INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Side Effect Free Functions in Object-Oriented Languages

Roberto Ierusalimschy* Noemi Rodriguez*

TR-94-066

December 1994

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.

*PUC-Rio, Brazil email: roberto,noemi@inf.puc-rio.br

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, procedural languages have to rely mainly on side effects to achieve their goals.

Functions in procedural languages, like Fortran or Pascal, can produce different kinds of side effects: there are changes of memory needed to produce a result, but there are also changes that go beyond the scope of the function, and thus can affect other parts of a program. A closer look gives rise to the identification of more subtle 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. This kind of function also provides *referential transparency*;

3. another kind of functions are those which do not change old values in the memory, but can create new objects. The real degree of side effects in these functions depends on the rest of the language. If the language provides an object equality operation[1], then such functions do not provide referential transparency, as objects created in different calls can be distinguished. If the language offers a way to know the current amount of free memory, then these functions can even produce real side effects;

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.

This paper proposes an algorithm to check whether a function is a *sef* function in the programming language School [RIR93b], [RIR93a]. School is an object-oriented language with separate type and class hierarchies, whose main goal is to achieve good flexibility with a secure static type system. 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, like in Smalltalk [GR83]. Types, on the other hand, are

[1] that is, pointer equality

used only by the compiler; a type represents the interface of an object. Every object belongs to a fixed class but can "satisfy" several different types.

In our proposal, a sef function cannot modify *pre-existent* objects, that is, objects created priorly to the function activation. Those objects are seen inside a function through formal parameters, instance variables, and the pseudo variable *self*. Pre-existent objects are viewed inside sef functions as having a type which does not allow the application of potentially harmful methods. This type is created through a transformation of the original type, called the *old* transformation. The resulting type allows only the application of sef methods, which on their turn may not return modifiable parts of the protected object.

Note that the described type conversions are carried out by the type checking routines, having no impact on runtime semantics.

Section 2 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 transformation is applied on types of objects used in sef methods in order to create views of these objects containing only harmless operations. Section 4 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.

2 Types in School

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 communication between objects are based on late-binding, and the binding depends solely on the receiver.

The rule that has guided the development of School's type system has been to avoid any construction that can cause a "message not understood 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.

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. Therefore, 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 [WZ88], 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.

The independence between subtyping and subclassing allows a programming language to adopt *structural compatibility*. That means that a type is a subtype of another one if they are compatible in some way.

Following our stated main rule, we want to allow a type A to be a subtype of B as long as there is no possibility of error when using an A object in the place of 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 arity $P_{B_1} \times \ldots \times P_{B_n} \rightarrow R_{B_1} \times \ldots \times R_{B_m}$, there is a method X in A, with arity $P_{A_1} \times \ldots \times P_{A_n} \rightarrow R_{A_1} \times \ldots \times R_{A_m}$, where for all $i \leq m$, $R_{A_i} \prec R_{B_i}$ and, for all $i \leq n$, $P_{B_i} \prec P_{A_i}$. If this condition is satisfied, we say that the arity of X in a is a subarity of the arity of X in B. The apparent inversion in the last condition is known as the "contra-variance rule", and is needed to assure correctness [CW85].

Notice that the above definition is not formal, since it can result in an infinite recursion if a type refers to itself. The formal definition of subtyping can be found in the appendix. A more complete treatment, as well as a formal proof that this definition avoids type errors, can be found in [Ier93].

The static verification of side-effect free functions implies in an extension of the type system defined above. 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.

3 The Concept of *Old* Type

The type operator *old* is applied on types of objects which are "older" than a function activation, 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 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.

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 arity

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

there is a method X in A, with arity

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

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

Again, it must be noted that the above definition is not formal, since it can result in an infinite recursion. 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*. A formal definition of subtyping, justifying this intuition, is presented in the appendix.

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 in the next section.

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 can not 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.

Another useful definition is of a sef type. A type T is sef if $sef(T) = T$. Examples of sef types are integers and reals. Moreover, for any type T, $sef(T)$ is a sef type.

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

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 method NoChange (A,B) -> (A,B);
  method 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 method NoChange (A,B) -> (old(A),old(B))
  excluded method 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 the previous section.

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

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

The type $A[old(B)]$ is defined by:

```
type A[old(B)] is
  sef method NoChange (A,U,old(B)) -> (A,U,old(B))
  method 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 method NoChange (A,U,B) -> (old(A),old(U),old(B))
  excluded method 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.

Intuitively, we may think of an object with type $A[T]$ as an object of type A which has references to objects of type T. Therefore, what this last property tells us is that a new object that points to old objects can be seen as an old object. Old objects can only point to other old objects, so when $A[T]$ is old there is no need to say that T is old too.

Finally, to see that, for any type A, we will have that $old(A)$ is a subtype of $sef(A)$, let us again consider our first definition of a general type A:

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

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

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

type sef(A) is
  sef method NoChange (A,B) -> (sef(A),sef(B))
end old(A)
```

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.

