ASM Semantics for C♯ 2.0

Horatiu V. Jula

Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
julah@inf.ethz.ch

Abstract The Abstract State Machines (ASMs) theory has been applied to formalize the semantics of the C♯ programming language in a rigorous mathematical manner. We have extended the C♯ ASM model in order to handle C♯ 2.0 specific features like *generics, anonymous methods and iterator blocks*. We discovered that the existing *operational model* (the dynamic semantics) can be reused entirely. It is not altered after the integration of the new constructions. The *static semantics* (e.g., subtype relation and type constraints) suffered some important modifications.

1 Introduction

In this paper we present the extension of the C♯ ASM model [1] with the C♯ 2.0 specific constructions, namely *generics, anonymous methods and iterator blocks*. We propose a robust abstract definition of the C♯ 2.0 programs' semantics, in order to provide a platform-independent interpreter view of the C♯ 2.0 language. We specify the static and the dynamic parts of the semantics separately, due to the ASM classification of abstract states into a *static* and a *dynamic* part [5]. The *dynamic semantics* of the language are captured by ASM operational rules which describe the run-time effect of program's execution on the abstract state of the program. The *static semantics* are a declarative description of the relevant static language features (e.g., grammar, subtype relation, type constraints, rules for definite assignment, syntactical translations).

The new C♯ constructions are very helpful. The generics increase the code flexibility and reusability. The other two features consist of syntactical sugar that ease the coding and also make the code more readable. We formalize the C♯ semantics in order to prove some properties of the C♯ programs, like *type safety*. We choose ASMs because they lead to a very precise, expressive and robust formalization.

Formalizations of the Java and C++ semantics have been developed already. For instance, a formal definition of the Java Semantics has been defined in an operational style using the Typol formalism [7]. One uses *natural semantics* to specify the object oriented and inheritance features of Java and *structural operational semantics* to describe concurrency. This formalism enables us to get a modular Java specification. Other Java formal semantics have been developed using ASMs [4]. They are quite similar to the C♯ ASM semantics. Formal operational semantics based on *evolving algebras* [6] have been developed for the

C++ language [8]. The formalizations that are based on the ASMs theory are very precise and have a very high level of abstraction. Therefore, they are more readable and comprehensive and can be easily used for proving some properties of the systems they describe, like soundness or completeness.

The paper is organized as follows: the next two sections illustrate how the *generics* and respectively the *anonymous methods* modify the static semantics (i.e., the grammar, the subtype relation, the type constraints and the definite assignment rules) and the dynamic (operational) semantics (i.e., the execution rules) by altering the existing semantics and/or adding new semantics. The fourth section describes the implementation of the *iterator blocks*, which is made through syntactic sugar; it implies only a few minor modifications of the static semantics, while the dynamic semantics remain unaltered, since the implementation is made by using syntactical transformations.

2 Generics

2.1 Background

The *generics* are special constructions that incorporate *type parameters*, which are instantiated at run-time with *type arguments*. The generics increase significantly the code flexibility and reusability, because the code is more generic and therefore, more reusable. They are classified in *generic types* and *generic methods*. In this section we present some characteristics of these new language features and we also describe how the static and dynamic C♯ semantics are modified after the addition of these constructions.

These semantics are integrated in a new layer of the C♯ ASM model, called C♯ $_G$, which extends the last layer, namely C♯ $_U$, which deals with *unsafe code* (pointer types, pointer arithmetic); for details see [1].
The generic types can be generic classes, generic structs, generic interfaces or generic delegates.

The new grammar for type definition is the following one:

Type ::= TypeParameter | [NameSpaceName .] TypeName
TypeName ::= Identifier [TypeList] [. TypeName]
TypeList ::= < Type {, Type}>
NameSpaceName ::= Identifier {. Identifier}

The type referenced by TypeName can be a *RefType*, *ValueType* or *PointerType*:

RefType ::= ClassType | InterfaceType | DelegateType | ArrayType | NullType
ValueType ::= StructType | EnumType
PointerType ::= UnmanagedType * | void *

An *unmanaged type* is a *simple type, enum type, pointer type or a struct type that contains fields of unmanaged types only.*

The *managed types* and the *unmanaged types* can be also formulated by the following clauses:

$\forall$ T $\in$ *Type*:
- T $\in$ *ReferenceType* $\Longrightarrow$ T $\in$ *ManagedType*
- T $\in$ *StructType* $\wedge$
  ($\exists$ f $\in$ *instanceFields*(T): f $\in$ *ManagedType* $\vee$ f $\in$ *ManagedType**) $\Longrightarrow$
  T $\in$ *ManagedType*
- T $\in$ *UnmanagedType* $\Longleftrightarrow$ T $\notin$ *ManagedType*

