Side Effect Free Functions in Object-Oriented Languages

Roberto Ierusalimschy
Noemi Rodriguez

Departamento de Informática — PUC-Rio
`roberto, noemi@inf.puc-rio.br`

Abstract

This paper presents a method for statically verifying that functions do not produce side-effects, in an object-oriented language. The described model, although not allowing any changes to pre-existing objects during a function call, permits an imperative style of programming, where new objects can be freely created and manipulated.

Keywords: side-effects, object-oriented languages, type checking

Resumo

Este artigo apresenta um método para se verificar estaticamente que funções não produzem efeitos colaterais, em uma linguagem orientada a objetos. O modelo descrito, apesar de não permitir modificações em objetos pré-existentes durante a execução de um função, permite um estilo imperativo de programação, onde novos objetos podem ser livremente criados e manipulados.

Palavras chave: efeitos colaterais, linguagens orientadas a objetos, verificação de tipos

1 Introduction

Since the early times of Fortran, mathematical functions have been recognized as an important abstraction to be incorporated in programming languages. Some languages, like FP and Miranda, rely completely on pure functions to do their computations. However, programs written in imperative languages have to rely mainly on side-effects to achieve their goals.

Nevertheless, the importance of controlling side-effects in imperative languages is well recognized, both for formal reasoning and code optimization. It is well known that the problem of statically verifying whether a function can produce side-effects is non-computable, in any Turing-equivalent language (e.g. [8], p. 188). Therefore, any checking algorithm must find a balance between flexibility and tractability. In an object-oriented setting, the extensive use of pointers (or *object references*) makes this balance even more difficult to achieve, as objects can freely refer to other objects, with little or none visibility control.

This paper proposes an algorithm to check whether a method is side-effect free (*sef*) in the programming language School [13, 12]. School is an object-oriented language with separate type and class hierarchies. Types in School are considered partial specifications, and type checking is a method of partial program verification. Following this principle, the main goal of this proposal is to encompass one more aspect of program correctness into the type system of the language. The main requirements of our proposal are:

- the checking must be efficiently done in compile time.
- it must be secure, that is, methods accepted as *sef* cannot by any means produce side-effects.
- it must be flexible, in the sense that *sef* functions must be able to do most tasks which are in fact side-effect free.

The idea underlying this method is to use types to restrict the kind of operations allowed over an object. We propose a type transformation, called *old*, that modifies a type in such a way that objects with an old type cannot be modified. Then, a method declared as *sef* views all external objects (parameters, instance variables, etc) as having an old type, thereby disallowing side-effects on these objects. Nevertheless, *sef* methods can create new objects and manipulate them without restrictions.

The paper is organized as follows. Section 2 characterizes side-effect free functions. It also presents a formal memory model for execution of object-oriented languages, and using this model formally defines *sef* functions. Section 3 describes the basics of the type system in School; only the necessary concepts for the understanding of this proposal are presented. The next section defines the *old* transformation. This section also shows some properties which are satisfied by this transformation. In Section 5 we describe how the static checking of sef methods is carried out by the compiler. This is discussed further in Section 6, where some examples are analyzed. This section also compares the current proposal with other approaches. Finally, the last section is reserved for some conclusions.

2 *Sef* Methods

The purpose of this section is to give a precise meaning to the concept of side-effect free functions. As a first step, we propose some "levels" of side-effects:

1. at one extreme there are functions that use only the functional subset of the language. Therefore these functions can not have assignments, loops, etc;

2. a little less restrictive are functions that do not produce any change in the global memory when called, although they can do assignments to local variables. When such a function returns the whole state of the memory has the same value it had before the call.

3. another kind of functions are those which do not change old values in the memory, but can create new objects.

4. finally, most languages allow any kind of side-effect in functions, providing at most a warning in the manual that such side-effects are not good programming practice.

Clearly, the first two levels are much too restrictive. For instance, even an operation to "cons" two elements in a list, a basic operation in many functional languages, should not be considered side-effect free, since it allocates memory. The fourth level, on the other hand, is much too permissive, mainly in an object oriented framework. References among objects can propagate changes in ways difficult to anticipate.

Therefore, we want to explore the third definition. That definition presents a good balance. Any object in existence before the function is called cannot be modified in any way. However, new objects can be freely created and connected, and can even contain references to old objects. From now on, we will call this kind of function a *side-effect free* (*sef*) function.

It is important to notice that the real degree of side-effects in any language depends on the abstraction level the language handles the memory. For instance, if the language offers direct memory access, like C, then even a single function call produces side-effects, as it stacks a return address. Some other low-level operations, like a way to know the current amount of free memory, or conversions from pointers to integers, can also change the meaning of side-effects. In order to avoid these subtleties, we will introduce a formal memory model for execution of School, using VDM [10, 4]. This model will allow us to give a more precise meaning to *sef* methods.

To reflect the referential semantics of object oriented programming languages, all objects are accessed through unique identifiers (or pointers), that here will be represented by natural numbers. We reserve 0 to denote the object Nil.

$$ObId = \mathbf{N}$$

$$\text{N}\textsc{il} = 0$$

An Object has a local state, that associates instance variable names with their values, and a reference to its class, that is used to access its methods.

$$State = Name \xrightarrow{m} ObId$$

$$\begin{array}{lll} Object :: & state & : State \\ & class & : className \end{array}$$

Again it is important to emphasize that types have no meaning at run-time.

