Type Safety of Generics for the .NET Common Language Runtime

Nicu G. Fruja

Computer Science Department, ETH Zürich, Switzerland
fruja@inf.ethz.ch

Abstract. The Microsoft .NET Common Language Runtime (CLR) offers support for generic types and methods. We develop a mathematical specification for the generics design through a type system and a model for the semantics of a subset of bytecode instructions with generics. We formalize the type-consistency checks performed for the subset by the CLR bytecode verifier. We then prove that adding support for generics maintains the type safety of the CLR.

1 Introduction

We have proved in [5] the soundness of the CLR bytecode verifier. The soundness proof takes advantage of the models for the CLR semantics [6, 7]. Kennedy and Syme proposed adding support for generics in the .NET CLR [11]. Their proposal was at the basis of the generics implementation in the .NET Framework v2.0 [1]. The official specification for the CLR generics support is given in prose form in the ECMA Standard [4].

Several versions of Eiffel turned out to be unsafe also due to the variance on generic parameters [3, 8]. Type holes [10] have been identified also in Generic Java [9, 12]. Consequently, the following question arises: *Is type safety preserved after adding generics as specified in the ECMA Standard* [4]?

So far, this question has only been addressed by Yu, Kennedy and Syme in [13]. They focus on aspects of the generics implementation, e.g., specialization of generic code up to data representation, efficient support for runtime types. As their goal was not the type safety, their formalization does not include: *variance on generic parameters*, *constraint types* and *boxed types* (critical for the specification of constraint types). These are exactly the generics features due to which the type safety might be violated.

In the context of generic parameter variance, virtual method calls are problematic in ensuring the type safety. Let us assume, for example, that the bytecode contains a virtual *call* of the method $C::M$. Let $D::M$ be the method that will be *invoked* at runtime. The following aspects are critical for the type safety: (1) due to the variance, the signature of $D::M$ does not necessarily match the signature of $C::M$; (2) if $C::M$ is a generic method, $D::M$ shall also be generic but its constraint types do not necessarily match the corresponding constraint types of $C::M$.

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 325–341, 2006.

This paper also considers generic parameter variance, constraint types, boxed types, addresses the above aspects and answers positively through Theorem 1 to the above question. A by-product of our work is the identification of a few bugs and gaps in the ECMA Standard [4].

Notational Conventions. Beside the list operations *pop*, *top*, *length*, $\cdot$ (the operation *append* for lists), we use two other operations: for a list L, $drop(L,n)$ returns the list resulting from dropping the last n elements from L and $split(L,n)$ splits off the last n elements of L, i.e., $split(L,n)$ is the pair (L', L'') where $L' \cdot L'' = L$ and $length(L'') = n$.

The rest of the paper is organized as follows. Section 2 gives a formalization of the polymorphic CLR type system. Section 3 provides a formal specification for the statical and operational semantics of a subset of bytecode instructions relevant for generics. Section 4 develops mathematical specifications for the type-consistency checks performed by the verifier and for the statically well-typed methods accepted by the verifier. Section 5 proves that the runtime execution of well-typed methods (with generics) does not corrupt the memory. Section 6 concludes.

2 Type System

This section defines a mathematical framework for the polymorphic type system of CLR. A *type* is a value type, a reference type or a generic parameter. A *value type* is either a value class (whose objects are composite values, i.e., values composed from other values) or a primitive type. The *reference types* are the object classes (whose objects are reference objects), the interfaces, the pointer types[1] and the boxed types. For every value type T, there exists a reference type $boxed(T)$ called *boxed type*. The value of a type $boxed(T)$ is a location where a value of type T can be stored. *Only* the verifier has knowledge of the boxed types. In the bytecode, a boxed type can only be referred to as `object` or as interfaces implemented by the associated value type.

$$\begin{aligned} Type &= ValueType \cup RefType \cup GenericParam \\ ValueType &= ValueClass \cup PrimitiveType \\ RefType &= ObjClass \cup Interface \cup PointerType \cup BoxedType \end{aligned}$$

Additionally, `void` can be used only as a method return type and `Null` (the type of `null`) is used only in the bytecode verification.

The methods are identified through *method references*, i.e., elements of the universe *MRef*. The references include the *signatures* consisting of the argument types and return type. We consider only instance (including virtual) methods.

A class or an interface whose declaration is parameterized by one or more types is called *generic type*. We denote by *GenericType* the universe of generic

[1] As the pointer types have been studied in [5], their use is very limited in this paper: a pointer (evaluated to an address) can be loaded on the stack by an *Unbox* instruction and can be passed as the **this** pointer to a call through the instructions *CallVirt* and *Constrained.CallVirt*.

Table 1. Selector functions for generic types/methods

$genParamNo : Map(GenericType \cup MRef, \mathbb{N})$	number of generic parameters of a type/method
$genArg : Map(GenericType \cup MRef, List(Type \setminus PointerType))$	generic arguments of a type/method
$constr_i^T : Type \setminus (PointerType \cup GenericParam)$	i-th constraint of generic type T
$constr_i^{mref} : Type \setminus PointerType$	i-th constraint of generic method $mref$
$(var_i^{I'n})_{i=0}^{n-1} : List(\{+, -, \emptyset\})$	variances array of generic interface $I'n$

types: $GenericType \subseteq ValueClass \cup ObjClass \cup Interface$. Typically, $C'n$ gives the name of a generic type C with n type parameters. A method declared within a type, whose signature includes one or more generic parameters (not present in the type declaration itself) is called *generic method*. The class generic parameters are written in the bytecode as $!i$ whereas the method generic parameters are addressed as $!!i$. Thus, $!i$ denotes the i-th class generic parameter (numbered from left-to-right in the relevant class declaration). Similarly, $!!i$ designates the i-th method generic parameter (numbered from left-to-right in the relevant method declaration). *GenericParam* denotes the universe of generic parameters.

Every generic parameter can have an optional constraint consisting of a type. This differs slightly from [4] which allows a constraint to have more than one type. The restriction does not reduce the complexity but simplifies the exposition. The generic types and methods can be *instantiated*[2] by replacing every generic parameter with a *generic argument*. Every generic argument shall be a subtype (*when boxed*) of the type given in the corresponding constraint (see Definition 1 for the subtype relation).

The generic interfaces can be covariant or contravariant in one or more of its generic parameters. A *covariant generic parameter* is marked with "+" in the interface declaration whereas "-" is used to denote a *contravariant generic parameter*[3]. For the sake of notation, we mark with "$\emptyset$" the non-variant parameters.

Table 1 gathers the selector functions which we define to deal with generics.

