr^n \mid validAdr(adr_i)\ \forall i = 0, n-1 \text{ and } \biguplus_{i=0}^{n-1} [adr_i, adr_i + sizes(i)) \subseteq Adr \setminus (StackAdr \cup HeapAdr)\big\}$$

If an attempt to allocate on the stack fails, a `StackOverflowException` is thrown (see $\text{CLR}_{\mathcal{E}}$ in [2]). Similarly, if one needs to allocate on the heap, then an `OutOfMemoryException` is raised.

The following explanations are for the macro MAKEARGLOC defined in [2], which we used when setting the frame for a method call. MAKEARGLOC is applied to three lists: two lists of types $types_1$ and $types_2$ and a list of values *vals*. It assumes that the sum of the first two lists' lengths is equal with the length of *vals*. The macro allocates addresses on the stack that would be needed for arguments and local variables of types $types_1$ and $types_2$ and writes the values *vals* in these addresses. It first checks if there is enough space to be allocated. It does not assume anything concerning the order of the arguments and local variables on the stack. At the same time, the allocated addresses are pushed on the *StackAdr*.

The macro DEALLOCMEM has been used when exiting a method. It reclaims the memory space allocated on the stack for arguments and local variables but also the memory allocated in the local memory pool defined in Section 2.5 (see $\text{CLR}_{\mathcal{P}}$).

DEALLOCMEM(A) $\equiv$
  $StackAdr := A$
  forall adr **in** $StackAdr \setminus A$ **do** $mem(adr) := undef$

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

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

Environment and State The universe *ObjType* describes object types. An object type is a reference type of a self-describing value. The object types are the object classes (not the value classes) $ObjType = ObjClass$. In [2] we consider in *ObjType* also the array types. The universe *CLRStackType* is extended with the special type O corresponding to *ObjType*. The function *CLRTypeOf* maps every object type to the special type O.

$CLRStackType = \ldots \mid O$

Beside static fields, in $\text{CLR}_{\mathcal{O}}$ we have also instance fields. The external function *fieldOffSet* computes for every instance field of a class the *field offset* within an instance of the given class. We denote by $ObjRef$ the universe of object references (note that the $evalStack$ does not work directly with objects but with references to objects). The $fieldAdr$ assigns to every instance field of an object reference its allocated address.

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

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

The objects, i.e. instances of object classes[10], are allocated on the heap. All objects on the heap are known as "boxed objects" in contrast with the value type instances introduced in $\text{CLR}_{\mathcal{VC}}$ and known as "unboxed objects". A class object is represented by its type and the addresses (and not the values as in JVM [6]) of its instance fields. The function $actualTypeOf : Map(ObjRef, Type)$ records the actual type of an object on the heap.

The function mem is redefined as $mem : Map(Adr, Byte \cup ObjRef \cup \{undef\})$. Also $memVal$ and WRITEMEM are refined as follows:

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

$$
\begin{aligned}
\text{WRITEMEM}(adr, t, val) \equiv {}& \textbf{if } t \in ObjType \textbf{ then } mem(adr) := val \\
& \textbf{else forall } i \in [0..sizeOf(t) - 1] \textbf{ do} \\
& \qquad mem(adr + i) := val(i)
\end{aligned}
$$

Rules Fig. 5, 6 and 7 define the rules for $\text{CLR}_{\mathcal{O}}$. $NewObj$ allocates an instance of an object class 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:

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

$$
\begin{aligned}
&\text{ALLOCFIELDS}(adr, t) \equiv \\
&\quad \textbf{skip}
\end{aligned}
$$

The object is allocated in the $HeapAdr$. The addresses of the instance fields are computed using the object starting address and the field offsets. In $\text{CLR}_{\mathcal{VC}}$, the fields might be of a value class type. In that case, one has to compute also the

[10] In [2] we have also arrays.