A *Memory* stores all objects different from Nil:

$$Memory = ObId \xrightarrow{m} Object$$

where

$$inv\text{-}Memory(m) \quad \triangleq \quad \text{N}\textsc{il} \notin \mathsf{dom}\, m \land \forall o \in \mathsf{rng}\, m \cdot \mathsf{rng}\, state(o) \subseteq \mathsf{dom}\, m \cup \{\text{N}\textsc{il}\}$$

The invariant states that Nil is the only primitive value. Therefore, it has no representation in the memory. Moreover, every *ObId* stored in an instance variable must be represented in the memory, unless it is Nil.

Our next domains are used to describe the semantics of methods (operations). A method acts over a global state, a receiver and a list of parameters, and returns a new global state and a result list[†]. To model this behavior we use the following declarations:

$$Method = Memory \times ObId \times ObId^* \rightarrow MethodResult$$

$$\begin{array}{lrcl} MethodResult :: & state & : & Memory \\ & results & : & ObId^* \end{array}$$

Using the above definitions, we can formalize our classification scheme for functions. Functions that neither create new objects nor change old ones are defined by the following predicate:

$$noChange : Method \rightarrow \mathbf{B}$$

$$\begin{array}{l} noChange(met) \;\; \triangleq \\ \quad \forall oldMem \in Memory, self \in ObId, par \in ObId^* \cdot \\ \qquad \mathsf{let}\ mk\text{-}MethodResult(newMem, res) = met(oldMem, self, par)\ \mathsf{in} \\ \qquad newMem = oldMem \end{array}$$

That is, the final state is always equal the initial one. The only real effect of the function is given by its result values.

The kind of function we are really interested in, which we called *sef* functions, can create new objects when called, but cannot change old ones. This notion is defined as:

$$expand : Memory \times Memory \rightarrow \mathbf{B}$$

$$\begin{array}{l} expand(oldMem, newMem) \;\; \triangleq \\ \quad \mathsf{dom}\, oldMem \subseteq \mathsf{dom}\, newMem \wedge \forall obj \in \mathsf{dom}\, oldMem \cdot oldMem(obj) = newMem(obj) \end{array}$$

$$isSef : Method \rightarrow \mathbf{B}$$

$$\begin{array}{l} isSef(met) \;\; \triangleq \\ \quad \forall oldMem \in Memory, self \in ObId, par \in ObId^* \cdot \\ \qquad \mathsf{let}\ mk\text{-}MethodResult(newMem, res) = met(oldMem, self, par)\ \mathsf{in} \\ \qquad expand(oldMem, newMem) \end{array}$$

Objects present in *oldMem* are called *old* objects. Objects present in *newMem* but not in *oldMem* are called *new* objects; these are exactly the objects created during the execution of the method. Because the invariant of *Memory* states that objects can only refer to objects present in memory, and because old objects do not change during a *sef* method, old objects cannot refer to new ones. It is important to notice that the definitions of old and new objects are relative to a specific *sef* method call.

3 Types in School

The concept of type in School is an important factor in this proposal. Types and classes are different concepts in School. A class specifies the implementation of similar objects. Types, on

[†]Methods in School can return multiple values

the other hand, represents the interface of an object, and are used only by the compiler. Every object belongs to a fixed class but can satisfy several different types.

School is a programming language designed to keep the basic semantics of object-oriented languages, mainly Smalltalk, while offering a secure type system. All values in School are objects. Variables do not contain objects, but references to them, and both assignment and parameter passing manipulate references. All messages between objects are based on late-binding, and the binding depends solely on the receiver.

In object-oriented programming languages, objects carry their own operations, and these operations are the only visible part of an object; the only thing one can do with an object is send a message to it. So, if we want to characterize types as, for example, "a collection of values sharing a common structure or shape" ([1], p. 104), we should use only the operations as the visible structure of an object. Therefore, a good definition for types in OO languages is that the type of an object is its interface, that is, its collection of operations. Following this definition, we can say that a type error in an OO program is the sending of a message to an object that has no method for it.

The main design goal of School's type system has been to avoid any construction that can cause a type error at run-time; obeying this rule, we have tried to make the language as flexible as possible. In order to do that, School has separate hierarchies for types and classes (specifications and implementations), structural subtyping, and constrained genericity.

The main idea underlying the concept of separate hierarchies is the understanding of types as specifications, and classes as implementations. The type of an object is its external appearance, that is, its interface to the outside world. In School, a type declaration states all operations available in objects of that type, and the types of parameters and eventual results of each operation. On the other hand, the class of an object dictates its internal shape, that is, its structure and the code to handle it.

Figure 1 shows typical type and class declarations. This program fragment defines a type and a class. The type declaration defines a new type, called `Stack`. This type is *generic*, because it has a type parameter (`T`). The declaration states that any object of this type must have an operation `push`, that accepts an object of type `T`, an operation `pop`, that returns an object of type `T`, and an operation `isEmpty`, that returns a boolean. The example also defines a class `StackArray`, an implementation for the type `Stack`. Each instance of this class has two variables, `values` and `top`, and operations `push`, `pop`, and `isEmpty`. Besides instance variables and operations, a class also defines *constructors*. Constructors are used to create new objects. For instance, the expression `StackArray{Int}!new(20)` creates a new stack of integers, and calls the constructor `new` to initialize it. Notice that constructor `new` of class `StackArray` calls constructor `new` of class `Array` to initialize the instance variable `values` with a new array.