Definition 1 introduces the subtype relation. The relation is defined also for *open generic types*, i.e., generic types involving generic parameters. We use "$\circ$" to denote a *substitution*. Thus, $T \circ [U_i/X_i]_{i=0}^{n-1}$ is the type T where each generic parameter X_i is substituted by the type U_i.

Definition 1 (Subtype Relation). *The* subtype relation $\sqsubseteq$ *is the least reflexive and transitive relation such that*

- *if* T_1 *is a non-generic object class which extends / implements the class / interface* T_2*, or*
- *if* T_1 *is boxed*(T) *where* T *is a non-generic value class which extends / implements the class / interface* T_2*, or*

[2] *GenericType* has *only* instantiated generic types *and not* generic types' raw names.
[3] Examples of .NET languages that support contravariance are Java 5.0 (through "wildcards" with lower bounds), Sather, Scala, OCaml.

- *if* T_1 *is boxed*(X) *where* X *is a generic parameter constrained by* T_2*, or*
- *if* T_1 *is* `Null` *and* $T_2 \in RefType$*, or*
- *if* $T_1 \in RefType$ *and* $T_2 =$ `object`*, or*
- *if* T_1 *is* $C\langle U_0, \ldots, U_{n-1}\rangle$ *where* $C'n$ *is a generic object class with the generic parameters* $(X_i)_{i=0}^{n-1}$ *which extends / implements the class / interface* T *and* T_2 *is given by* $T \circ [U_i/X_i]_{i=0}^{n-1}$*, or*
- *if* T_1 *is boxed*$(C\langle U_0, \ldots, U_{n-1}\rangle)$ *where* $C'n$ *is a generic value class with the generic parameters* $(X_i)_{i=0}^{n-1}$ *which extends / implements the class / interface* T *and* T_2 *is given by* $T \circ [U_i/X_i]_{i=0}^{n-1}$*, or*
- *if* T_1 *is* $T\langle U_0, \ldots, U_{n-1}\rangle$ *and* T_2 *is* $T\langle V_0, \ldots, V_{n-1}\rangle$ *and for every* $i = 0, n-1$ *the following conditions hold:*
  - *if* $var_i^{T'n} = \emptyset$ *or* $V_i \in ValueType$ *or* $V_i \in GenericParam$*, then* $U_i = V_i$*;*
  - *if* $var_i^{T'n} =$ `+`*, then* $U_i \sqsubseteq V_i$*;*
  - *if* $var_i^{T'n} =$ `-`*, then* $V_i \sqsubseteq U_i$*;*

then $T_1 \sqsubseteq T_2$*.*

To state Definition 2 and Definition 3, we need to define the *negation* $-(var_i)$ *of a variances array* (var_i): $-var_i$ is `+`, if $var_i =$ `-`, $-var_i$ is `-`, if $var_i =$ `+`, and $-var_i$ is $\emptyset$, if $var_i = \emptyset$.

To enforce type safety, the ECMA Standard [4] imposes, unlike [8], several requirements on the instance methods declared by a generic interface which is co-/contra-variant in at least one generic parameter. These methods shall be valid according to Definition 3. However, that definition requires the notion of *valid type with respect to a variances array* which we specify in Definition 2.

Definition 2 (Valid Type). *The predicate* ***validType*** *checks the validity of a type* T *with respect to an array* (var_i) *of variances.*

$$
\begin{array}{l}
validType(T, (var_i)) :\Longleftrightarrow \\
\quad T \notin GenericType \cap GenericParam \vee \\
\quad T \text{ is a generic parameter } !!j \text{ of the enclosing method} \vee \\
\quad T \text{ is a generic parameter } !j \text{ of the enclosing type} \wedge var_j \in \{\texttt{+}, \emptyset\} \vee \\
\quad T = C\langle U_0, \ldots, U_{n-1}\rangle \in GenericType \wedge \\
\qquad \forall k = 0, n-1 \\
\qquad\quad (var_k^{C'n} = \texttt{+} \Longrightarrow validType(U_k, (var_i))) \wedge \\
\qquad\quad (var_k^{C'n} = \texttt{-} \Longrightarrow validType(U_k, -(var_i))) \wedge \\
\qquad\quad (var_k^{C'n} = \emptyset \Longrightarrow validType(U_k, (var_i)) \wedge validType(U_k, -(var_i)))
\end{array}
$$

Closed Generic Types Not Valid? The ECMA Standard [4, Partition II, §9.7] states that $T = C\langle U_0, \ldots, U_{n-1}\rangle$ in the above definition shall refer to a "closed" generic type. This does not make a lot of sense, since in this case the definition has nothing to do with the array of variances. This remark and the experiments we have run with CLR indicate that T shall not necessarily be a closed type.

Definition 3 specifies when *a method is valid with respect to a variances array*. A method is valid if its return type "behaves covariantly" whereas its

argument types and possibly constraint types "behave contravariantly". The functions *retType*, *argTypes* and *argNo* (introduced in Table 3) are used for a method to retrieve the return type, the list of argument types and its length, respectively. Note that the argument indexed with 0 gives the **this** pointer.

Definition 3 (Valid Method). *The predicate* ***validMeth*** *checks the validity of the method mref declaration with respect to the array* (var_i) *of variances.*

$$
\begin{aligned}
&validMeth(mref, (var_i)) :\Longleftrightarrow \\
&\quad validType(retType(mref), (var_i)) \wedge \\
&\quad \forall j = 1, argNo(mref) - 1 \quad validType(argTypes(j), -(var_i)) \wedge \\
&\quad \forall j = 0, genParamNo(mref) - 1 \quad validType(constr_j^{mref}, -(var_i))
\end{aligned}
$$

The declaration of a generic interface is *valid* if all the instance methods declared by the interface and all the implemented interface types are valid with respect to the variances array of the given interface[4].

Definition 4 (Valid Interface Declaration). *The predicate* ***validDecl*** *checks the validity of the generic interface* $I'n$ *declaration.*

$$
\begin{aligned}
validDecl\,(I'n) :\Longleftrightarrow\ & \forall I'n::M \quad validMeth(I'n::M, (var_i^{I'n})) \wedge \\
& \forall J \text{ implemented by } I'n \quad validType(J, (var_i^{I'n}))
\end{aligned}
$$

The type safety proof in Section 5 takes advantage of the following lemmas. Lemma 1 shows that the subtype relationship of a given type varies *directly* with the relationship of the covariant generic parameters and *inversely* with the relationship of the contravariant generic parameters.

Lemma 1. *If the types* T, $(U_i)_{i=0}^{n-1}$, $(V_i)_{i=0}^{n-1}$ *and the array* $(var_i)_{i=0}^{n-1}$ *of variances are such that* $validType(T, (var_i)_{i=0}^{n-1})$ *and*

$$\forall i = 0, n-1((var_i = \texttt{+} \Longrightarrow V_i \sqsubseteq U_i) \wedge (var_i = \texttt{-} \Longrightarrow U_i \sqsubseteq V_i) \wedge (var_i = \emptyset \Longrightarrow V_i = U_i))$$

then $T \circ [\,V_i/!i\,]_{i=0}^{n-1} \sqsubseteq T \circ [\,U_i/!i\,]_{i=0}^{n-1}$.