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

$$
\begin{array}{l}
\mathrm{CLR}_{\mathcal{O}} \equiv \mathit{execScheme}(\textsc{exec}\mathrm{CLR}_{\mathcal{O}}, \textsc{switch}\mathrm{CLR}) \\
\\
\textsc{exec}\mathrm{CLR}_{\mathcal{O}}(\mathit{instr}) \equiv \\
\quad \textsc{exec}\mathrm{CLR}_{\mathcal{C}}(\mathit{instr}) \\
\quad \textbf{match } \mathit{instr} \\
\quad\quad \mathit{NewObj}(c{::}\texttt{.ctor}) \rightarrow \\
\quad\quad\quad \textbf{if } c \in \mathit{ObjClass} \textbf{ then} \\
\quad\quad\quad\quad \textbf{if } \neg \mathit{reqinit}(c) \textbf{ then} \\
\quad\quad\quad\quad\quad \textbf{if } \mathit{SpaceFor}([\mathit{sizeOf}(c)]) \neq \emptyset \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \textbf{let } (\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{paramNo}(c{::}\texttt{.ctor})) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{let } r = \mathit{new}(\mathit{ObjRef}, c) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(r, O)] \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{actualTypeOf}(r) := c \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{forall } f \in \mathit{instFields}(c) \textbf{ do} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{fieldAdr}(r, f), \mathit{type}(f), \mathit{defVal}(\mathit{type}(f))) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{switch} := \mathit{Invoke}(\mathit{False}, c{::}\texttt{.ctor}, [r] \cdot \mathit{vals}) \\
\quad\quad\quad\quad \textbf{else } \mathit{switch} := \mathit{InitClass}(c) \\
\quad\quad \mathit{LoadField}(t, c{::}f) \rightarrow \\
\quad\quad\quad \textbf{if } \mathit{initialized}(c) \textbf{ then} \\
\quad\quad\quad\quad \textbf{let } (\mathit{evalStack}', [x]) = \mathit{getVals}(\mathit{evalStack}, 1) \textbf{ in} \\
\quad\quad\quad\quad\quad \textbf{if } \mathit{static}(c{::}f) \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{globals}(c{::}f), t), \mathit{CLRTypeOf}(t))] \\
\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1 \\
\quad\quad\quad\quad\quad \textbf{elseif } x \neq \texttt{null} \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{fieldAdr}(x, c{::}f), t), \mathit{CLRTypeOf}(t))] \\
\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1 \\
\quad\quad\quad \textbf{else } \mathit{switch} := \mathit{InitClass}(c) \\
\quad\quad \mathit{StoreField}(t, c{::}f) \rightarrow \textbf{ if } \mathit{initialized}(c) \textbf{ then} \\
\quad\quad\quad\quad \textbf{let } (\mathit{evalStack}', [x, \mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 2) \textbf{ in} \\
\quad\quad\quad\quad\quad \textbf{if } \mathit{static}(c{::}f) \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{globals}(c{::}f), t, \mathit{val}) \\
\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1 \\
\quad\quad\quad\quad\quad \textbf{elseif } x \neq \texttt{null} \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{fieldAdr}(x, c{::}f), t, \mathit{val}) \\
\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1 \\
\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \\
\quad\quad\quad \textbf{else } \mathit{switch} := \mathit{InitClass}(c) \\
\quad\quad \mathit{Call}(\mathit{tail}, \_, c{::}m) \rightarrow \\
\quad\quad\quad \textbf{if } \mathit{instance}(c{::}m) \textbf{ then} \\
\quad\quad\quad\quad \textbf{if } \neg \mathit{reqinit}(c) \textbf{ then} \\
\quad\quad\quad\quad\quad \textbf{let } (\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{argNo}(c{::}m)) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad \textbf{if } \mathit{vals}(0) \neq \texttt{null} \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \\
\quad\quad\quad\quad\quad\quad\quad \mathit{switch} \quad := \mathit{Invoke}(\mathit{tail}, c{::}m, \mathit{vals}) \\
\quad\quad\quad\quad \textbf{else } \mathit{switch} := \mathit{InitClass}(c)
\end{array}
$$

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

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

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 2.6. Beside the allocation, *NewObj* 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*. In [2], we explain the differences between the JVM `new` and CLR `newobj`.

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

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

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

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

Fig. 7. The execution of non-verifiable $\mathrm{CLR}_{\mathcal{O}}$ instructions

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

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 $\mathrm{CLR}_{\mathcal{E}}$ in [2]). *IsInstance* pops from *evalStack* a reference to a boxed object. If the reference is `null` or the actual type of the reference is not compatible with the given class, then a `null` reference is pushed on *evalStack*. Otherwise, it lets the *evalStack* unchanged. The interested reader can find in [2] a detailed comparison of the instructions *IsInstance* and *CastClass* with their correspondents in JVM. Although *Jmp* is not verifiable, we still model its semantics (see Fig. 7). If the method is declared by a class which requires initialization, *Jmp* initializes the class first. *Jmp* is an optimization of a tail call. Consequently, the current frame is discarded while invoking the given method. Note that, the given method is also the method to be invoked - there is no lookup. The invocation arguments are exactly the arguments of the current frame *at the time* when *Jmp* is fired.

