Of first-class methods and dynamic scope

Theo D'Hondt — Wolfgang De Meuter

Programming Technology Laboratory
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{tjdhondt,wdmeuter}@vub.ac.be

ABSTRACT. *When considering the wide range of object-oriented programming languages, one hardly ever finds methods to be first-class entities. At first sight, this phenomenon seems to be caused by a concern for an efficient implementation. Closer inspection however, reveals more subtle grounds that are rooted in issues more fundamental than performance. This paper investigates this aspect of object-oriented programming languages using an extensible object model that is sufficiently simple to reveal the various concerns. In particular, it argues in favor of dynamic scoping as a setting in which to manipulate first-class methods.*

RÉSUMÉ. *Parmi l'éventail très large de langages orientés objet, la notion de méthodes de première classe est en général absente. A première vue, ce phénomène semble être causé par un souci d'efficacité. Une étude plus approfondie dévoile néanmoins des raisons plus subtiles, ayant une origine dans des considérations plus fondamentales. Ce papier examine cet aspect de langages de programmation orientés objet en utilisant un modèle objet extensible qui est suffisamment simple pour exposer les soucis divers. Nous y proposerons en particulier de choisir pour la portée dynamique de variables comme contexte optimal dans lequel utiliser des méthodes de première classe.*

KEYWORDS: *object models, first class values, scoping.*

MOTS-CLÉS : *modèles objet, valeurs de première classe, portée de variables.*

1. Introduction

Consider the programming language Smalltalk. Many view it as a paragon of object-oriented programming, bringing together in one elegant and expressive package most of the things that objects are supposed to represent. Although largely supplanted by Java — for reasons that are beyond the scope of this paper — it is still very much alive and in use. It was conceived thirty years ago but it continues to be a remarkable example of language design and implementation. One of the few faults that we can find with Smalltalk is the dichotomy between methods and closures: methods are syntactical structures (barely accessible at the meta-level) while (block) closures are true objects. Considering other programming languages, we find similar limitations: methods are purely syntactical (e.g. Java), addressable (e.g. C++) or convertible to first-class values (e.g. CLOS). Using the same languages as examples, we find that closures can be simulated by class nesting (e.g. Java), limiting scope nesting (e.g. C++) or using conversion operators (e.g. Common Lisp). Two of the few exceptions treating methods as objects are Self [SLF 87] and NewtonScript [NWS 95]; we refer to them here because both are prototype-based languages and additionally, NewtonScript also supports frames. But more of this later.

We would like to explore the notion of first-class methods in a controlled environment. To this end we use an experimental language framework, described in the next section. It will be possible to navigate an extremely simple design space and zoom in on a minimal but stable object model that supports first-class methods. By *object model* we denote an interpreter for some object flavor that can conceivably be translated into some formalism (e.g. denotational semantics). It will on the one hand offer a sufficiently strict view on the semantics of the model; on the other, it will allow us to investigate the model's expressiveness by experimenting with some of its concrete instances.

We shall see that we will simplify things by relaxing the inheritance strategy (hence our preference for a prototype-based model) and we will also want to lift any kind of differentiation between attributes and methods (hence the interest in frame-based models). We will also need to investigate the reentrancy properties of our objects. In fact, in section 4 we will associate the method-sharing in more conventional object-oriented languages with properties of (im)mutability of variables in our model.

Finally, we want to conclude this introduction with a final question: is a concern for first-class methods more than purely academic? Is there any response beyond *small is beautiful*? We have no intention here to explore the full range in expressiveness of first-class methods, but the least we can do is state a case in their favor. Let us therefore (re)consider the block-context/inner class constructs mentioned earlier on. These happen to occur in two of the most widely used object-oriented programming languages. Hardly anyone is impressed by either of these constructs as examples

of elegant language design and as a matter of fact they can be viewed as *remedial*. They are often used to act as closures — quoted and parameterized expressions paired off with a static environment — in a context where first-class methods are indicated. Consider a functional programming style, which is often encountered in a collection hierarchy. For instance in Smalltalk the do and collect are the counterparts of the Scheme foreach and map operations, but on general sets rather than on lists. Both do and collect require a closure argument in the form of a BlockContext instance as in:

someSet collect: [:item | ... *do something with* item ...]

where the collect: message instructs the object someSet to apply the content of the block to each of its items. Within the context of an object-oriented language such as Smalltalk, it would make more sense to define the block as a method, for instance:

someMethod: item
...*do something with* item ...

and then transmit someMethod: as an argument to the collect: message. In languages like C++, a low-level approach is used (i.e., using a reference to a member function), but in Smalltalk it is a lot more complicated to find a solution that does not violate the clean semantics of the language. Several object-oriented languages, such as Java, Smalltalk and Self, allow the manipulation of messages via the meta-object protocol, but this is fundamentally different from having first-class methods — and generally much less expressive.