Proof. By induction on the structure of the (possibly generic) type T. Definition 2 is applied. □

Lemma 2 proves that, if $T_1 \sqsubseteq T_2$, then the instantiation of the generic parameters (corresponding to an enclosing generic class and/or method) occuring in T_1 and T_2 with generic arguments satisfying the corresponding constraints preserves the subtype relation between T_1 and T_2.

[4] [4, Partition II, §9.7] is unclear. It requires that "every inherited interface declaration" shall be valid with respect to the variances array. Firstly, the interfaces are not "inherited" but implemented. Secondly, it is not about the interface "declaration" but about the interface type present in the `extends` clause of the given interface.

Lemma 2. *Let T_1 and T_2 be two types such that $T_1 \sqsubseteq T_2$. Assume T_1 and T_2 occur in the declaration of $C'n$ possibly in the declaration of a generic method $C'n::M$. Let $(U_i)_{i=0}^{n-1}$ and $(V_j)_{j=0}^{m-1}$ be generic arguments for $C'n$ and $C'n::M$, respectively, assumed to satisfy the constraints:*

$$\begin{array}{ll} boxed(U_i) \sqsubseteq constr_i^{C'n} \circ [U_i/!i]_{i=0}^{n-1}, & \text{for every } i = 0, n-1 \\ boxed(V_j) \sqsubseteq constr_j^{C'n::M} \circ ([U_i/!i]_{i=0}^{n-1} \cdot [V_j/!!j]_{j=0}^{m-1}), & \text{for every } j = 0, m-1 \end{array}$$

It then holds $T_1 \circ [U_i/!i]_{i=0}^{n-1} \cdot [V_j/!!j]_{j=0}^{m-1} \sqsubseteq T_2 \circ [U_i/!i]_{i=0}^{n-1} \cdot [V_j/!!j]_{j=0}^{m-1}$.

Proof. By induction on the structure of the (possibly generic) type T_1. Definition 1 is applied. □

Since it is possible for different methods to have identical signatures when the declaring types are instantiated, the method references have the signatures uninstantiated. Unlike the signatures, the type constraints are instantiated in our approach unless explicitly stated otherwise. To get the instantiated return type and argument types of a method reference, we define *inst* as the substitution that shall be applied to the reference:

$$inst(C::M) := [genArg(C)(i)/!i]_{i=0}^{genParamNo(C)-1} \cdot [genArg(C::M)(i)/!!i]_{i=0}^{genParamNo(C::M)-1}$$

3 Bytecode Semantics

In this section we formally define the semantics of the instructions from Table 2 by means of a small-step operational semantics modeled in ASM[5] syntax in Table 5. The static semantics is given in terms of the functions defined in Table 3 while the dynamic state of the considered bytecode language is described by the functions introduced in Table 4.

The reasons for considering only the instructions from Table 2 are the following. Adding generic types increases the complexity of $\sqsubseteq$ on which *CastClass* and *IsInstance* strongly depend. To give a flavor of the boxed types (possibly involving generic parameters) critical for handling generic arguments, we consider also the instructions *Box*, *Unbox* and *Unbox.Any*. To accommodate method calls on generic parameter values, we analyze also *Constrained.CallVirt*. The most critical feature for type safety is the generic parameter variance. As this aspect is reflected in virtual method calls, *CallVirt* and *Return* are also considered. The other CLR instructions are left out since they do not pose any problems in ensuring type safety of the generics features.

We briefly describe the instruction semantics defined in Table 5. Every instruction is executed under the assumption that the current method *meth* is instantiated, i.e., every generic parameter is replaced by the corresponding generic argument. Consequently, *inst*(*meth*) is applied to every instruction. Every time

[5] The definition of ASMs is skipped here, because ASMs can be correctly understood as pseudo-code operating over abstract (domains of) data. See their definition in [2].

Table 2. CLR instructions considered

$$
\begin{aligned}
Instr = \; & CastClass(ObjClass \cup Interface \cup ValueClass) \\
\mid \; & IsInstance(ObjClass \cup Interface \cup ValueClass) \\
\mid \; & Box(Type) \\
\mid \; & Unbox(ValueType) \\
\mid \; & Unbox.Any(Type) \\
\mid \; & CallVirt(Type, MRef) \\
\mid \; & Constrained(GenericParam).CallVirt(Type, MRef) \\
\mid \; & Return
\end{aligned}
$$

Table 3. Static functions

$code$	$: Map(MRef, List(Instr))$	list of instructions of a method body
$argTypes$	$: Map(MRef, List(Type))$	argument types of a method
$argNo$	$: Map(MRef, \mathbb{N})$	number of arguments of a method
$retType$	$: Map(MRef, Type)$	return type of a method
$lookup$	$: Map(Type \times MRef, MRef)$	*dynamic binding* function

Table 4. Dynamic functions

$memVal$	$: Map(Address \times Type, Value)$	memory function
$meth$	$: MRef$	current method
pc	$: Pc$	current program counter of $meth$
$argVal$	$: Map(\mathbb{N}, Value)$	argument values of $meth$
$evalStack$	$: List(Value)$	current evaluation stack of $meth$
$actualTypeOf$	$: Map(ObjRef, Type)$	runtime type of an object reference
$addressOf$	$: Map(ObjRef, Address)$	address of value type in a boxed object

an exception occurs, control is passed to the exception handling mechanism defined in [7] which preserves type safety as proved in [5]. Since we do not consider exceptions here, we do not model this control switching either.