We show some examples of generic type names:

List, List<T>, Dictionary<T, U>, Tree<int>
HashTable<K, List<string>>, Tree<U, T>.Node<U, T>

These are some examples of generic methods:

T F<T, U>(T t, U u) {...}
T G<T>(T t) where T : IComparable {...}

Constructed Types

A *constructed type* is a type which has a TypeList. It can be

- *open* (with type parameters)
  (e.g., List<T>, HashTable<K, List<string>>)
- *closed* (without type parameters)
  (e.g., List<int>, TreeMap<int, List<string>>).

An *open type* is a type parameter or an open constructed type. A *closed type* is type that is not open. At run-time only closed types can occur. A type argument is a closed type which replaces a type parameter at run-time.

Type parameters

The grammar for the type parameters is the following one:

TypeParameter ::= [Attributes] Identifier
TypeParameterList ::= < TypeParameter { , TypeParameter} >

The scope of a type parameter T includes the entire generic type/method declaration in which T is used. Type parameters are present only at compile-time. At run-time, they must be replaced by type arguments, because the open types can not appear at run-time.

Constraints for type parameters

The grammar for the type parameters constraint clauses is the following one:

TypeParameterConstraintClauses ::=
    TypeParameterConstraintClause {TypeParameterConstraintClause}
TypeParameterConstraintClause ::=
    where TypeParameter : TypeParameterConstraints

TypeParameterConstraints ::= Constraint {, Constraint}
Constraint ::= ClassConstraint | InterfaceConstraint | ConstructorConstraint
ClassConstraint ::= ClassType
InterfaceConstraint ::= InterfaceType
ConstructorConstraint ::= new()

Let TP be type parameter and TA the type argument supplied for type parameter TP. If TP has the class/interface constraint C, TA must be implicitly convertible to C (TA must be C or a type derived from C). If TP has a constructor constraint, TA must have a paramaterless constructor. A class constraint, if any, must be the first constraint. A constructor constraint, if any, must be the last constraint.

2.2 Subtype relation (Standard implicit conversion)

$\mathrm{A} \preceq \mathrm{B} \equiv \mathrm{A}$ "implicitly convertible to" B

Let T be a type parameter. We have the following relations:

- $\mathrm{T} \preceq \mathrm{T}$
- $\mathrm{T} \preceq$ object
- $\forall$ I $\in$ InterfaceConstraints(T)
  - $\mathrm{T} \preceq \mathrm{I}$
  - $\forall$ interface J: $\mathrm{I} \preceq \mathrm{J} \Longrightarrow \mathrm{T} \preceq \mathrm{J}$,
- $\forall$ C $\in$ ClassConstraints(T)
  - $\mathrm{T} \preceq \mathrm{C}$
  - $\forall$ interface I : $\mathrm{C} \preceq \mathrm{I} \Longrightarrow \mathrm{T} \preceq \mathrm{I}$,
  - $\forall$ class D : $\mathrm{C} \preceq \mathrm{D} \Longrightarrow \mathrm{T} \preceq \mathrm{D}$
- if ClassConstraints(T) $\neq \emptyset$ then
  - $\Lambda \preceq \mathrm{T}$ (Λ denotes the *null-type*)

So, we have:

- $\mathrm{T} \preceq \mathrm{U} \equiv \mathrm{U} \in \mathrm{Constraints(T)} \vee \exists\, \mathrm{V} \in \mathrm{Constraints(T)} : \mathrm{V} \preceq \mathrm{U}$
- $\mathrm{T}[\mathrm{R}_1]\ldots[\mathrm{R}_k] \preceq \mathrm{U}[\mathrm{R}_1]\ldots[\mathrm{R}_k]$ if $\mathrm{T} \preceq \mathrm{U}$

If $\mathrm{T}_1 <\mathrm{P}_1, \ldots, \mathrm{P}_n> \preceq \mathrm{T}_2 <\mathrm{P}_1, \ldots, \mathrm{P}_n>$ then

$\forall$ type arguments A_k supplied for type parameters P_k:
$\mathrm{T}_1 <\mathrm{A}_1, \ldots, \mathrm{A}_n> \preceq \mathrm{T}_2 <\mathrm{A}_1, \ldots, \mathrm{A}_n>$

2.3 Type constraints

Type constraints for $\mathrm{C}\sharp_G$ expressions and statements are described in the Tables 1 and 2.