2. The Pico language model and virtual machine

The Pico language framework [PIC] was originally developed as a teaching environment, putting into practice such various topics as language design, grammars and semantics, memory management and interpreter techniques. It is intended as an alternative to Scheme as used in e.g. [EOP 01], but with an explicit interest in computational and storage models. It was rapidly adopted to function as a framework for research into reflective virtual machines and strong mobility [BRG 00]. Pico was designed with a very tight but expressive abstract language kernel and a simple and extensible language front-end. Consider as an example the following code fragment:

```
QuickSort(V,Low,High):
  { Left: Low; Right: High;
    Pivot: V[(Left + Right) // 2];
    Save: 0;
    until(Left > Right,
      { while(V[Left] < Pivot,
          Left:= Left+1);
        while(V[Right] > Pivot,
```

```
            Right:= Right-1);
          if(Left <= Right,
            { Save:= V[Left];
              V[Left]:= V[Right];
              V[Right]:= Save;
              Left:= Left+1;
              Right:= Right-1 }) });
    if(Low < Right, QuickSort(V, Low, Right));
    if(High > Left, QuickSort(V, Left, High)) }
```

It implements the well-known quicksort algorithm — the program:

```
QuickSort(V[100]: random(), 1, size(V))
```

performs an in-line sort of a hundred randomly chosen numbers. It should be sufficiently recognizable to develop a first impression of the language.

Pico was developed very much in the spirit of Scheme, but with a conventional syntax (featuring infix operators and a canonical notation for function application), tables instead of lists and no special forms. The latter is the result of using an Algol-like call-by-name binding mechanism and it results in a standard function format for all control structures. In Pico all language elements are first-class (including procedures [1], declarations, environments and continuations) and the abstract grammar consists of essentially five productions (references, applications, tabulations, definitions and assignments). A metacircular specification of the evaluator numbers 220 lines but despite the fact that no lexical addressing is used (environments are simple linked lists of bindings) an optimized continuation-passing-style virtual machine was constructed that matches Scheme implementations (e.g. DrScheme [DRS]) in efficiency.

The Pico framework proved to be an excellent environment in which to experiment with languages and paradigms. It was for instance extensively used in the *European Master in Object-Oriented Software Engineering* [EMO] program to compare class-based and prototype-based inheritance.

3. A simple object model

Since environments are first-class Pico values, they can easily act as objects: it suffices to add a native function `clone()` to clone the current environment:

1. We use the Scheme term *procedure*, otherwise known as closure, for the value of a lambda-expression in a statically scoped language; it typically adds the defining environment to the lambda expresion.

```
counter(n):
  { incr(): n:= n+1;
    decr(): n:= n-1;
    clone() }
```

An expression of the kind `c: counter(0)` will define a variable `c` bound to a counter object with an attribute `n` initialized to zero and two methods `incr()` and `decr()` to change the state of the object. These will be stored in the closure that is created when the function `counter` is applied. An extension of the Pico machine, involving name qualification (by means of the ubiquitous dot-qualifier) is readily implemented as qualified name lookup in a Pico environment and results in a simple (but expressive) message passing semantics:

```
c: counter(0);
d: counter(100);
(c.incr())+(d.decr())
```

In order to introduce inheritance, we opted for nested mixin methods (see e.g. Agora [AGO 94]). In order to see what mixin methods (also referred to as *modular inheritance*) consider the example below.

The same `clone()` function allows us to extend the `counter` generating function with a mixin `protect(limit)` such that `p: c.protect(2)` produces a counter object that limits its state to the interval [-2,2]. Note the presence of an extra `super` attribute and the assignment to it of the cloned value, in order to allow super sends.

```
counter(n):
  { incr(): n:= n+1;
    decr(): n:= n-1;
    super: void;
    protect(limit):
      { incr():
          if(n = limit,
            error("overflow"),
            super.incr());
        decr():
          if(n = -limit,
            error("underflow"),
            super.decr());
        clone() };
    super:= clone() }
```

We have effectively introduced a prototype-based language that allows for the generation of an hierarchy of prototypes, constructed by the successive application of

nested mixins and that exhibits an overriding semantics that can be found in most object-oriented systems. In order to avoid having to apply the same mixin over and over again each time we want to instantiate an object, we have also provided a variation `clone(object)` to the cloning function that clones a given object (the actual meaning of cloning will be one of the topics in the next section).