The instruction $CastClass(C)$ checks if the topmost value of the *evalStack* is of type C. If not, an exception is thrown. The instruction $IsInstance(C)$ pops from the *evalStack* a reference to an (possibly boxed) object. If the object is not an instance of C, `null` is pushed on the *evalStack*. The $Box(T)$ instruction (where T can also be a generic parameter) turns a boxable value into its boxed form. Applied to a value type, the instruction loads a boxed object created through the macro *NewBox* defined below on the *evalStack*.

$$
\begin{aligned}
&\textbf{let } r = NewBox(val, T) \textbf{ in } P \equiv \textbf{let } r = new(ObjRef) \textbf{ and } adr = new(Address, T) \textbf{ in} \\
&\qquad\qquad \textsc{WriteMem}(adr, T, val) \\
&\qquad\qquad addressOf\ (r) \quad := adr \\
&\qquad\qquad actualTypeOf\ (r) := T \\
&\qquad \textbf{seq } P
\end{aligned}
$$

Table 5. Execution of the bytecode instructions

$$
\begin{array}{l}
\textbf{match}\ \mathit{code}(\mathit{meth})(\mathit{pc}) \\
\quad \mathit{CastClass}(C) \circ \mathit{inst}(\mathit{meth}) \rightarrow \\
\qquad \textbf{let}\ r = \mathit{top}(\mathit{evalStack})\ \textbf{in} \\
\qquad\quad \textbf{if}\ r = \texttt{null} \vee \mathit{actualTypeOf}\ (r) \sqsubseteq C \circ \mathit{inst}(\mathit{meth})\ \textbf{then} \\
\qquad\qquad \mathit{pc} := \mathit{pc} + 1 \\
\\
\quad \mathit{IsInstance}(C) \circ \mathit{inst}(\mathit{meth}) \rightarrow\ \textbf{let}\ (\mathit{evalStack}', [r]) = \mathit{split}(\mathit{evalStack}, 1)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pc} := \mathit{pc} + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ \mathit{actualTypeOf}\ (r) \not\sqsubseteq C \circ \mathit{inst}(\mathit{meth})\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{evalStack} := \mathit{evalStack}' \cdot [\texttt{null}] \\
\\
\quad \mathit{Box}(T) \circ \mathit{inst}(\mathit{meth}) \qquad\qquad \rightarrow\ \mathit{pc} := \mathit{pc} + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ T \circ \mathit{inst}(\mathit{meth}) \in \mathit{ValueType}\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ (\mathit{evalStack}', [\mathit{val}]) = \mathit{split}(\mathit{evalStack}, 1)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ r = \mathit{NewBox}(\mathit{val}, T \circ \mathit{inst}(\mathit{meth}))\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{evalStack} := \mathit{evalStack}' \cdot [r] \\
\\
\quad \mathit{Unbox}(T) \circ \mathit{inst}(\mathit{meth}) \rightarrow\ \textbf{let}\ (\mathit{evalStack}', [r]) = \mathit{split}(\mathit{evalStack}, 1)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ r \neq \texttt{null} \wedge \mathit{actualTypeOf}\ (r) = T \circ \mathit{inst}(\mathit{meth})\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{evalStack} := \mathit{evalStack}' \cdot [\mathit{addressOf}\ (r)] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pc} := \mathit{pc} + 1 \\
\\
\quad \mathit{Unbox.Any}(T) \circ \mathit{inst}(\mathit{meth}) \rightarrow \\
\qquad \textbf{let}\ (\mathit{evalStack}', [r]) = \mathit{split}(\mathit{evalStack}, 1)\ \textbf{in} \\
\qquad\quad \textbf{if}\ T \circ \mathit{inst}(\mathit{meth}) \in \mathit{ValueType}\ \textbf{then} \\
\qquad\qquad \mathit{evalStack} := \mathit{evalStack}' \cdot [\mathit{memVal}(\mathit{addressOf}\ (r), T \circ \mathit{inst}(\mathit{meth}))] \\
\qquad\qquad \mathit{pc} := \mathit{pc} + 1 \\
\qquad\quad \textbf{elseif}\ r = \texttt{null} \vee \mathit{actualTypeOf}\ (r) \sqsubseteq T \circ \mathit{inst}(\mathit{meth})\ \textbf{then}\ \mathit{pc} := \mathit{pc} + 1 \\
\\
\quad \mathit{CallVirt}(\_, C{::}M) \circ \mathit{inst}(\mathit{meth}) \rightarrow \\
\qquad \textbf{let}\ (\mathit{evalStack}', [r] \cdot \mathit{vals}) = \mathit{split}(\mathit{evalStack}, \mathit{argNo}(C{::}M))\ \textbf{in} \\
\qquad\quad \mathit{evalStack} := \mathit{evalStack}' \\
\qquad\quad \textsc{VirtCall}(r, C{::}M \circ \mathit{inst}(\mathit{meth}), \mathit{vals}) \\
\\
\quad \mathit{Constrained}(T). \\
\quad \mathit{CallVirt}(\_, C{::}M) \circ \mathit{inst}(\mathit{meth}) \rightarrow \\
\qquad \textbf{let}\ (\mathit{evalStack}', [\mathit{adr}] \cdot \mathit{vals}) = \mathit{split}(\mathit{evalStack}, \mathit{argNo}(C{::}M))\ \textbf{in} \\
\qquad\quad \mathit{evalStack} := \mathit{evalStack}' \\
\qquad\quad \textbf{if}\ T \circ \mathit{inst}(\mathit{meth}) \in \mathit{RefType}\ \textbf{then} \\
\qquad\qquad \textbf{let}\ r = \mathit{memVal}(\mathit{adr}, T \circ \mathit{inst}(\mathit{meth}))\ \textbf{in} \\
\qquad\qquad\quad \textsc{VirtCall}(r, C{::}M \circ \mathit{inst}(\mathit{meth}), \mathit{vals}) \\
\qquad\quad \textbf{elseif}\ T \circ \mathit{inst}(\mathit{meth}) \in \mathit{ValueClass} \wedge \\
\qquad\qquad T{::}M \circ \mathit{inst}(\mathit{meth})\ \text{implements}\ C{::}M \circ \mathit{inst}(\mathit{meth})\ \textbf{then} \\
\qquad\qquad \textsc{Invoke}(T{::}M \circ \mathit{inst}(\mathit{meth}), [\mathit{adr}] \cdot \mathit{vals}) \\
\qquad\quad \textbf{else let}\ r = \mathit{NewBox}(\mathit{memVal}(\mathit{adr}, T \circ \mathit{inst}(\mathit{meth})), T \circ \mathit{inst}(\mathit{meth}))\ \textbf{in} \\
\qquad\qquad \textsc{VirtCall}(r, C{::}M \circ \mathit{inst}(\mathit{meth}), \mathit{vals}) \\
\\
\quad \mathit{Return} \circ \mathit{inst}(\mathit{meth}) \rightarrow\ \textbf{if}\ \mathit{retType}(\mathit{meth}) = \texttt{void}\ \textbf{then}\ \textsc{Result}([\,]) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \textsc{Result}([\mathit{top}(\mathit{evalStack})])
\end{array}
$$

NewBox(val, T) creates a fresh object reference and allocates an address where *val* of type T is stored through WriteMem. The definition of WriteMem (which can be found in [6]) is beyond the scope of this paper.