We use the following notations:

- $A \preceq_i B$ denotes *A implements B*
- $A \preceq_e B$ denotes *an explicit conversion exists from A to B*

Table1. Type constraints for C$\sharp_G$ expressions

Expression	Constraints	Type of expression
T.default	T $\in$ Type	T
new T::M(args)	if T $\in$ TypeParameter then Length(args) $= 0$ ConstructorConstraint(T) $\neq \emptyset$	T
e as T	if T $\in$ TypeParameter then ClassConstraints(T) $\neq \emptyset$	T

Table2. Type constraints for C$\sharp_G$ statements

Statement	Constraints
throw exp	if type(exp) $\in$ TypeParameter then type(exp) $\preceq$ System.Exception
try stm catch (T exp) stm	if T $\in$ TypeParameter then T $\preceq$ System.Exception
using (T x = exp) stm	if T $\in$ TypeParameter then T $\preceq$ System.IDisposable
using (exp) stm	if type(exp) $\in$ TypeParameter then type(exp) $\preceq$ System.IDisposable

foreach (T x in exp) stm	let Te = type(exp)
	if Te $\not\preceq_i$ *CollectionPattern* then {
	(Te $\preceq$ System.Collections.Generic.IEnumerable<U> $\lor$
	Te $\preceq$ System.Collections.IEnumerable) $\land$
	elementType(Te) $\preceq_e$ T
	}

2.4 Evaluation of C♯ $_G$ expressions

ExecCsharpExp$_G$ ≡ **match** context(*pos*)

type.default ⟹ YieldUp(defaultVal(type))
exp == null ⟹ *pos* := exp
▶ val == null ⟹
    if type(val) ∈ ValueType **then** YieldUp(false)

3 Anonymous methods

3.1 Background

They are integrated in the new layer C♯ $_A$, which extends the previous layer, C♯ $_G$. Essentially, they are delegates, whose blocks are later converted into a method. They simplify the instantiation of a delegate. We are not enforced to create a separate method, just to associate it later with the delegate. Inside a delegate block, we can also access local variables and parameters of the enclosing method, which otherwise, if we would create a separate method, would be transmitted by parameters.

The grammar for anonymous methods is shown bellow:

AnonymousMethod ::= delegate [AnonymousMethodSignature] Block
AnonymousMethodSignature ::= (ParameterList)
ParameterList ::= Parameter {, Parameter}
Parameter ::= [Modifier] Type Identifier

The set of expressions is extended as follows:

Dexp ::= . . . | AnonymousMethod
Exp ::= . . . | (DelegateType) AnonymousMethod
(DelegateType) AnonymousMethod ≡ new DelegateType (AnonymousMethod)

Note: *Dexp* denotes a method invocation expression.

3.2 Subtype relation

Let D ∈ DelegateType and A ∈ AnonymousMethod

type(A) ⪯ D if

– A and D have *compatible parameter types* ≡
    if ParameterList(A) = ∅ then
        OutParams(D) = ∅
    else
        ParameterList(A) = $\{PA_1, \ldots, PA_N\}$
        ParameterList(D) = $\{PD_1, \ldots, PD_N\}$

$\text{type}(\text{PA}_K) = \text{type}(\text{PD}_K)$
$\text{isRef}(\text{PA}_K) = \text{isRef}(\text{PD}_K)$
$\text{isOut}(\text{PA}_K) = \text{isOut}(\text{PD}_K)$

– A and D have *compatible return types* $\equiv$
if retType(D) = void then
$\forall$ return statement S $\in$ block(A) : S = return;
else
$\forall$ return statement S $\in$ block(A) $\exists$ exp :
S = return exp; $\wedge$ type(exp) $\preceq$ retType(D)

3.3 Type constraints

The type constraints for $C\sharp\ _A$ expressions are described in Table 3.

Table3. Type constraints for $C\sharp\ _A$ expressions

Expression	Constraints	Type of Expression
new D(exp)	D $\in$ DelegateType if exp $\in$ AnonymousMethod then type(exp) $\preceq$ D	D

3.4 Definite assignment

The rules for the definite assignment [1], state if a variable is definitely assigned in its scope. They are the following:

before(block) = before(delegate block) = after(delegate block) = after(block)
before(block) = before(delegate (ParameterList) block) =
after(delegate (ParameterList) block) = after(block)

Where:

x $\in$ before(*item*) $\Longleftrightarrow$ x is definitely assigned before the elaboration of *item*
x $\in$ after(*item*) $\Longleftrightarrow$ x is definitely assigned after the elaboration of *item*