Note that our model suffers from the problems that are inherent to the nested mixin approach: the design of a prototype hierarchy requires prescience on the part of the designer. Mixins must be defined a priori and any extension of some hierarchy requires an intervention in existing code. Solutions exist but are beyond the scope of this paper.

Another issue that we should resolve is the one about self-reference: we have adopted a Java-style notion of `self` which is linked to the absence of qualification, but the semantics are as yet unclear. On the other hand, and without really trying, we have covered the first part of this paper's title: first class methods. Indeed, consider the following example:

```
object(variable):
  { method(argument):
      argument+variable;
    export():
      method;
    clone() }
```

An invocation of `m: object(123).export()` produces a procedure `m` that implements the method `method` in combination with the attribute `variable` the value of which is `123`. Actually, the same result is obtained by qualifying the method as a variable, as in `m: object(123).method`. In both cases evaluating `m(876)` will produce the value `999`.

The second part of the title, referring to dynamic scope, is what the following section is about. Indeed, we have been naive in thinking that first-class methods can be implemented as procedures. Strangely enough, the most elegant way to tackle this problem passes through the need for code sharing, otherwise known as reentrancy. It also goes hand-in-hand with the architecture of an environment.

4. Reentrancy and dynamic scope

Probably the most important reason why class-based inheritance is so popular is because it can so easily and efficiently be mapped to static types. A less appreciated (but at least as important) advantage lies with code sharing. Classes constitute a specification of the behaviour of a family of objects, and this specification is moreover

incremental, through the concept of inheritance. But classes are also a repository for the implementation of common behaviour and in most object-oriented languages this consists of the reentrant code of the various methods. Reentrancy is typically obtained by reusing technology to support recursion and equipping methods with a hidden *self* parameter. On the downside, shared code should not be modified and therefore most object-oriented languages implement methods as syntactic structures that are immune to side-effects.

In order to give a tangible meaning to all of the preceding observations, we shall investigate how they apply to the model introduced in the preceding section. First of all, we need to have a closer look at *environments* as the prime support for an object. The original Pico environment is a simple list of bindings, as is represented in figure 1.

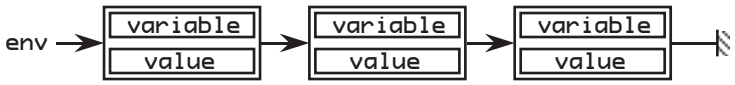

Figure 1. *Environment layout*

This is arguably the most simple semantics one might imagine for an environment. Adding a variable implies pushing a new binding onto the list, while looking up a variable or changing its value implies a sequential search through the list. The treatment of homonyms is unambiguously defined (and in the object model used to support overriding), but obviously, in the absence of frames [2] , it is impossible to eliminate them from a single scope level. When a function is called, it suffices to make a shallow copy of the function's static environment — as stored in the procedure — to implement static scoping and sharing of those non-local variables that should be visible to the function. The actual Pico implementation, using smart caching, proves that this approach does not raise impossible performance issues, although variable lookup cannot be but the most time consuming part of the virtual machine. On the other hand, it allows us to reason about the execution model in a simple way without losing the benefits of an operational system.

Consider the naive model we introduced in the previous section. An object, as represented by such an environment, is composed of bindings that associate variables with their values. We can arbitrarily distinguish methods from attributes depending on whether these values are procedures or not. In the example of the protected counter in the previous section, the attribute `n` is shared by both `c` and `p`; on the other hand, the methods `incr` and `decr` are truly overridden when we go from `c` to `p`. However,

2. *Frame* is used to refer to a grouping within an environment of parameters and local variables belonging to a single procedure activation.

a proper definition of a `clone(object)` function will not be possible without a profound revision of our model.

The problem lies with the fact that in a statically scoped language with first-class procedures, environments are recursive [3]. Consequently, if an application of `clone(object)` is defined as a deep copy, as it necessarily must be, this implies a computation of the transitive closure of `object`. Indeed, if we do not equally clone the environments nested inside the procedures that we have identified as methods, we will never have objects with a unique state. But this in turn leads to the observation that we will never have method sharing in our model. The only way out of this seeming contradiction is a revised definition of a *procedure* and as we shall see, the introduction of dynamic scope. In fact, the declaration of:

```
incr(): n:= n+1
```

will as of now result in a procedure, composed of an empty parameter list and the body `n:= n+1`, but *without* a static environment, bound to the variable `incr`. The counter object `c` — read: environment — containing this binding will upon reception of a message `c.incr()` have to activate the procedure bound to `incr` with respect to itself — read: self. In other words, during function application — read: message — all non-locally bound variables will be looked up in the receiver.