The *Unbox* instruction takes a reference to a boxed object from the *evalStack* and loads the address of the value embedded into the boxed object. An exception is thrown if the object is not a boxed object or the value type of the value in the box does not match the instantiation of the type operand of the instruction. Unlike *Unbox*, for value types, the *Unbox.Any* instruction leaves the value, not the address of the value, on the *evalStack*. Moreover, the type embedded in *Unbox* can only represent value types and instantiations of generic value types. The function *memVal* (whose definition is given in [6]) is used in *Unbox.Any* to compute the value of a given type stored at a given address. For reference types, *Unbox.Any* has the same effect as *CastClass*.

To specify the virtual method calls implied by the instructions *CallVirt* and *Constrained.CallVirt*, we need to define the *lookup* function.

Definition 5 (Lookup). *Given a type and a (possibly generic) method reference, the function lookup* : *Map*$(Type \times MRef, MRef)$ *determines the method to be invoked at runtime when the given method is called on an object whose runtime type is the given type.*

$$
\begin{array}{l}
lookup(C, D{::}M\langle T_0, \ldots, T_{n-1}\rangle) := \\
\quad \textbf{if } C{::}M \in MRef \wedge \\
\qquad (D \in ObjClass \Longrightarrow C{::}M \text{ overrides } D{::}M) \wedge \\
\qquad (D \in Interface \setminus GenericType \Longrightarrow C{::}M \text{ implements } D{::}M) \wedge \\
\qquad (D = I\langle U_0, \ldots, U_{m-1}\rangle \in Interface \cap GenericType \Longrightarrow \\
\qquad\quad C{::}M \text{ implements } I\langle V_0, \ldots, V_{m-1}\rangle{::}M \wedge \\
\qquad\qquad \forall\, j = 0, m-1\,((var_j^{I\,'m} = \texttt{+} \Longrightarrow V_j \sqsubseteq U_j) \wedge \\
\qquad\qquad\qquad\qquad (var_j^{I\,'m} = \texttt{-} \Longrightarrow U_j \sqsubseteq V_j) \wedge \\
\qquad\qquad\qquad\qquad (var_j^{I\,'m} = \emptyset \Longrightarrow U_j = V_j))) \\
\quad \textbf{then } C{::}M\langle T_0, \ldots, T_{n-1}\rangle \\
\quad \textbf{elseif } C = \texttt{object} \textbf{ then } undef \\
\quad \textbf{else } lookup(C', D{::}M\langle T_0, \ldots, T_{n-1}\rangle) \text{ where } C' \text{ is the direct base class of } C
\end{array}
$$

The ECMA Standard [4] does not specify what is the effect of adding generic parameter variance on the dynamical method lookup. As one can see in Definition 5, the definition of *lookup* becomes more complex: *lookup*$(\_, D{::}M)$ shall not necessarily be a method which overrides or implements $D{::}M$.

The instruction *CallVirt*$(T, C{::}M)$ calls the virtual method $C{::}M$ whose (possibly open generic) return type is T. It pops the necessary number of arguments from the *evalStack*. Based on the type of the **this** pointer, it looks up the method to be invoked (through the Invoke macro defined in Table 6) with the popped arguments dynamically by means of the *lookup* function. *lookup* is applied to $C{::}M \circ inst(meth)$, i.e., the method $C{::}M$ where *only* the generic parameters present in C or possibly in the generic argument list of M are replaced by the generic arguments indicated in $inst(meth)$.

$$
\begin{array}{l}
\textsc{VirtCall}(r, C{::}M, vals) \equiv \textbf{let } D{::}M = lookup(actualTypeOf\ (r), C{::}M) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad \textbf{if } r \neq \texttt{null} \textbf{ then } \textsc{Invoke}(D{::}M, [r] \cdot vals)
\end{array}
$$

Table 6. Invoking a method and returning from a method

INVOKE(C::M, *args*) ≡	RESULT(*vals*) ≡
PUSHFRAME	POPFRAME
seq	**seq**
pc := 0	*pc* := *pc* + 1
evalStack := []	*evalStack* := *evalStack* · *vals*
meth := C::M	
SETARG(C::M, *args*)	

The instruction *Constrained*(T).*CallVirt*(S, C::M) calls the virtual method C::M (whose return type is S) on a value of a generic parameter T. It pops the necessary number of arguments from the *evalStack*. The first value is expected to be a pointer (evaluated to an address) *adr*. If T is a reference type, then *adr* is dereferenced and passed as the **this** pointer to VIRTCALL. If T is a value type and T implements C::M, then *adr* is passed as the **this** pointer to the method implemented by T which is then called with INVOKE. If T is a value type which does not implement C::M, then *adr* is dereferenced, boxed, and passed as the **this** pointer to a virtual call of C::M. Normally, the above outlined transformation of the **this** pointer is performed at compile time, depending on the type of *adr* and the method being called. However, such a transformation would not be possible when the type of the **this** pointer is a generic type (unknown at compile time). Thus, the prefix *Constrained* allows .NET compilers to make a call to a virtual function in an *uniform way* independent of whether the **this** pointer is a value type or reference type.

The *Return* instruction returns from the current method *meth* by means of the RESULT macro which we define in Table 6. If the return type of *meth* is not `void`, *evalStack* shall contain a value to be returned through RESULT.

The macros PUSHFRAME and POPFRAME in Table 6 are used to push a new frame and to pop the current frame, respectively. The macro SET(C::M,*args*) sets the arguments of C::M, i.e., the *argVal* function, to the values *args*.

As stated at the end of Section 2, the method references have the signatures uninstantiated. Therefore, for example, the return type (specified in the signature) of a method which overrides/implements another method shall not necessarily match the return type (specified in the signature) of the overridden/implemented method. The conditions that shall actually be satisfied when a generic method C::M overrides/implements D::M are listed below:

(at) for every $i = 1, argNo(C::M) - 1$,

$$argTypes(C::M)(i) \circ inst(C::M) = argTypes(D::M)(i) \circ inst(D::M)$$

(rt) $retType(C::M) \circ inst(C::M) = retType(D::M) \circ inst(D::M)$

(ct) for every $j = 0, genParamNo(D::M) - 1$,

$$constr_j^{C::M} \text{ is not defined or } constr_j^{D::M} \sqsubseteq constr_j^{C::M}$$

No More Restrictive? Concerning (**ct**), [4, Partition II,§9.9] states that any constraint type specified by the overriding method shall be "no more restrictive" than the corresponding constraint type specified in the overridden method. However, this does not match the Microsoft implementation [1]. It seems that the Microsoft verifier checks whether one of the following conditions is satisfied: either the constraint type in the overriding method is not defined, i.e., as it would have been `object`, or the constraint types (assumed to be instantiated) coincide.