The independence between subtyping and subclassing allows School to adopt *structural compatibility*. That means that a type is a subtype of another one if they are compatible in some way. To say that a type A is a *subtype* of a type B means that A is compatible, from an external point of view, with B. That is, an A object can be used wherever a B object is expected. Notice that A and B do not need to have similar implementations. On the other hand, to say that a class A is a *subclass* of a class B means that A inherits methods and variables from B. As they do not need to have compatible interfaces, A can freely modify the inherited features. Using the classification proposed in [15], subtyping must have behavior compatibility, or at least signature compatibility, while subclassing is free to adopt cancel compatibility. Therefore, there is no compromise between the flexibility of cancel compatibility and the security of strong typing.

Following our stated main rule, we want to allow a type A to be a subtype of B as long

```
type Stack{T} is
  proc push (T);
  proc pop -> T;
  proc isEmpty -> Boolean;
end Stack;


class StackArray{T} : Stack{T} is
  var values : Array{T};
      top : Int;

  proc push (v : T) is
    values.putAt(top, v);
    top := top+1;
  end;

  proc pop -> T is
    top := top-1;
    Result := values[top];
  end;

  proc isEmpty -> Boolean is
    Result := top = 0;
  end;

  constructor new (size : Int) is
    values := Array{T}!new(size);
    top := 0;
  end;

end StackArray;
```

Figure 1: Example of type and class declarations in School

as there is no possibility of error when using an A object in the place of an object of type B. As errors occur when a message is sent to an object which has no method for it, we can avoid them with the following definition: a type A is a subtype of B ($A \prec B$) if and only if, for each method X in B, with signature $P_{B_1} \times \ldots \times P_{B_n} \rightarrow R_{B_1} \times \ldots \times R_{B_m}$, there is a method X in A, with signature $P_{A_1} \times \ldots \times P_{A_n} \rightarrow R_{A_1} \times \ldots \times R_{A_m}$, where for all $i \leq m$, $R_{A_i} \prec R_{B_i}$ and, for all $i \leq n$, $P_{B_i} \prec P_{A_i}$. The apparent inversion in the last condition is the well known "contra-variance rule", and is needed to ensure correctness [3].

Notice that the above definition is not well grounded, since it can result in an infinite recursion if a type refers to itself. This definition is, however, useful for intuitive reasoning. The key idea for understanding recursive types is that we assume that a type A is subtype of B whenever this assumption has no possibility of giving rise to an execution error. Therefore, if we arrive at a conclusion such as $A \prec B$ iff $A \prec B$, we can safely assume that $A \prec B$. This will be referred to as the *recursion hypothesis*. The formal definition of subtyping, justifying this intuition, is presented in the appendix. A more complete treatment, as well as a formal proof that this definition avoids type errors, can be found in [9].

Generic types do not take part in the subtype relationship. School has no rules like $A \prec B \Rightarrow X\{A\} \prec X\{B\}$, where $X\{T\}$ is a generic type. Only after instantiation a type can be compared with another one, following the above rules.

4 The Concept of *Old* Type

The static verification of side-effect free functions implies in an extension of the type system defined in Section 3. In this extension, methods which are allegedly side-effect free (*sef*) will be prefixed with the keyword `sef`. The use of this prefix will cause the compiler to check if the method is effectively sef.

The main restriction over *sef* functions is that they cannot change old objects. The idea underlying our algorithm is to use the type system to keep track of old objects, and to distinguish between new and old objects. In order to do that, we create the concept of *old* types, which are created by a type operator called *old*. Then we will ensure that, during the execution of a *sef* function, old objects can only be referred through old types.

The type operator *old* is applied on types of old objects, creating a safe view of them. This is the view used by the compiler to check if a method is effectively side-effect free. As previously noted, the concept of type is used in School solely for static checking. Looking at a same object through different views is simply a compiler strategy, and does not imply in transformations or conversions of this object at runtime.

If T is a type, $old(T)$ is the type which results from executing the following algorithm:

1. in all methods, substitute each result type U for $old(U)$;

2. mark as *excluded* all methods in T which are not sef methods.

The second rule is the most obvious, since it prevents a program from directly modifying an old object, excluding methods which should not be called. Methods marked as excluded may not be called but are still part of the type. This is important for reasons which will be discussed later.

The recursive application of the old transformation on the result types is needed because parts of an old object are always old. We must avoid modifications made on preexisting objects by first querying the object (with a sef method) and obtaining some part of its state, and then applying a non-sef method on this part. By placing protection restrictions on any object returned by a method called on an old object, we are effectively prohibiting any modifications on parts of this object.