2.5 The $\mathrm{CLR}_{\mathcal{P}}$ submodule

$\mathrm{CLR}_{\mathcal{P}}$ extends $\mathrm{CLR}_{\mathcal{O}}$ with pointer types, i.e. types whose values are memory addresses. $\mathrm{CLR}_{\mathcal{P}}$ provides type-safe operations on pointers (e.g. "read"/"write" a value from/into the address referenced by a pointer) and non-verifiable operations (e.g. initialize a block of memory to a given value, copy data from memory to memory). The main purpose of having pointer types is to permit methods to receive arguments and return values "by reference".

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

$$
\begin{array}{ll}
\mathit{CLRStackType} = \ldots \mid \& & \mathit{SignedInt} = \texttt{int8} \mid \texttt{int16} \mid \texttt{int32} \mid \texttt{int64} \\
\mathit{Float} = \texttt{float32} \mid \texttt{float64} & \mathit{UnsignedInt} = \texttt{uint8} \mid \texttt{uint16} \mid \texttt{uint32} \mid \texttt{uint64} \\
\multicolumn{2}{l}{\mathit{LoadIndType} = \mathit{SignedInt} \mid \mathit{UnsignedInt} \mid \mathit{Float} \mid \texttt{native int} \mid \texttt{object}} \\
\multicolumn{2}{l}{\mathit{StoreIndType} = \mathit{SignedInt} \mid \mathit{Float} \mid \texttt{native int} \mid \texttt{object}}
\end{array}
$$

In Section 2.3 we have introduced the external function *validAdr* to check addresses for validity. An address is invalid if it is `null` or is not in the range of *Adr* or is not "naturally aligned" for the target architecture or is "not mapped"

Fig. 8. The execution of $\mathrm{CLR}_{\mathcal{P}}$ instructions

$$
\begin{array}{l}
\mathrm{CLR}_{P} \equiv \mathit{execScheme}(\textsc{exec}\mathrm{CLR}_{P}, \textsc{switch}\mathrm{CLR}) \\
\\
\textsc{exec}\mathrm{CLR}_{P}(\mathit{instr}) \equiv \\
\quad \textsc{exec}\mathrm{CLR}_{O}(\mathit{instr}) \\
\quad \mathbf{match}\ \mathit{instr} \\
\qquad \mathit{LoadLocA}(n) \rightarrow \ \mathbf{if}\ \mathit{zeroInit}(\mathit{meth})\ \mathbf{then} \\
\qquad\qquad\qquad\qquad \mathit{evalStack} := \mathit{evalStack} \cdot [(\mathit{locAdr}(n), \&)] \\
\qquad\qquad\qquad\qquad \mathit{pc} := \mathit{pc} + 1 \\
\qquad \mathit{LoadArgA}(n) \rightarrow \ \mathit{evalStack} := \mathit{evalStack} \cdot [(\mathit{argAdr}(n), \&)] \\
\qquad\qquad\qquad\quad \mathit{pc} := \mathit{pc} + 1 \\
\qquad \mathit{LoadStaticA}(\_, c{::}f) \rightarrow \ \mathbf{if}\ \mathit{initialized}(c)\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\quad \mathit{evalStack} := \mathit{evalStack} \cdot [(\mathit{globals}(c{::}f), \&)] \\
\qquad\qquad\qquad\qquad\quad \mathit{pc} := \mathit{pc} + 1 \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{switch} := \mathit{InitClass}(c) \\
\qquad \mathit{LoadFieldA}(\_, c{::}f) \rightarrow \\
\qquad\quad \mathbf{if}\ \mathit{initialized}(c)\ \mathbf{then} \\
\qquad\qquad \mathbf{let}\ (\mathit{evalStack}', [x]) = \mathit{getVals}(\mathit{evalStack}, 1)\ \mathbf{in} \\
\qquad\qquad\quad \mathbf{if}\ \mathit{static}(c{::}f)\ \mathbf{then}\ \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{globals}(c{::}f), \&)] \\
\qquad\qquad\qquad\qquad\qquad \mathit{pc} := \mathit{pc} + 1 \\
\qquad\qquad\quad \mathbf{elseif}\ x \neq \texttt{null}\ \mathbf{then}\ \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{fieldAdr}(x, c{::}f), \&)] \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pc} := \mathit{pc} + 1 \\
\qquad\quad \mathbf{else}\ \mathit{switch} := \mathit{InitClass}(c) \\
\qquad \mathit{LoadInd}(t) \rightarrow \ \mathbf{let}\ (\mathit{evalStack}', [\mathit{adr}]) = \mathit{getVals}(\mathit{evalStack}, 1)\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{if}\ \mathit{validAdr}(\mathit{adr})\ \mathbf{then} \\
\qquad\qquad\qquad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{adr}, t), \mathit{CLRTypeOf}(t))] \\
\qquad\qquad\qquad\quad \mathit{pc} := \mathit{pc} + 1 \\
\qquad \mathit{StoreInd}(t) \rightarrow \ \mathbf{let}\ (\mathit{evalStack}', [\mathit{adr}, \mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 2)\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{if}\ \mathit{validAdr}(\mathit{adr})\ \mathbf{then} \\
\qquad\qquad\qquad\quad \textsc{WriteMem}(\mathit{adr}, t, \mathit{val}) \\
\qquad\qquad\qquad\quad \mathit{evalStack} := \mathit{evalStack}' \\
\qquad\qquad\qquad\quad \mathit{pc} := \mathit{pc} + 1 \\
\qquad \textsc{NonVerifiable}_{P}
\end{array}
$$