3.5 Evaluation of $C\sharp\ _A$ expressions

$\text{ExecCsharpExp}_A \equiv$ **match** context(*pos*)

new D(delegate block) $\Longrightarrow$
let T::M = meth, Mnew = newMethod **in**
params(Mnew) := $\emptyset$
retType(Mnew) := retType(block)

```
        modifiers(Mnew) := modifiers(M)
        body(Mnew) := block
        captureVars(Mnew)
        methods(T) := methods(T) . [Mnew]
        if static ∈ modifiers(M) then
            let d = new(Ref,D) in
                runTimeType(d) := D
                invocationList(d) := [T::Mnew]
                Yield(d)
        else let val =
           if T is Class then memValue(locals(this), T) else locals(this) in
            if val = null then FailUp(NullReferenceException)
            else let d = new(Ref,D) in
                runTimeType(d) := D
                invocationList(d) := [(val, T::Mnew)]
                Yield(d)
new D(delegate (p_1, ..., p_n) block) ⟹
    let T::M = meth, Mnew = newMethod in
        params(Mnew) = [p_1, ..., p_n]
        retType(Mnew) := retType(block)
        modifiers(Mnew) := modifiers(M)
        body(Mnew) := block
        captureVars(Mnew)
        methods(T) := methods(T) . [Mnew]
        if static ∈ modifiers(M) then
            let d = new(Ref,D) in
                runTimeType(d) := D
                invocationList(d) := [T::Mnew]
                Yield(d)
        else let val =
           if T is Class then memValue(locals(this), T) else locals(this) in
            if val = null then FailUp(NullReferenceException)
            else let d = new(Ref,D) in
                runTimeType(d) := D
                invocationList(d) := [(val, T::Mnew)]
                Yield(d)
```

We summarize the behavior of the above algorithm. When an anonymous method *AM* is encountered, one *dynamically* creates a new method *Mnew*, which imports its block, return type, parameters and modifiers from *AM*. Then, *Mnew* is incorporated in the enclosing class.

The *outer variables* (declared outside the delegate) that belong to the set of variables or parameters of the enclosing method (the method where the delegate is declared) must be captured in order to extend their lifetime.

They are encapsulated in the set OuterVars(Mnew) and *captured* using the following macro:

$\text{captureVars}(\text{Mnew}) \equiv$
$\quad$ **forall** $x \in \text{OuterVars}(\text{Mnew})$ **do**
$\quad\quad$ $\text{capturedLocals} := \text{capturedLocals} \oplus \{x \longmapsto \text{locals}(x)\}$

The set *capturedLocals* maps each variable x from OuterVars(Mnew) to its address, namely locals(x). We need this address when we actually invoke the delegate and consequentially we call the method *Mnew*.

The return type of the delegate block is obtained through the following macro:

$\text{retType}(p) \equiv$
$\quad$ **match** context(p)
$\quad\quad$ return $\Longrightarrow$ void
$\quad\quad$ return *exp* $\Longrightarrow$ *type*(*exp*)
$\quad\quad$ **otherwise** $\text{sup}([\text{retType}(c) \mid c \text{ in children}(p)])$

Where:

$\text{children}(p) \equiv [c \mid \text{up}(c) = p]$
$\text{sup}([t_1, \ldots, t_n]) \equiv$
$\quad t : t_1 \preceq t, \ldots, t_n \preceq t \wedge \forall\, x : t_1 \preceq x, \ldots, t_n \preceq x \rightarrow t \preceq x$
$\text{sup}([\ ]) = \mathit{Undef}$
$\text{sup}([t_1, \ldots, t_k, \mathit{Undef}, t_{k+1}, \ldots, t_n]) = \text{sup}([t_1, \ldots, t_n])$ (*Undef* is neglected)

At the end, we create a delegate object d of type D, to which we associate (i.e., add to its invocation list) the method *Mnew*. If *Mnew* is a non-static method, we need also the reference of the current object (i.e., *this*).

Some existing macros must be modified:

InitLocals $\equiv$
$\quad$ ... (the old implementation)
$\quad$ **forall** $x \in \text{OuterVars}(\text{meth})$ **do**
$\quad\quad$ $\text{locals}(x) := \text{capturedLocals}(x)$

FreeLocals $\equiv$
$\quad$ **forall** $x \in \text{LocalVars}(\text{meth}) \cup \text{ValueParams}(\text{meth})$ **do**
$\quad\quad$ **if not** captured(x) **then**
$\quad\quad\quad$ FreeMemory(locals(x), type(x))

Where:

capturedLocals = the set of captured variables
$\text{captured}(x) \equiv x \in \text{Indices}(\text{capturedLocals})$

InitLocals handles the memory allocation and initialization of the local variables and parameters during a *method call*. One simply imports the addresses of the outer variables, which are saved in the *capturedLocals* map. Then, we can normally access these variables.