Now, the subtyping rules presented in the previous section must be extended to deal with excluded and sef methods. A type A is a subtype of B ($A \prec B$) if and only if, for each method X in B, with signature

$$P_1 \times \ldots \times P_n \rightarrow R_1 \times \ldots \times R_m,$$

there is a method X in A, with signature

$$Q_1 \times \ldots \times Q_n \rightarrow S_1 \times \ldots \times S_m,$$

where for all $i \leq m$, $S_i \prec R_i$, and for all $i \leq n$, $P_i \prec Q_i$. If X is a sef method in B, X must also be a sef method in A. Furthermore, if X is not excluded in B, X cannot be excluded in A.

The rule which prohibits that a sef method be redefined as non-sef is the natural provision for dealing with side-effect free calls in the presence of polymorphism. The second new rule, which states that a method may not be excluded in a type A and not excluded in one of its supertypes, B, is necessary to guarantee that it is not possible to call a method that is marked as excluded on an object of type A by previously assigning this object to a variable of type B.

The type $old(T)$ has several properties which will be useful in allowing for programming flexibility in methods guaranteed to be side-effect free. These properties are discussed below.

We also define another type transformation, *sef*, which is quite similar to *old*. The only reason for this definition is to provide programmers with a shorthand, avoiding the need for explicit definition of new types when programming sef methods, as will be discussed in Section 6.

If T is a type, $sef(T)$ is the type which results from executing the following algorithm:

1. eliminate from T all non-sef methods;

2. substitute each result type U in the remaining methods for $sef(U)$.

The difference between $sef(T)$ and $old(T)$ is that $old(T)$ can be said to "retain a memory" of non-sef methods. In the *old* transformation, these methods are marked as excluded, which implies that they cannot be called but must still be taken into consideration when analyzing subtype relationships. In the *sef* transformation, these methods are totally forgotten, so $sef(T)$ may be regarded as a projection of T on the space of sef methods.

Two useful definitions are of a *sef* type and an *old* type. A type T is sef if $sef(T) = T$. Likewise, a type T is old if $old(T) = T$. Particularly, for any type T, $sef(T)$ is a sef type, and $old(T)$ is an old type. Sef types represent immutable objects, like integers and reals. Old types are typically created by the old transformation, because this is the only way to create excluded functions. Moreover, the *old* transformation does not change a *sef* type.

Some Nice Properties

It is always possible to guarantee that functions are side-effect free by requiring them to satisfy restrictions which nullify their practical use. This section discusses some properties of the *old* type transformation defined in the previous section which have an important role in the programming flexibility achieved by side-effect free functions in School. The "proofs" presented here intend to show the intuition behind these properties, and therefore use the informal definition of subtyping presented in the previous section. For a formal treatment, we refer the reader to the appendix.

The properties to be discussed are:

1. $A \prec old(A)$

2. If $B \prec A$ then $old(B) \prec old(A)$

3. $A\{old(B)\} \prec old(A\{B\})$

4. $old(A) \prec sef(A)$

5. If $old(B) \prec A$ then A is old

The first property states that a type A is always a subtype of the type created by the *old* transformation. To see why this is the case, let us consider a type A such as:

```
type A is
  sef proc NoChange (A,B) -> (A,B);
  proc Change (A,B) -> (A,B);
end A;
```

Note that most of the possible cases are covered in the definition above, since A has a sef and a non-sef method, each of these taking as parameters objects of type A itself and of another type, B, and returning objects of these two types. So, although the following reasoning is based on this example, it is in fact quite general.

Applying the *old* transformation over A results in:

```
type old(A) is
  sef proc NoChange (A,B) -> (old(A),old(B));
  excluded proc Change (A,B) -> (old(A),old(B));
end old(A);
```

The basic difference between A and $old(A)$ is that method `Change` is excluded in $old(A)$. Methods which are excluded in a type can be excluded or not in its subtypes, so this is all right. The other effect of the transformation is to recursively transform the result types of methods `NoChange` and `Change`. The co-variance rule for result types requires that $A \prec old(A)$ and $B \prec old(B)$. This follows directly from the recursion hypothesis, mentioned in Section 3.

Our second property states that the *old* transformation preserves subtype relationships. This is clearly a nice property since, intuitively, it means that assignments which were correct under the common typing rules will still be correct in side-effect free functions, as long as they are safe.

To see that this property holds, consider types A and B defined as:

```
type A is
  sef proc NoChange (A,U) -> (A,U);
  proc Change (A,U) -> (A,U);
end A;
type B is
  sef proc NoChange (C,V) -> (D,X);
  proc Change (C,V) -> (D,X);
  ...
end B;
```

Note that, in the definition above, if $B \prec A$, then the following relationships hold: $A \prec C$, $U \prec V$, $D \prec A$, and $X \prec U$. If we apply the *old* transformation on both types, we have:

```
type old(A) is
  sef proc NoChange (A,U) -> (old(A),old(U));
  excluded proc Change (A,U) -> (old(A),old(U));
end;
type old(B) is
  sef proc NoChange (C,V) -> (old(D),old(X));
  excluded proc Change (C,V) -> (old(D),old(X));
  ...
end;
```

To see that $old(B) \prec old(A)$, note first that all methods of B which become excluded in $old(B)$ and are present in A also become excluded in $old(A)$. All methods will have their result types transformed both in $old(A)$ and in $old(B)$. So, we need to show that $old(D) \prec old(A)$ and that $old(X) \prec old(U)$, but this is again a consequence of our recursion hypothesis.