into the process. An address is "naturally aligned" if it is aligned wrt the machine dependent `native int` type. Note that, as a consequence of the definition of *SpaceFor*, the results of all CIL instructions that return addresses (e.g. *LoadLocA* and *LoadArgA*) are valid.

Rules The rules for $\mathrm{CLR}_{\mathcal{P}}$ are defined in Fig. 8 and 9. Assuming that the *zeroInit* flag of the current method is set, $\mathit{LoadLocA}(n)$ pushes the address of the local variable indexed with n on the *evalStack*. If the *zeroInit* is not set, then a `VerificationException` is thrown (see $\mathrm{CLR}_{\mathcal{E}}$ in [2]). Similarly, $\mathit{LoadArgA}(n)$ pushes on the *evalStack* the address of the current method argument indexed with n . In JVM [6], one cannot take the address of local variables and arguments. The address of a static field is loaded using *LoadStaticA*. If the class which declares the static field is not yet initialized, the execution proceeds first with the class initialization. *LoadFieldA* is similar with *LoadStaticA* and can be

Fig. 9. The execution of non-verifiable $\text{CLR}_{\mathcal{P}}$ instructions

$$
\begin{array}{l}
\textsc{NonVerifiable}_{P} \equiv \\
\quad \textit{InitBlock} \rightarrow\ \textbf{let}\ (\textit{evalStack}', [\textit{adr}, \textit{val}, \textit{size}]) = \textit{getVals}(\textit{evalStack}, 3)\ \textbf{in} \\
\qquad\qquad\qquad \textbf{if}\ \textit{validAdr}(\textit{adr})\ \textbf{then} \\
\qquad\qquad\qquad\quad \textbf{forall}\ i = 0, \textit{size} - 1\ \textbf{do}\ \textit{mem}(\textit{adr} + i) := \textit{val} \\
\qquad\qquad\qquad\quad \textit{evalStack} := \textit{evalStack}' \\
\qquad\qquad\qquad\quad \textit{pc} := \textit{pc} + 1 \\
\quad \textit{CopyBlock} \rightarrow \\
\qquad \textbf{let}\ (\textit{evalStack}', [\textit{dest\_adr}, \textit{src\_adr}, \textit{size}]) = \textit{getVals}(\textit{evalStack}, 3)\ \textbf{in} \\
\qquad\quad \textbf{if}\ \textit{validAdr}(\textit{dest\_adr}) \wedge \textit{validAdr}(\textit{src\_adr})\ \textbf{then} \\
\qquad\qquad \textbf{if}\ \neg \textit{overlap}(\textit{dest\_adr}, \textit{src\_adr}, \textit{size})\ \textbf{then} \\
\qquad\qquad\quad \textbf{forall}\ i = 0, \textit{size} - 1\ \textbf{do}\ \textit{mem}(\textit{dest\_adr} + i) := \textit{mem}(\textit{src\_adr} + i) \\
\qquad\qquad\quad \textit{evalStack} := \textit{evalStack}' \\
\qquad\qquad\quad \textit{pc} := \textit{pc} + 1 \\
\qquad\qquad \textbf{else}\ \textsc{Report}(\textit{UndefinedBehavior}) \\
\quad \textit{LocAlloc} \rightarrow\ \textbf{let}\ (\textit{evalStack}', [\textit{size}]) = \textit{getVals}(\textit{evalStack}, 1)\ \textbf{in} \\
\qquad\qquad\qquad \textbf{if}\ \textit{SpaceFor}([\textit{size}]) \neq \emptyset\ \textbf{then} \\
\qquad\qquad\qquad\quad \textbf{choose}\ [\textit{adr}] \in \textit{SpaceFor}([\textit{size}])\ \textbf{do} \\
\qquad\qquad\qquad\qquad \textit{evalStack} := \textit{evalStack}' \cdot [(\textit{adr}, \&)] \\
\qquad\qquad\qquad\qquad \textbf{if}\ \textit{zeroInit}(\textit{meth})\ \textbf{then} \\
\qquad\qquad\qquad\qquad\quad \textbf{forall}\ i = 0, \textit{size} - 1\ \textbf{do}\ \textit{mem}(\textit{adr} + i) := 0 \\
\qquad\qquad\qquad\qquad \textit{StackAdr} := \textit{StackAdr} \cup [\textit{adr}, \textit{adr} + \textit{size}) \\
\qquad\qquad\qquad\quad \textit{pc} := \textit{pc} + 1
\end{array}
$$