Our revised model has identical semantics to the original one, except in the case of a self-send, i.e., in the absence of an explicit receiver. In this particular case self is bound to the current environment and the above process implies nothing short of dynamic scope. Indeed, any self-send activating a method that refers to a non-locally bound variable will result in this variable being looked up in the calling environment and not in the defining environment of the function that implements the method.

```
make(x):
  { get(): x;
    extend():
      { x: 0;
        set(y): x:= y;
        clone() };
    clone() }
```

An interesting new feature that can be derived from this modified model is that of *variable overriding* — not to be confused with *variable shadowing* as in Java. Consider the following — artificial — application of the example above.

---

3. A procedure contains a reference to the static environment in which the corresponding function was defined; any assignment of the procedure consequently introduces a recursion.

Imagine a prototype is created by performing `p: make(1)`; imagine moreover that a second prototype is derived by means of `q: p.extend()`. Because `x` was overridden, a call to `q.get()` produces the value `0`. However, `q.set(2)` will have an effect on `q` only.

Let us return to the subject of code sharing since in the modified model we now effectively dispose of reentrant methods. We could consider using a shallow copy to re-implement the `clone(object)` function but this would be incompatible with the use of destructive operations. Indeed, any assignment of a variable — method or otherwise — would impact all of the clones, which contradicts the notion of individual state. We will therefore need to introduce *immutable* entries in an environment, similar to constant declarations in Java, called constants as opposed to variables:

```
<constant>:: <expression>
```

The `<constant>` will be initialized with the value of `<expression>` and can consequently be used in any expression except:

```
<constant>:= <expression>
```

Constant entries in an environment are candidates for sharing, hence we require an updated architecture for environments (see figure 2). In all honesty we should add that with this approach, resolving homonyms is less than satisfactory [4]. However, for the purpose of this discussion, the proposed architecture is sufficient.

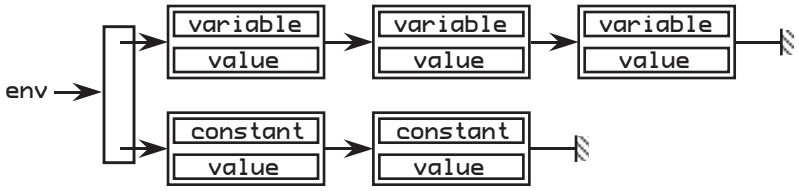

Figure 2. *Updated environment layout*

Armed with the updated environment from figure 2, and as described in figure 3, we will redefine the `clone(object)` operation, i.e., a deep copy of the variable part and a shallow copy of the constant part.

This effectively provides sharing of methods, provided they have been declared as constant. Moreover, this holds for any constant, irrespective of whether it is bound to

4. In the presence of a constant and variable with the same name, one will arbitrarily — depending on the lookup strategy — hide the other. In order to solve this, frames are required.

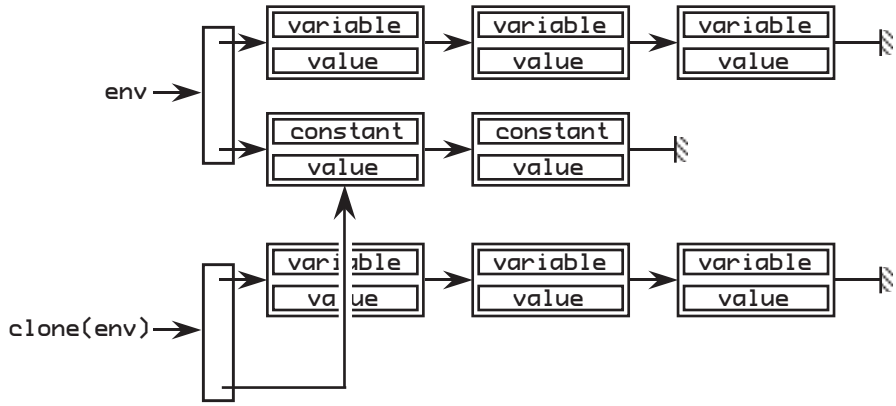

Figure 3. *Cloning an environment*

a scalar, procedure, table [5] or environment.