4 Bytecode Verification

The bytecode verification is performed on a *per-method* basis. The verifier simulates the bytecode execution. It attempts to associate a *stack state* $evalStackT$ with every instruction. The stack state is a list of types which specifies the number of values on the *evalStack* at that point in the code and for each slot of the *evalStack* a required type that shall be present in that slot. Before simulating the execution of an instruction, the verifier performs several type-consistency checks specified by means of the predicate *check* defined in Table 7. Its definition follows the specification of the ECMA Standard [4, Partition III]. The stack state of an instruction is constrained by referring to the stack states of the next instruction. Table 8 defines the function *succ* which, given an instruction and a stack state, computes the stack states of the next instruction.

To deal with stack states, we introduce the relations $\sqsubseteq_{suf}$ and $\sqsubseteq_{len}$. If L' and L'' are two lists of types of lengths m and n, respectively, then

$$L' \sqsubseteq_{suf} L'' :\Longleftrightarrow m \geq n \text{ and } L'(m-n+i) \sqsubseteq L''(i) \text{ for every } i = 0, n-1.$$

$$L' \sqsubseteq_{len} L'' :\Longleftrightarrow m = n \text{ and } L'(i) \sqsubseteq L''(i) \text{ for every } i = 0, m-1.$$

To shorten the specification of *check* and *succ*, we use the following notation:

$$void(T) := \textbf{if } T = \texttt{void} \textbf{ then } [\,] \textbf{ else } [T]$$

Table 7. Type-consistency checks performed by the verifier

$check(meth, pos, evalStackT) :\Longleftrightarrow$
match $code(meth)(pos)$

$CastClass(\_)$	$\rightarrow$	$evalStackT \sqsubseteq_{suf} [\texttt{object}]$
$IsInstance(\_)$	$\rightarrow$	$evalStackT \sqsubseteq_{suf} [\texttt{object}]$
$Box(T)$	$\rightarrow$	$evalStackT \sqsubseteq_{suf} [T]$
$Unbox(\_)$	$\rightarrow$	$evalStackT \sqsubseteq_{suf} [\texttt{object}]$
$Unbox.Any(\_)$	$\rightarrow$	$evalStackT \sqsubseteq_{suf} [\texttt{object}]$
$CallVirt(\_, C::M)$	$\rightarrow$	$evalStackT \sqsubseteq_{suf} argTypes(C::M) \circ inst(C::M)$
$Constrained(T).$		
$CallVirt(\_, C::M)$	$\rightarrow$	$boxed(T) \sqsubseteq C \wedge evalStackT \sqsubseteq_{suf}$ $[T\&] \cdot [argTypes(C::M)(i)]_{i=1}^{argNo(C::M)-1} \circ inst(C::M)$
$Return$	$\rightarrow$	$evalStackT \sqsubseteq_{len} void(retType(meth))$

Table 8. Determining successor stack states

$succ(meth, pos, evalStackT) :=$
match $code(meth)(pos)$

$CastClass(C)$	$\rightarrow$	$\{pop(evalStackT) \cdot [C]\}$
$IsInstance(C)$	$\rightarrow$	$\{pop(evalStackT) \cdot [C], pop(evalStackT) \cdot [\texttt{Null}]\}$
$Box(T)$	$\rightarrow$	**if** $T \in RefType$ **then** $\{pop(evalStackT) \cdot [T]\}$ **else** $\{pop(evalStackT) \cdot [boxed(T)]\}$
$Unbox(T)$	$\rightarrow$	$\{pop(evalStackT) \cdot [T\&]\}$
$Unbox.Any(T)$	$\rightarrow$	$\{pop(evalStackT) \cdot [T]\}$
$CallVirt(T, C::M)$	$\rightarrow$	$\{drop(evalStackT, argNo(C::M)) \cdot void(T \circ inst(C::M))\}$
$Constrained(\_)$.		
$CallVirt(T, C::M)$	$\rightarrow$	$\{drop(evalStackT, argNo(C::M)) \cdot void(T \circ inst(C::M))\}$
$Return$	$\rightarrow$	$\emptyset$

Since we have no definition of a particular bytecode verifier, we need a characterization of the type properties of the bytecode that is accepted by the verifier. This leads us to Definition 6. A method is *well-typed* if the verifier succeeds to compute a valid stack state for every instruction of the method. Definition 6 makes this precise: a method is well-typed if there exists a stack state family that satisfies an initial condition, the type-consistency checks and the relations dictated by a top-down pass through the bytecode and by the rules for merging stack states specified in [4, Partition III,§1.8.1.3].

Definition 6 (Well-typed Method). *A method mref is* well-typed *if there exists a family* $(evalStackT_i)_i$ *of stack states satisfying the following conditions:*

(wt1) $evalStackT_0 = [\,]$.

(wt2) $check(mref, pos, evalStackT_{pos})$ *holds for every position pos in mref.*

(wt3) *If* $evalStackT' \in succ(mref, pos, evalStackT_{pos})$, *then* $evalStackT' \sqsubseteq_{len} evalStackT_{pos+1}$.

5 Type Safety

We prove that the bytecode with generics is type safe. As the bytecode verifier statically checks the type safety of the bytecode, we need to show that the verifier is sound. That means that if the verifier succeeds to compute a valid stack state for every instruction of a method, i.e., the method is well-typed according to Definition 6, then several type safety properties are ensured to hold at runtime. For example, the *evalStack* shall have at runtime values of the types assigned in the stack state and the same length as the stack state. Furthermore, the generic arguments of instantiated generic types and methods shall satisfy the corresponding constraints.

For this section, we assume the following: (1) every method is well-typed; (2) every generic interface declaration is valid; (3) the generic arguments of all

the generic types and method references satisfy at verification time the corresponding constraints. In particular, this means that every generic argument used in the *inst* functions satisfies the corresponding constraints.

The following two lemmas establish relations between the argument types, respectively the return type of the method called at compile time, i.e., the method embedded in a *CallVirt*, and the argument types, respectively the return type of the method that is determined through a lookup and is invoked at runtime.

Lemma 3. *If* $D::M = lookup(T, C::M)$*, then* $T \sqsubseteq argTypes(D::M)(0)$ *and for every* $i = 1, argNo(C::M) - 1$

$$argTypes(C::M)(i) \circ inst(C::M) \sqsubseteq argTypes(D::M)(i) \circ inst(D::M)$$

Proof. By Definition 3, Definition 4, Definition 5, Lemma 1, Lemma 2, (**at**). □

Lemma 4. *If* $D::M = lookup(T, C::M)$ *holds for some type* T*, then*

$$retType(D::M) \circ inst(D::M) \sqsubseteq retType(C::M) \circ inst(C::M)$$

Proof. By Definition 3, Definition 4, Definition 5, Lemma 1, Lemma 2, (**rt**). □

Lemma 5 shows that a generic method determined through a lookup has the same generic arguments as the original method and the generic arguments satisfy the corresponding constraints.