used for both static and instance fields. *LoadInd* takes the value of the topmost *evalStack* slot, which is supposed to be a pointer (address) and loads the value stored at this address. *StoreInd* takes the values of the two topmost *evalStack* slots, which are supposed to be a pointer (address) and a value. It then stores the value at the address. In both cases, of *LoadInd* and *StoreInd*, the address must be a valid address, otherwise a `NullReferenceException` is thrown (see $\text{CLR}_{\mathcal{E}}$ in [2]).

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. 8 offer support for computing addresses of variables.

The following explanations apply to the non-verifiable $\text{CLR}_{\mathcal{P}}$ instructions in Fig. 9. *InitBlock* 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, in order, a "pointer" (the block's first address), a value and the "size" of the block. It writes the value into a number of addresses given by the block's "size" starting with the address given by the "pointer". *CopyBlock* copies data from memory to memory. It takes the values of the three topmost *evalStack* slots, which are supposed to be, in order, a "destination" address, a "source" address and a "size" of a block of addresses. It then copies a number of bytes given by the block's "size" from the "source" address to the "destination" address. The "destination" and "source" addresses must be valid addresses and the "destination" and "source" areas shall not overlap. If they overlap, the behavior is undefined. The predicate *overlap* decides whether two blocks of addresses overlap:

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

The *LocAlloc* instruction is used for the compilation of a C♯ `stackalloc` statement (see [7] for details). It allocates space on the stack, in the so-called *local memory pool* (see [1, Partition I,§12.3.2.4] for details). It takes the value of the topmost *evalStack* slot, which represents the "size" of the block to be allocated. If there is sufficient space for a block of the given "size", the starting address of the arbitrary choosen 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.

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

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

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

$$\begin{array}{ll} CLRStackType = \ldots \mid ValueClass & LoadIndType = \ldots \mid ValueClass \\ Class = \ldots \mid ValueClass & StoreIndType = \ldots \mid ValueClass \end{array}$$

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

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

$$
\begin{array}{l}
\text{WRITEMEM}(adr, t, val) \equiv \textbf{if } t \in \mathit{ObjType} \textbf{ then } mem(adr) := val \\
\qquad \textbf{elseif } t \in \mathit{ValueClass} \textbf{ then} \\
\qquad\quad \textbf{forall } f \in \mathit{instFields}(t) \textbf{ do} \\
\qquad\qquad \text{WRITEMEM}(\mathit{fieldAdr}(adr, f), type(f), val(f)) \\
\qquad \textbf{else forall } i \in [0..\mathit{sizeOf}(t) - 1] \textbf{ do } mem(adr + i) := val(i)
\end{array}
$$

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 $\mathit{addressOf} : \mathit{Map}(\mathit{ObjRef}, \mathit{Adr})$ applied to the corresponding boxed object reference.