In order to see that the third property stated above also holds, consider a type A defined as:

```
type A{T} is
  sef proc NoChange (A,U,T) -> (A,U,T);
  proc Change (A,U,T) -> (A,U,T);
end A;
```

This type instantiated with $old(B)$ gives:

```
type A{old(B)} is
  sef proc NoChange (A,U,old(B)) -> (A,U,old(B));
  proc Change (A,U,old(B)) -> (A,U,old(B));
end A;
```

On the other hand, applying *old* to $A\{B\}$, we have:

```
type old(A{B}) is
  sef proc NoChange (A,U,B) -> (old(A),old(U),old(B));
  excluded proc Change (A,U,B) -> (old(A),old(U),old(B));
end A;
```

So, for $A\{old(B)\}$ to be a subtype of $old(A\{B\})$ what we need to have is that $B \prec old(B)$ (contra-variance on parameter types), $A \prec old(A)$, and $U \prec old(U)$ (co-variance on result types), but this is our first property.

A generic type parameter may be used as the type of a method parameter, a method result, or an instance variable. In the last case, we may think of an object with type $A\{T\}$ as an object of type A which has references to objects of type T; this situation typically arises when A is a container for objects of type T. Therefore, one interesting way to read this last property is that when $A\{T\}$ is old there is no need to say that T is old too, because old objects can only point to other old objects. We will show later on how this can be used in the creation of container objects.

The fourth property states that, for any type A, $old(A)$ is a subtype of $sef(A)$. To prove it, let us again consider our definition of a general type A:

```
type A is
  sef proc NoChange (A,B) -> (A,B);
  proc Change (A,B) -> (A,B);
end A;
```

The transformations *old* and *sef* will result in the types:

```
type old(A) is
  sef proc NoChange (A,B) -> (old(A),old(B));
  excluded proc Change (A,B) -> (old(A),old(B));
end;

type sef(A) is
  sef proc NoChange (A,B) -> (sef(A),sef(B));
end;
```

For $old(A)$ to be a subtype of $sef(A)$ we only need to have the correct co-variance in result types of method `NoChange`. This is again a consequence of our recursion hypothesis.

Finally, let us consider property 5. If A is supertype of an old type $old(B)$, all non-sef methods in A must be excluded, because they are excluded in $old(B)$. Let T be any type returned by a method of A. This type must be supertype of its correspondent return type in $old(B)$, which is old. Using our recursion hypothesis, we conclude that T is also old. But, if all non-sef methods in A are excluded, and all return types of A methods are old, then $A = old(A)$, that is, A is old.

The importance of these properties will be discussed in Section 6.

5 Checking Side Effect Free Methods

We have hinted that checking of sef methods is done by ensuring that, during the execution of a *sef* function, old objects can only be referred through old types. In order to keep this invariant, the following rule is introduced:

1. When typing a *sef* method, all types T for objects external to the method, i.e. parameters, instance variables, and the pseudo-variable *self*, are substituted for $old(T)$;[‡]

The proof that the above rule keeps the invariance is as follows: when a *sef* method starts, all directly accessible old objects are its parameters, instance variables, and *self*, which have old types. By definition of the *old* transformation, all indirectly accessible old objects also have old types. Moreover, the type system of School ensures that any object can only be referred through supertypes of its original type. But property 5 states that a supertype of an old type is always old. Therefore, there is no way to access an old object through a non-old type.

It is clear that a *sef* method should not modify any instance variable. Therefore, one more rule is needed when checking a *sef* method:

2. assignments to instance variables are not allowed.

With these two rules, we can prove that *sef* methods cannot have side-effects: the only way to create a side-effect is by calling a non-sef method over an old object, because only the object itself can access its instance variables, and a sef method cannot directly change an instance variable (Rule 2). On the other hand, as long as old objects have old types, they cannot be modified, as old types exclude any non-sef method, and excluded methods cannot be called.

Notice that, besides the extra rule that prohibits assignments to instance variables, all the checking after the transformations is carried on by the type checker. This is one point we consider important in this proposal: the same set of rules used for type checking in general is the

[‡]It is important to notice that School does not have global variables.

most important tool for guaranteeing "side-effect freeness". This means that the introduction of side-effect free methods does not imply in much extra implementation effort or complexity of understanding.

Although the two rules above are enough to ensure safeness, the resulting system is a little too rigid. The two rules below are introduced solely with the purpose of increasing flexibility. Their importance will be discussed in the next section.

3. when checking calls on methods of objects currently seen as *old*, the formal parameter types T of these methods are viewed as $old(T)$;

4. when checking the return statement, the result type T is viewed as *old*(T).

Rule 3 states that when checking calls on methods of the objects currently seen as old, the formal parameter types T of these methods are also viewed as $old(T)$. This is always safe because methods called on old objects will in no case allow modifications on parameters, and their return values will also be protected by the *old* transformation. Notice that any object passed to a *sef* method will be seen as old inside it, which makes it perfectly safe to carry on type checking as if the formal parameter was old.