FreeLocals handles memory deallocation during a *method exit*. One simply avoids the deallocation of the captured variables, because we need them when we actually call the anonymous method which uses them.

4 Iterator blocks

4.1 Background

Definition:

An iterator block is a block that yields an ordered sequence of values through yield statements.

They are integrated in the layer C♯ $_B$, which extends the layer C♯ $_A$. They ease the creation of iterators due to their *yield statements*.

The new grammar for the C♯ statements and for the *yield statements* is shown below:

Stm := . . . | YieldStm
YieldStm := yield return Exp ; | yield break ;

The rules for definite assignment [1] are:

before(exp) = before (yield return exp;)
after (yield return exp;) = after(exp)

4.2 The syntactical transformations

The implementation of the iterator blocks is done through *syntactical transformations*. The procedure is the following one:

1. One creates and initializes an array at the beginning of the block.
2. Then, if we meet a *yield return exp;* statement, we simply add the expression *exp* at the end of the array. If we meet a *yield break;* we return the existing array.
3. Finally, at the end of the block, we return the array.

The algorithm is illustrated bellow:

IEnumerable<T> FunctionName(ParList) {
    ArrayList<T> list = new ArrayList<T>;
    . . .
    yield return exp; $\Longrightarrow$ *list.add(exp);*
    . . .
    yield break; $\Longrightarrow$ *return list;*
    . . .
    return list;
}

```
IEnumerator<T> FunctionName(ParList) {
    ArrayList<T> list = new ArrayList<T>;
    ...
    yield return exp; ⟹ list.add(exp);
    ...
    yield break; ⟹ return list.GetEnumerator();
    ...
    return list.GetEnumerator();
}
```

The emphasized statements show the additions and the replacements made in the code. The arrows indicate the replacements.

5 Conclusion and Future work

Regarding the *generics*, they have very few operational (dynamic) semantics, which are implemented by adding some trivial transitions rules to the operational model, as described in 2.4. Furthermore, the *anonymous methods* and the *iterator blocks* can be implemented through syntactic sugar, as described in [3]. Therefore, the existing dynamic model doesn't suffer modifications in this case. If the *anonymous methods* are implemented as we described in this paper, some macros of the dynamic semantics must be modified, as shown in 3.5, but we think that our implementation is simpler than the one described in [3]; basically, it's just a *dynamic creation* of a new method, which implements the delegate block of an anonymous method. With respect to the ASM semantics, we consider that this methodology is easier than implementing the syntactical transformations described in [3], since they require very complex modifications in the program's abstract syntax tree structure. Our implementation of the *iterator blocks* is done through syntactic sugar, and we think is simpler than the one presented in [3]. It doesn't involve modifications in the existing dynamic model.

Of course, these new constructions triggered some important modifications in the existing static semantics (in subtype relation and type constraints), but the most important thing is that they didn't trigger major modifications in the existing dynamic semantics. Essentially, there were just additions of some new macros or transition rules. Modifications of the existing operational model arise only if one chooses to implemented the anonymous methods as we described in 3.5. In this case, the macros *InitLocals* and *FreeLocals* must be modified.

As future work, we may extend the implementation of the existing C♯ ASM model, in order to handle these new language features. In fact, it would be the implementation of the semantics described in this paper. By providing an executable version of the C♯ model, one can test the internal correctness of the model.

References

1. E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C♯. Journal Theoretical Computer Science, 2004.
2. S. Wiltamuth and A. Hejlsberg. C♯ Language Specification. MSDN, 2003.
3. C♯ Language Specification 2.0. MSDN, 2004.
4. R. F. Stärk and J. Schmid and E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.
5. E. Börger and R. F. Stärk. Abstract State Machines. A Method for High–Level System Design and Analysis. Springer–Verlag, 2003.
6. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1993.
7. Isabelle Attali, Denis Caromel, Marjorie Russo. A Formal Executable Semantics for Java. Proceedings of Formal Underpinnings of Java - An OOPSLA'98 Workshop, Vancouver, October 1998.
8. Charles Wallace. The Semantics of the C++ Programming Language. Specification and Validation Methods, ed. E. Brger, Oxford University Press, 1995, 131–164.