Rules Fig. 10 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.

$$
\begin{array}{l}
\textbf{let } adr = \mathit{stackalloc}(vc) \textbf{ in } P \equiv \\
\quad \textbf{choose } [adr] \in \mathit{SpaceFor}([n]) \textbf{ do} \\
\qquad \mathit{StackAdr} := \mathit{StackAdr} \cup [adr, adr + n) \\
\qquad \text{ALLOCFIELDS}(adr, vc) \\
\quad \textbf{seq } P \\
\quad \textbf{where } n = \mathit{sizeOf}(vc)
\end{array}
\qquad
\begin{array}{l}
\text{ALLOCFIELDS}(adr, t) \equiv \\
\quad \textbf{if } t \in \mathit{ValueClass} \textbf{ then} \\
\qquad \textbf{forall } f \in \mathit{instFields}(t) \textbf{ do} \\
\qquad\quad \textbf{let } a = adr + \mathit{fieldOffSet}(t, f) \textbf{ in} \\
\qquad\qquad \mathit{fieldAdr}(adr, f) := a \\
\qquad\qquad \text{ALLOCFIELDS}(a, type(f))
\end{array}
$$

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

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

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

The *InitObj* instruction initializes an instance of a value type. It takes the value of the topmost *evalStack* slot, which is supposed to be a pointer to a value type instance. Then, it initializes all the instance fields of the instance to

Fig. 10. The execution of $\mathrm{CLR}_{\mathcal{VC}}$ instructions

$$
\begin{array}{l}
\mathrm{CLR}_{VC} \equiv \mathit{execScheme}(\textsc{execCLR}_{VC}, \textsc{switchCLR})\\[1em]
\textsc{execCLR}_{VC}(\mathit{instr}) \equiv\\
\quad \textsc{execCLR}_{P}(\mathit{instr})\\
\quad \textbf{match } \mathit{instr}\\
\quad\quad \mathit{NewObj}(\mathit{vc}::\texttt{.ctor}) \rightarrow\\
\quad\quad\quad \textbf{if } \mathit{vc} \in \mathit{ValueClass}\\
\quad\quad\quad\quad \textbf{if } \neg \mathit{reqinit}(\mathit{vc}) \textbf{ then}\\
\quad\quad\quad\quad\quad \textbf{if } \mathit{SpaceFor}([\mathit{sizeOf}(\mathit{vc})]) \neq \emptyset \textbf{ then}\\
\quad\quad\quad\quad\quad\quad \textbf{let } (\mathit{evalStack}', \mathit{vals}) = \mathit{getVals}(\mathit{evalStack}, \mathit{paramNo}(\mathit{vc}::\texttt{.ctor})) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad\quad \textbf{let } \mathit{adr} = \mathit{stackalloc}(\mathit{vc}) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{memVal}(\mathit{adr}, \mathit{vc}), \mathit{vc})]\\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{forall } f \in \mathit{instFields}(\mathit{vc}) \textbf{ do}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{fieldAdr}(\mathit{adr}, f), \mathit{type}(f), \mathit{defVal}(\mathit{type}(f)))\\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{switch} := \mathit{Invoke}(\mathit{False}, \mathit{vc}::\texttt{.ctor}, [\mathit{adr}] \cdot \mathit{vals})\\
\quad\quad\quad\quad \textbf{else } \mathit{switch} := \mathit{InitClass}(\mathit{vc})\\[0.5em]
\quad\quad \mathit{InitObj}(\mathit{vt}) \rightarrow \textbf{ let } (\mathit{evalStack}', [\mathit{adr}]) = \mathit{getVals}(\mathit{evalStack}, 1) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{adr}, \mathit{vt}, \mathit{defVal}(\mathit{vt}))\\
\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}'\\
\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1\\[0.5em]
\quad\quad \mathit{CopyObj}(\mathit{vt}) \rightarrow \textbf{ let } (\mathit{evalStack}', [\mathit{dest\_adr}, \mathit{src\_adr}]) = \mathit{getVals}(\mathit{evalStack}, 2) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad \textbf{if } \mathit{validAdr}(\mathit{dest\_adr}) \wedge \mathit{validAdr}(\mathit{src\_adr}) \textbf{ then}\\
\quad\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{src\_adr}, \mathit{vt}, \mathit{memVal}(\mathit{dest\_adr}, \mathit{vt}))\\
\quad\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}'\\
\quad\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1\\[0.5em]
\quad\quad \mathit{Box}(\mathit{vt}) \rightarrow \textbf{ if } \mathit{SpaceFor}([\mathit{vt}]) \textbf{ then}\\
\quad\quad\quad\quad\quad \textbf{let } (\mathit{evalStack}', [\mathit{val}]) = \mathit{getVals}(\mathit{evalStack}, 1) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad \textbf{let } r = \mathit{new}(\mathit{ObjRef}) \textbf{ and } \mathit{adr} = \mathit{heapalloc}(\mathit{vt}) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad\quad \mathit{actualTypeOf}(r) := \mathit{vt}\\
\quad\quad\quad\quad\quad\quad\quad \mathit{addressOf}(r) := \mathit{adr}\\
\quad\quad\quad\quad\quad\quad\quad \textsc{WriteMem}(\mathit{adr}, \mathit{vt}, \mathit{val})\\
\quad\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(r, O)]\\
\quad\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1\\[0.5em]
\quad\quad \mathit{Unbox}(\mathit{vt}) \rightarrow \textbf{ let } (\mathit{evalStack}', [r]) = \mathit{getVals}(\mathit{evalStack}, 1) \textbf{ in}\\
\quad\quad\quad\quad\quad\quad \textbf{if } \mathit{actualTypeOf}(r) = \mathit{vt} \textbf{ then}\\
\quad\quad\quad\quad\quad\quad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [(\mathit{addressOf}(r), \&)]\\
\quad\quad\quad\quad\quad\quad\quad \mathit{pc} := \mathit{pc} + 1
\end{array}
$$