Rule 4 states that an object of type $old(T)$ is compatible with a result of type T, and is a kind of reverse of rule 3. It may seem at first sight that this rule represents a breach in our checking, since it allows an old object to be seen as new. The point here is that the goal of a sef method is to guarantee the absence of side-effects only during its execution. So it is possible to introduce the special rule for the return statement without creating any danger of unwanted modifications. Modifications on the object returned by a method M will be ruled out in the checking of the method which called M, if necessary, as discussed in the examples.

Notice that the introduction of rules 3 and 4 for checking purposes is not the same as adding them to the *old* transformation. To understand this, let us consider rule 3. We could try to include a third step in the *old* transformation, substituting each parameter type T for $old(T)$. But in this case, we would lose a very important property, namely, the one that states that $A \prec old(A)$; contra-variance would rule it out.

In the next section, we discuss the use of side-effect free methods using some examples to illustrate relevant issues.

6 Discussion and Examples

To begin our discussion, we present some simple examples which illustrates the basics of the proposed mechanism. The last example shows more of its power.

Figure 2 shows a fragment of code containing a sef method `Front` defined inside a parameterized class `queue{T}`, and a method `M`, outside class `queue{T}`, which contains a call to `Front`.

Method `Front` is a very simple example of a common pattern, a method for querying a structure; in this case, the query is for the first element in the queue. `GetLow` is a sef method in class `array{T}`, with result type T, making the code in method `Front` obviously harmless. One point to be noted is that this implementation would not be allowed had we not added rule 4, for checking the return statement. Since the type of `self` has undergone the *old* transformation, the result of the call to `GetLow` has type $old(T)$, which is not a subtype of the declared result type T. This would obviously rule out many useful program fragments.

Method `M` takes as a parameter a queue of objects of type *Process*, and illustrates actions which are considered illegal in a sef method. A local variable `first` is defined with the same

```
class queue{T} is
      includes array{T};
  ...
  sef proc Front() -> T is
    Result := self.GetLow();
    -- returns first object in queue
  end Front;
end queue;

class C is
  sef proc M(sleeping: queue{Process}) -> Bool is
  var
    first: Process;
    first := sleeping.Front();    -- illegal!!
    if first.TimeLeft() = 1 then
      wake_up_the_first_process_in_the_queue;
      Result := true;
    else
      first.DecrementTimeLeft();  -- affects the actual process in queue
      Result := false;
    end
  end M;
end C;
```

Figure 2: A simple example of sef methods

```
type Comp is
  sef proc LessEqual (Comp) -> Boolean;
  ...
end Comp;
...
sef proc Min (o1: Comp, o2: Comp) -> Comp is
  if o1.LessEqual(o2) then
    Result := o1
  else
    Result := o2
  end
end Min;
```

Figure 3: Method `Min` returns the least of two objects

type *Process*. So, one could wonder whether it would not be possible to modify a part of the pre-existing object `sleeping` by first querying this object for its first process and assigning the result to a local variable, and then applying a non-sef method to this local variable, since local variables suffer no restrictions. However, since `M` is sef, `sleeping` has type *old*(*queue*{*Process*}) for checking purposes, and the call to `sleeping.Front()` will thus return an object of type *old*(*Process*), which may not be assigned to a variable of type *Process*.

The incompatibility between the type returned by `sleeping.Front()` and the type of a local variable such as `first`, while an asset in the example, may sometimes represent a burden, since, in order to use a local variable for temporary storage of the value returned by this call, explicit definition of a new type, compatible with *old*(*Process*), would be needed. To avoid this burden, we allow the *old* and *sef* transformations to be used explicitly in declarations.

The explicit use of the *sef* transformation is allowed in any declaration. The *old* transformation, however, cannot be used in the declaration of parameters or results. This is because the concept of *old* types is local to each method call, and does not make sense out of this context.

Using the above facility, the variable `first` could be declared with type *old*(*Process*), making the offending assignment legal. In this case, however, the call

```
first.DecrementTimeLeft()
```

would be illegal, since `DecrementTimeLeft` would be excluded from the type of `first`. In one way or another, side-effect freeness is ensured.

We now discuss an example where we make use of the two rules added for flexibility. Figure 3 shows a method which takes two parameters of a same type *Comp* and returns the minimum between them.

There are two points worth noting in this example. Parameters `o1` and `o2` have type *old*(*Comp*) inside method `Min`. If the third rule had not been introduced in the sef checking algorithm, the call to `o1.LessEqual` would not be legal. Again we point out that this rule does not introduce any potential harm, since calls on old objects may not result in any modification.

The other important point in this example is that rule 4 is again needed for the typing of the return statement, since the method returns an object with type *old*(*Comp*) and its declared return type is *Comp*.

It may be interesting at this point to highlight the difference between *sef* and *old*. It has been said before that the *old* transformation retains a memory of the transformed type, and

```
type A is
  sef proc M1;
  proc M2;
end A;
type B is
  sef proc M1;
  proc Copy(A);
end B;
...
sef proc Clone (a: A) -> A is
  var b: B;
  ...
  b.Copy(a);
  Result := b;  -- Illegal
end Clone;
```

Figure 4: Method `Clone` returns an object of the wrong type

we can now point out the importance of this memory in the use of the fourth rule used for checking sef methods.