Lemma 5. *If* $D::M$ *and* $C::M$ *are generic methods and* T *is a type such that* $D::M = lookup(T, C::M)$ *and* $boxed(genArg(C::M)(i)) \sqsubseteq constr_i^{C::M}$ *for every* $i = 0, genParamNo(C::M) - 1$*, then* $genParamNo(D::M) = genParamNo(C::M)$ *and for every* $i = 0, genParamNo(C::M) - 1$

$$boxed(genArg(C::M)(i)) = boxed(genArg(D::M)(i)) \sqsubseteq constr_i^{D::M}$$

Proof. By Definition 3, Definition 4, Definition 5, Lemma 1, Lemma 2, (**ct**). □

We assume that the generic arguments of a (instantiated) generic type or generic method satisfy at verification time the corresponding constraints. As the generic arguments might be open generic types, the following question appears: *Do they satisfy the constraints also after the runtime instantiation?* The following proposition answers positively to this question.

Proposition 1 (Preserving Constraints). *The instantiated (not necessarily closed) type* $C\langle T_0, \ldots, T_{n-1}\rangle$ *occurs in the declaration of a generic type* $D'm$ *possibly in the declaration of a generic method* $D'm::M$*. Assume that* $(T_i)_{i=0}^{n-1}$ *satisfy the constraints of* $C'n$*. If* $D'm$ *and* $D'm::M$ *are instantiated at runtime with the generic arguments* $(U_j)_{j=0}^{m-1}$ *and* $(V_k)_{k=0}^{p-1}$ *(which are assumed to satisfy the constraints of* $D'm$ *and* $D'm::M$*, respectively), then* $(T_i \circ \rho)_{i=0}^{n-1}$ *satisfy the constraints of* $C'n$ *where* ρ *is the substitution* $\rho = [U_j/!j]_{j=0}^{m-1} \cdot [V_k/!!k]_{k=0}^{p-1}$ *defined in the context of* $D'm$ *and* $D'm::M$ *declarations. A similar result holds also for a referenced generic method, i.e.,* $C::M\langle T_0, \ldots, T_{n-1}\rangle$ *instead of* $C\langle T_0, \ldots, T_{n-1}\rangle$*.*

Proof. By Lemma 2. □

The typing judgment $\vdash val : T$ is defined as follows. Thus, $\vdash val : T$ holds if at least one of the following holds: (1) $T \in RefType \setminus (PointerType \cup BoxedType)$ and either val is `null` or $actualTypeOf\,(val) \sqsubseteq T$; (2) $T = S\&$ and $\exists r \in ObjRef$ such that $addressOf\,(r) = val$ and $actualTypeOf\,(r) = S$; (3) $boxed(S) \sqsubseteq T$ where $S \in ValueType$ and $actualTypeOf\,(val) = S$. (4) the CLR type associated (see [4, Partition III, §1.1]) to val is a value type which is a subtype of T.

We now extend the soundness proof done in [5] for the CLR without generics to the CLR with generics. The theorem proved in [5] guarantees that several type-safety invariants hold at runtime for well-typed methods without generics. We consider here only the invariants which are affected upon adding generics. Additionally, an invariant (**constr**) for generic methods is considered. The invariant (**stack1**) guarantees that *evalStack* has the same length as the assigned stack state. The invariant (**stack2**) ensures that the values on the *evalStack* are of the types assigned in the stack state. By (**arg**), we have that the arguments contain values of the declared types. The invariant (**constr**) ensures that the generic arguments of a generic method satisfy the declared constraints.

Theorem 1 (Type Safety). *The following invariants are satisfied at runtime for the current method meth:*

(stack1) $length(evalStack) = length(evalStackT_{pc})$.

(stack2) $\vdash evalStack(i) : evalStackT_{pc}(i) \circ inst(meth)$, *for every* $i = 0, length(evalStack) - 1$.

(arg) $\vdash argVal(0) : argTypes(meth)(0)$. *If meth takes at least two arguments,* $\vdash argVal(i) : argTypes(meth)(i) \circ inst(meth)$, *for every* $i = 1, argNo(meth) - 1$.

(constr) *If meth is generic,* $boxed(genArg(meth)(i)) \sqsubseteq constr_i^{meth}$, *for every* $i = 0, genParamNo(meth) - 1$.

Proof. The proof is by induction on the run of the model for the operational semantics. The invariants obviously hold in the initial state of the virtual machine, i.e., for the `entrypoint`. Due to the lack of space, we consider here only two critical cases for virtual method calls and method returns.

Case 1. $code(meth)(pc) = CallVirt(T, C::M) \circ inst(meth)$: Since *meth* is well-typed, we get $check(meth, pc, evalStackT_{pc})$ from Definition 6 (**wt2**). According to the definition of *check* in Table 7, $evalStackT_{pc} \sqsubseteq_{suf} argTypes(C::M) \circ inst(C::M)$. By (**constr**) and Lemma 2, we have[6]

$$evalStackT_{pc} \circ inst(meth) \sqsubseteq_{suf} (argTypes(C::M) \circ inst(C::M)) \circ inst(meth) \quad (1)$$

By (1) and by the induction hypothesis – that is, by the invariants (**stack1**) and (**stack2**) – there exists a list of values L, two lists of types L' and L'' and the values $(val_i)_{i=0}^{argNo(C::M)-1}$ such that: $evalStack = L \cdot [val_i]_{i=0}^{argNo(C::M)-1}$,

$$\begin{array}{l} evalStackT_{pc} = L' \cdot L'', length(L'') = argNo(C::M), \text{ for every} \\ i = 0, length(evalStack) - argNo(C::M) - 1 \quad \vdash L(i) : L'(i) \circ inst(meth) \end{array} \quad (2)$$

[6] Note that $evalStackT_{pc}$ might involve generic parameters.

$$\text{for every } i = 0, argNo(C::M) - 1 \quad \vdash val_i : L''(i) \circ inst(meth) \tag{3}$$

By (1), (2) and (3), we have for every $i = 0, argNo(C::M) - 1$

$$\vdash val_i : (argTypes(C::M)(i) \circ inst(C::M)) \circ inst(meth) \tag{4}$$

When the instruction $CallVirt(T, C::M) \circ inst(meth)$ is executed, the macro VirtCall$(val_0, C::M \circ inst(meth), [val_i]_{i=1}^{argNo(C::M)-1})$ is invoked. Assuming that val_0 is not `null`, Invoke does the following for the next current method, i.e., $D::M = lookup(actualTypeOf\ (val_0), C::M \circ inst(meth))$:

- sets the *pc* to 0,
- sets the *evalStack* to [] (this, the above initialization of the *pc* and Definition 6 (**wt1**) ensure that the invariants (**stack1**) and (**stack2**) hold also upon entering $D::M$)
- sets through SetArg, the arguments $(argVal(j))_{j=0}^{argNo(D::M)-1}$ (corresponding to $D::M$) to $(val_i)_{i=0}^{argNo(C::M)-1}$. Note that we have $argNo(C::M) = argNo(D::M)$.