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 having the compiler apply the *old* transformation on the types of objects which are created prior to the method call, before carrying out the checking of the sef method code. In this section we specify how this is done.

A method declared as a sef method is checked by the compiler in the following way:

1. all types T for objects external to the method, i.e. parameters, instance variables, and *self*, are substituted for *old(T)*;

2. assignments to instance variables are not allowed;

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)*.

The first two rules are needed to ensure that a method is side effect free. The use of restricted views over the parameters, instance variables, and *self* is quite intuitive given our goal of protecting pre-existing objects. Note that the only extra rule which is effectively needed to ensure that a method is side effect free is the one that prohibits assignments to instance variables. This is one point we consider important in this proposal: the same set of rules used for type checking in general is the 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.

The last two rules are introduced solely with the purpose of increasing flexibility.

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. Note 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. The importance of this rule will be discussed in the next section. 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 returned object are part of the calling method, and will be ruled out in the checking of that method, if necessary, as discussed in the examples.

Note 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.

Motivation for rule 3 is given for situations such as the one shown in the following fragment:

```
class A is
  ...
  sef method M1 (b: B) -> B is
    Result := (self.M2(b));
  end M1
  sef method M2 (b: B) -> B is
    ...
  end M2
end A
```

Using only the first two rules in our checking algorithm, the call to *self.M2* in method *M1* would be illegal, since what $M2$ expects is a parameter of type B, and b is regarded as having type *old(B)*. Besides representing a common programming pattern, this is clearly safe, and to make it legal a special rule is added.

```
class queue[T] is
      inherits from class array[T]
  ...
  sef method Front() -> T is     -- OK!!
    Result := self.GetLow();
    -- returns first object in queue
  end Front
end queue;

  sef method M(sleeping: queue[Process]) -> T1 is
  var
    first: Process;
    first := sleeping.Front();    -- illegal!!
    if first.TimeLeft() = 1 then
      -- wake up the first process in the queue
    else
      first.DecrementTimeLeft()
          -- affects the actual process in queue
    end
    ...
  end M
```

Figure 1: A simple example of sef methods

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 two simple examples which illustrates the basics of the proposed mechanism. The third and last example shows more of its power.

Figure 1 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. Note that *GetLow* would typically be 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 possible 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

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

Figure 2: Method *Min* returns the least of two objects

actions which are considered illegal in a sef method. A local variable *first* is defined with the same 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, may only be used explicitly in the declaration of local variables. This is because the concept of *old* types is local to each method call, and does not make sense out of this context.

The variable *first* could thus 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*.

We now discuss an example where we make use of the two rules added for flexibility. Figure 2 shows a method which takes two parameters of a same type *Comp* and returns the one which is least 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

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

Figure 3: Method *Clone* returns an object of the wrong type

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 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 3 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)*, and method $M2$ would not be part of *old(A)*, so 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 4 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 sef method *Reverse* which returns as its result a list which is the reverse of the receiver. Note that the new list is created by parameterizing class *LinkedList* with type *old(T)*. (The syntax for creating new objects in School is a class name followed by an exclamation mark and the name of a constructor.) This means that method *revList.Append* will only accept objects of type *old[T]* as a parameter. This is exactly the type of object that is retrieved

```
type List[T] is
  method Append(T)
  sef method Reverse () -> List[T]
  ...
end List;

class LinkedList[T]: List[T] is
  ...
  constructor New() is
    ...
  end New

  sef method Reverse() ->  List[T] is
  -- returns a new list which is the reverse of self
    var
      i: Integer := self.Length;
      revList: List := 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 4: Creating a new list from an existing one

by the call *self.Get(i)*, so the call to *Append* is correctly typed.

One interesting point in *Reverse* is the type checking of its return statement. 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)]*.

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 [Car89], 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 still do not 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 this paper. We would also like to thank the International Computer Science Institute, where a major part of this work was developed.

References

[Car89] L. Cardelli. Typeful programming. In *notes of IFIP Advanced Seminar on Formal Description of Programming Concepts*, Petropolis – Brazil, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and its Implementation.* Addison-Wesley, 1983.

[Ier93] R. Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer Languages*, 19(1):19–40, 1993.

[RIR93a] 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.

[RIR93b] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.

[WZ88] 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
  method F (A) -> B;
end A

type B is
  method 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 arity
> $$P_1 \times \ldots \times P_n \rightarrow R_1 \times \ldots \times R_m,$$
> there is a method X in A, with arity
> $$Q_1 \times \ldots \times Q_n \rightarrow S_1 \times \ldots \times S_m,$$
> 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 [Ier93], 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*.

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

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 arity

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

there is a method X in A, with arity

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

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_B is a sef method, X_A must also be a sef method. Furthermore, if X_B is not excluded, X_A can not be excluded.

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 $A \prec_{\mathcal{H}} B$.[2]

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 arity

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

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

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

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.

[2] 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.