the default value of the proper type by means of the recursively defined macro WriteMem. The *CopyObj* instruction copies an instance of a value type. It takes two pointers from the *evalStack* and copies the value type object stored at the address given by the pointer to the address given by the second pointer.

However, if one pointer refers to an invalid address, a `NullReferenceException` is thrown (see $\mathrm{CLR}_{\mathcal{E}}$ in [2]).

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

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

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

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

3 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 eight stepwise refined CLR program layers. We assume that the inputs of the interpreter are CIL bytecode programs successfully loaded and linked (i.e. prepared and verified to satisfy the required link-time constraints). As a next step of our project, we propose ourselves to relax the assumption that the CIL code is verified by accompanying the execution machine with a run-time checking machine. This *defensive* machine is going to serve for proving the *soundness* and *completeness* of the CLR bytecode verifier.

References

1. Common Language Infrastructure (CLI), Standard ECMA–335. Web pages at `http://www.ecma-international.org/publications/`.
2. Nicu G. Fruja. A Modular Design for the Common Language Runtime (CLR) Architecture. Technical Report, ETH Zürich, 2005.
3. Nicu G. Fruja and Egon Börger. Analysis of the .NET CLR Exception Handling Mechanism. To be submitted to *.NET Technologies'05*, 2005.
4. Andrew D. Gordon and Don Syme. Typing a Multi-Language Intermediate Code. Microsoft Technical Report MSR-TR-2000-106, 2000.
5. K. J. Gough. Stacking them up: a Comparison of Virtual Machines. ACM International Conference Proceeding Series. pag. 55–61. IEEE Computer Society, Washington, DC, USA, 2001.
6. R. F. Stärk, J. Schmid, E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.

7. E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C♯. To appear in *Journal Theoretical Computer Science*, 2005.
8. E. Börger and R. F. Stärk. Abstract State Machines–A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.
9. Nicu G. Fruja. Specification and Implementation Problems for C♯. Proceedings of the Workshop on Abstract State Machines (*ASM'04*), Germany, 2004.
10. Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C♯. *Journal of Object Technology*, vol. 3, no. 9, 2004.
11. Nicu G. Fruja. Type Safety in C♯ and .NET CLR. PhD Thesis in preparation.
12. Horatiu V. Jula and Nicu G. Fruja. An Executable Specification of C♯. Proceedings of the Workshop on Abstract State Machines (*ASM'05*), France, 2005.
13. C. Marrocco. An Executable Specification of the .NET CLR. Diploma Thesis guided by Nicu G. Fruja, ETH Zürich, in preparation.
14. AsmL, Foundations of Software Engineering Group, Microsoft Research, Web pages at `http://research.microsoft.com/foundations/AsmL/`.