It remains to prove that the invariants (**arg**) and (**constr**) hold also for $D::M$. By Lemma 3, we have

$$actualTypeOf\ (val_0) \sqsubseteq argTypes(D::M)(0) \tag{5}$$

and the following relations for every $i = 1, argNo(C::M) - 1$:

$$\begin{array}{l}(argTypes(C::M)(i) \circ inst(C::M)) \circ inst(meth) = \\ argTypes(C::M \circ inst(meth))(i) \circ inst(C::M \circ inst(meth)) \sqsubseteq \\ argTypes(D::M)(i) \circ inst(D::M)\end{array} \tag{6}$$

The invariant (**arg**) for the first argument is ensured by (5) and the relation $argTypes(D::M)(0) \circ inst(D::M) = argTypes(D::M)(0)$ (the type of the **this** pointer is instantiated in contrast with the other argument types). For the other arguments, (**arg**) follows from (4) and (6).

To prove (**constr**), we assume that $D::M$ is a generic method. This implies that also the method $C::M$ is generic. By Lemma 5, we get $genParamNo(D::M) = genParamNo(C::M)$ and for every $i = 0, genParamNo(C::M) - 1$:

$$boxed(genArg(C::M \circ inst(meth))(i)) = boxed(genArg(D::M)(i)) \sqsubseteq constr_i^{D::M}$$

Proposition 1 applied to $C::M$ and the above relations imply (**constr**).

Case 2. $code(meth)(pc) = Return \circ inst(meth)$: Since the current method *meth* is well-typed, Definition 6 (**wt2**) and the definition of *check* in Table 7 imply $evalStackT_{pc} \sqsubseteq_{len} void(retType(meth))$.

Let $T' = retType(meth)$. If T' is `void`, then $evalStackT_{pc}$ shall be []. If T' is not `void`, then $evalStackT_{pc} = [T' \circ inst(meth)]$. Note that T' might be an open generic type. The induction hypothesis – that is, the invariants (**stack1**)

and (**stack2**) applied to *meth* – implies that *evalStack* is [ ] if the return type of *meth* is `void` and $evalStack = [val]$ where $\vdash val : T' \circ inst(meth)$ if the return type of *meth* is not `void`. Let $D::M$ be the reference of *meth*. Accordingly,

$$\vdash val : T' \circ inst(D::M) \tag{7}$$

If $D::M$ is the `entrypoint`, there is nothing to prove. Otherwise, let $E::M'$ be the method which invoked $D::M$ through an instruction $CallVirt(T'', C::M)$ (the case when the call is through a *Constrained.CallVirt* instruction is similar). We need to show that the invariants (**stack1**) and (**stack2**) hold for $E::M'$ after $D::M$ returns.

When the instruction $Return \circ inst(D::M)$ is executed, the frame of $D::M$ is popped off by RESULT, $E::M'$ becomes the current method *meth*, the *pc* of $E::M'$ is incremented by 1 and *val* is pushed on the *evalStack* of $E::M'$.

The invariants held before VIRTCALL selected $D::M$ through a lookup applied to $C::M$. Similarly as in Case 1, there exists the lists of types L' and L'' that satisfy the relations (2). By Definition 6 (**wt3**) and definition of *succ* for *CallVirt*, we have: $drop(evalStackT_{pc}, argNo(C::M)) \cdot void(T'' \circ inst(C::M)) = L' \cdot void(T'' \circ inst(C::M)) \sqsubseteq_{len} evalStackT_{pc+1}$. By this, the invariant (**constr**) for *meth* and Lemma 2, we get

$$(L' \cdot void(T'' \circ inst(C::M))) \circ inst(meth) \sqsubseteq_{len} evalStackT_{pc+1} \circ inst(meth) \tag{8}$$

If T'' is `void`, Lemma 4 implies that also T' is `void`. (**stack1**) and (**stack2**) follow then from (8) and (2). If T'' is not `void`, by Lemma 4 we get $T' \circ inst(D::M) \sqsubseteq T'' \circ inst(C::M)$. This relation, (**constr**) and Lemma 2 imply $(T' \circ inst(D::M)) \circ inst(meth) \sqsubseteq (T'' \circ inst(C::M)) \circ inst(meth)$. This, together with (8), (2) and (7) guarantees the invariants (**stack1**) and (**stack2**). □

6 Conclusion

We have provided a mathematical specification for the CLR generics design via a type system and a model for the semantics of a subset of bytecode instructions with generics. We have formalized the type-consistency tests checked for the subset by the CLR bytecode verifier. Finally, we have proved that adding generics maintains the type safety of the CLR.

Acknowledgment. The author gratefully acknowledges the four reviewers, Viktor Schuppan and Horatiu Jula for their valuable comments and suggestions.

References

1. Microsoft .NET Framework 2.0. `http://msdn.microsoft.com/netframework/`.
2. Egon Börger and Robert F. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer–Verlag, 2003.
3. W. R. Cook. A Proposal for Making Eiffel Type-Safe. *Comput. J.*, 32(4):305–311, 1989.

4. Common Language Infrastructure (CLI) – Standard ECMA-335, 2002.
5. Nicu G. Fruja. The Soundness of the .NET CLR Bytecode Verifier. submitted.
6. Nicu G. Fruja. A Modular Design for the .NET CLR Architecture. In A. Slissenko D. Beauquier and E. Börger, editors, *12th International Workshop on Abstract State Machines, ASM 2005, Paris, France*, pages 175–199, March 2005.
7. Nicu G. Fruja and Egon Börger. Analysis of the .NET CLR Exception Handling. In Vaclav Skala and Piotr Nienaltowski, editors, *3rd International Conference on .NET Technologies, .NET 2005, Pilsen, Czech Republic*, pages 65–75, June 2005.
8. Mark Howard, Éric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stapf, Karine Arnout, and Markus Keller. Type-safe covariance: Competent compilers can catch all catcalls. Draft at `http://se.ethz.ch/~meyer/`, April 2003.
9. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
10. A. Jeffrey. Generic Java type inference is unsound. The Types Forum, 2001.
11. Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation, PLDI 2001, Snowbird, Utah, United States*, pages 1–12, May 2001.
12. Mirko Viroli and Antonio Natali. Parametric Polymorphism in Java: an Approach to Translation based on Reflective Features. *SIGPLAN Not.*, 35(10):146–165, 2000.
13. Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of Generics for the .NET Common Language Runtime. In *31st ACM Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 39 – 51, January 2004.