Let us consider what could happen if the *old* transformation simply eliminated all non-sef methods. Figure 4 shows a method `Clone` which should return an object of the same type A as was passed to it. The implementation is erroneous, and returns an object of type B, which is not a subtype of A. However, in method `Clone` the return type is viewed as $old(A)$; if method $M2$ were not part of $old(A)$, the error would not be detected.

Our last example deals with a common situation of creating a copy of an existing structure, in this case, a list. This example is interesting because it deals with the problem of having new objects point to old ones, a potentially dangerous situation, since one could imagine modifications could be made to the old object through the new one.

Figure 5 shows the definition of a parameterized class `LinkedList` which implements a parameterized type *List*, also shown in the figure. The point of interest is the definition of the *sef* method `Reverse`, that returns a list which is the reverse of the receiver. The new list is created in the expression

```
LinkedList{old(T)}!New
```

and has type $List\{old(T)\}$. This means that the call `revList.Append` will only accept objects of type $old(T)$ as parameters. This is exactly the type of object that is retrieved by the call `self.Get(i)`, so the call to `Append` is correctly typed.

The return statement of `Reverse` also has an interesting type checking. As noted, the new list has type $List\{old(T)\}$, while the result type is $List\{T\}$. Here again rule 4 comes into play. The result type is viewed as $old(List\{T\})$ which, according to property 3 of Section 4, is a supertype of $List\{old(T)\}$.

It is important to emphasize that, because `self` is seen as old, its *sef* method `Get` returns old objects. Therefore, there is no way to modify these elements inside method `Reverse`. Nevertheless, these elements can still be manipulated, allowing the construction of the new list. At the end, the type checking of the return statement allows the whole list to be typed as new.

```
type List{T} is
  proc Append(T);
  sef proc Reverse () -> List{T};
  sef proc Get (index: Int) -> T;
  ...
end List;

class LinkedList{T}: List{T} is
  ...
  constructor New() is
    ...
  end New;

  sef proc Reverse() ->  List{T} is
  -- returns a new list which is the reverse of self
    var
      i: Integer := self.Length;
      revList: List{old(T)} := LinkedList{old(T)}!New;
    while i >= 1 do
      revList.Append(self.Get(i));  -- OK: old(T) expected in Append
      i := i-1
    end;
    Result := revList; -- OK: List{old(T)} is a subtype of old(List{T])
  end Reverse;
end LinkedList;
```

Figure 5: Creating a new list from an existing one

Finally, let us see how the checking behaves in a method calling `Reverse`. If `Reverse` is applied over a new list, its result is new, as expected. On the other hand, if `Reverse` is applied over an old list, its result is seen as old, too, because of the *old* transformation.

Some Other Approaches

Side-effect free functions are the basis of many functional languages, like ML and Miranda. However, these languages need a completely different approach to programming. Moreover, even among declarative languages, many allow side-effects, and some need side-effects in order to do specific tasks, like input (e.g. Lisp and Prolog).

Most imperative languages try to restrain side-effects through small restrictions. In Ada, for instance, "The formal parameters of a function must all have the mode **in** (discouraging the definition of functions with side-effects; however global variables can still be changed to produce side-effects)" ([5], p. 36).

An early attempt to effectively control side-effects was done in the design of Euclid [11]. This system does not allow the creation of new objects inside a function. For an object-oriented language, where objects are dynamically created, this is too restrictive. A more recent effort in a similar direction is Super Pascal [7]. However, this language does not have pointers, which represent the main difficulty in object-oriented languages.

Another approach comes from C++ [14]. Through different uses of the keyword *const*, a programmer can limit modifications over function parameters and *self*. This is a valuable feature for program documentation and verification. However, there is no simple way to ensure that a given function (or method) is completely side-effect free. Moreover, the mechanism is not secure, and can be circumvented by typecasts or by holding a non-const pointer to *self* (see [14], p. 149).

Griesemer, in [6], proposes a very different approach, using dynamic verification instead of a static technique. This method has the advantage of allowing, in a *sef* function, any program construct that does not cause a side-effect. On the other hand, errors can be detected only by exhaustive tests.

7 Conclusions

We have presented a proposal for statically verifying that a function does not produce side-effects. The described model, although not allowing any changes to pre-existing objects, permits an imperative style of programming, where new objects can be freely created and manipulated inside a function.

One important property of this model is that it is completely based on the type system of the language. Thus, it goes in the direction proposed in [2], extending the meaning of types to encompass more aspects of program correctness. A key point behind the model is that types, in School, are partial specifications; implementations are described by classes, an independent concept. This allows School to use structural compatibility for types, which results in the high degree of flexibility presented by our proposal. It is important to notice that the version of School adopted here does not support global variables. An extension to the presented solution would have to be made in order to deal correctly with this feature.

The model proposed here was based on the programming language School, but most of its ideas can be applied to other object oriented languages. Clearly, this is harder for languages that do not yet recognize the importance of separating specifications from implementations.

Acknowledgments

The authors are indebted to Robert Griesemer for starting the discussion about side-effect free methods in an object-oriented framework, and giving valuable comments on an early version of this paper. We would also like to thank the International Computer Science Institute (ICSI — Berkeley), where part of this work has been developed. This work has been partially supported by CNPq (the Brazilian Research Council).

References

[1] R. Amadio and L. Cardelli. Subtyping with recursive types. In *ACM Conference on Principles of Programming Languages*, 1991.

[2] L. Cardelli. Typeful programming. In *notes of IFIP Advanced Seminar on Formal Description of Programming Concepts*, Petropolis – Brazil, 1989.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[4] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[5] N. Gehani. *Ada, an Advanced Introduction*. Prentice Hall Software Series, 1983.

[6] R. Griesemer. Detection of side-effects in function procedures. ICSI, 1994.

[7] P. Brinch Hansen. the programming language Super Pascal. *Software — Practice and Experience*, 24(5):467–484, 1994.

[8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[9] R. Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer Languages*, 19(1):19–40, 1993.

[10] C. B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice Hall, second edition, 1990.

[11] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *Sigplan Notices*, 12(2), 1977.

[12] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Conciliação de flexibilidade e verificação estática em linguagens orientadas a objetos. In *VII Simpósio Brasileiro de Engenharia de Software*, pages 282–294, 1993.

[13] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.

[14] B. Stroustrup. *The C++ Programming Language*. Prentice-Hall, second edition, 1992.

[15] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP'88 Proceedings*, pages 55–77, 1988. LNCS 322.

Appendix: A Formal Approach to Recursive Types

Types in School can be freely recursive. For instance, the following definitions:

```
type A is
  proc F (A) -> B;
end A;

type B is
  proc F (A,B) -> B;
end B;
```

are valid and correct in the language. This implies that when we talk about structural compatibility we must take some care, to avoid circular definitions.

As a first difficulty, if we say something like "a type T is a set of operations with their signatures", where the signature of an operation is a list of the parameter and result types, then the type A written above would have an infinite structure. One simple way to avoid that problem is to rely on the names of the types, changing the definition of signature to be a list of *the names* of the parameter and result types.

The second difficulty is with recursive properties about types. The usual definition of structural subtyping goes like:

> a type A is a subtype of a type B ($A \prec B$) if and only if, for each method X in B, with signature
> $$P_1 \times \ldots \times P_n \rightarrow R_1 \times \ldots \times R_m,$$
> there is a method X in A, with signature
> $$Q_1 \times \ldots \times Q_n \rightarrow S_1 \times \ldots \times S_m,$$
> where for all $i \leq m$, $S_i \prec R_i$, and for all $i \leq n$, $P_i \prec Q_i$.

Such definition leads to an endless recursion when applied to a recursive type. We arrive at conclusions like "A is subtype of B if and only if A is subtype of B".

The solution we will adopt here has been presented in [9], and again it uses type names. The mechanism to avoid recursion problems is to use an explicit map that, for each type name, gives a set of all supertypes of it. We call this kind of map a *hierarchy*.

$\mathit{Hierarchy} = \mathit{TypeName} \xrightarrow{m} \mathit{TypeName}\text{-}\mathsf{set}$

Type A is a subtype of type B *with respect to a hierarchy* $\mathcal{H}$ ($A \prec_{\mathcal{H}} B$) if and only if $B \in \mathcal{H}(A)$. Type A is a subsignature of B (with respect to a hierarchy $\mathcal{H}$) if and only if for each method X in B, with signature

$$P_1 \times \ldots \times P_n \rightarrow R_1 \times \ldots \times R_m,$$

there is a method X in A, with signature

$$Q_1 \times \ldots \times Q_n \rightarrow S_1 \times \ldots \times S_m,$$

where for all $i \leq m$, $S_i \prec_{\mathcal{H}} R_i$, and for all $i \leq n$, $P_i \prec_{\mathcal{H}} Q_i$. If X is a sef method in B, X must also be a sef method in A. Furthermore, if X is not excluded in B, X can not be excluded in A.

We say that a hierarchy $\mathcal{H}$ is *consistent* if, for all types A and B, if $A \prec_{\mathcal{H}} B$, then A is a subsignature of B.

Now we are able to give a formal definition for subtyping. We say that A is subtype of B ($A \prec B$) if and only if there is a consistent hierarchy $\mathcal{H}$ such that $A \prec_{\mathcal{H}} B$.[§]

With the above definitions we can formally prove the stated properties about the *old* transformation. For instance, to prove that $A \prec old(A)$ we build a hierarchy $\mathcal{H}$ such that, for any type T, $\mathcal{H}(T) = \{T, old(T)\}$. We have to prove that $\mathcal{H}$ is consistent. Suppose a type A. If X is a method of A, with signature

$$P_1 \times \ldots \times P_n \rightarrow R_1 \times \ldots \times R_m,$$

then there is a method X in $old(A)$ with signature

$$P_1 \times \ldots \times P_n \rightarrow old(R_1) \times \ldots old(\times R_m).$$

Clearly $R_i \prec_{\mathcal{H}} old(R_i)$, by construction of $\mathcal{H}$. If X is sef in A, X is also sef in $old(A)$. If X is non-sef in A, then X is an excluded method in $old(A)$. In any case, the subsignature relationship is assured.

The other properties can be proved in a similar way.

[§]It is not difficult to show that this definition is equivalent to assuming that A is subtype of B unless there is a contrary reason. This is the same as getting the largest solution to the recursive equations, instead of the usual least fixed-point solution.