We need to cover a last topic, namely *closures.* Indeed, we need to establish what happens when a first-class use is made of a method, as in:

```
method(argument):: ...
      ...
variable: method;
      ...
function(method);
      ...
variable:= method;
      ...
function(): method;
```

We have opted for a conservative approach, automating what is practice in Common Lisp. Upon retrieval from an environment, a procedure value is implicitly converted to a closure by including the environment. Hence, in the program on the next page, the value 2 is displayed. It is clear that other strategies are available to us, but their exploration would — again — be beyond the scope of this paper.

```
{ bean(x):
    { get():: x;
      set(y):: x:= y;
      freeze():: get;
```

5. Note that a constant table's slots can be modified through assignment.

```
      clone() };
  b: bean(1);
  c: b.freeze();
  b.set(2);
  display(c()) }
```

The following example describes a `Stack(n)` mixin containing a nested mixin `makeProtected()`. They are expressed in a variation of Pico, called Pic% [6], based on the concepts introduced earlier in this section [7]. In order to have a workable object-oriented language, a `self()` native function was introduced that returns the current receiver. Moreover, an extension was made to the message passing syntax: anonymous dot-qualification denotes a super send.

```
Stack(n):
  { T[n]: void;
    t: 0;
    empty():: t = 0;
    full():: t = n;
    push(x)::
      { T[t:= t+1]:= x;
        self() };
    pop()::
      { x: T[t];
        t:= t-1; x };
    makeProtected()::
      { push(x)::
          if(full(),
            error("overflow"),
            .push(x));
        pop()::
          if(empty(),
            error("underflow"),
            .pop());
        clone() };
    clone() }
```

This example mimics a traditional object-oriented style: `n`, `T` and `t` are *attributes* while `empty`, `full`, `push` and `pop` are *methods*. The first two are *inherited* and the other two are *overridden*. It should be noted that Pic% is at least as expressive as, say, Smalltalk — but it is a true multi-paradigm language. In our quest for first-class methods we produced a language model that effectively smoothes the transition between the functional and the object-oriented style.

---

6. Object-oriented = o/o = %, for want of a better name.
7. This includes the suggested frame based environments.

5. Conclusion

We live in an age were the vast majority of software developers believes that a modern programming environment is necessarily statically typed, class-based and garbage-collected. This conviction is moreover spreading to the academic community as witness the technical presentations at many of the conferences on software technology. So many will view this paper as an academic exercise, and to a certain extent it is true. But there is more.

We have reported on the conception and development of a simple but expressive object model that incorporates the notion of first-class methods. It is clear that a full report on the details of such a venture exceeds the scope of this paper, but we hope we have succeeded in presenting a convincing synopsis of the various issues — and their solution. Anyway, the resulting model is consistent and complete and although we do not claim that it is unique, anyone would be hard put to improve on it. Note that the model does not consider first-class methods from a reflection perspective — which is what is involved with languages such as Self.

The Pic% model provides a number of insights in the architecture of an object-oriented programming language — again we don't claim that this is unique, only that it is probably more accessible than other approaches, which are either more formally or more technically slanted. In particular, Pic% can be viewed as a proof by construction of the existence of a stable object model with first-class methods. It clearly establishes the relationship with closures and provides an environment in which functions and methods are seamlessly integrated. It might be possible to use a similar approach to tackle more challenging paradigms: the ad-hoc way in which NewtonScript uses slots in which to store methods might be used as an inspiration for the integration of functional programming, object-oriented programming and frame-based programming.

One of the more remarkable conclusions from our experiment shows that the introduction of first-class methods — as a coherent and stable given in an object-oriented system — requires a return to dynamic scope. Since dynamic scope has been as good as dead these past 25 years — remember that Scheme is Lisp with static scope — this is something of a surprise. At the very least it raises an interesting question: is static scoping compatible with object orientation or is its inclusion in modern object-oriented languages an artifice?

At the very least the Pic% model is a useful experimentation tool as witness the results with three generations of EMOOSE students. It has also proven to be an efficient *sandbox* in which to *play* with programming language features. Some interesting developments show that this in not at all academic: consider the notion of *domain specific languages* which has steadily been gaining attention (an interesting survey is

available on [DSL 02]); also consider emerging tools such as for instance JScheme [JS]. Language engineering is alive and kicking and environments such as Pico are its workbench.

6. References

[AGO 94] CODENIE W., DE HONDT K., D'HONDT T. , STEYAERT P., *Agora: Message Passing as a Foundation for Exploring OO Language Concepts*, SIGPLAN Notices, 29(12):48-57, December 1994.

[BRG 00] VAN BELLE W. , D'HONDT T., *Agent Mobility and Reification of Computational State, an experiment in migration, in: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, Springer Verlag Lecture Notes in Artifical Intelligence 1887, June 2000.

[DRS] *The PLT Scheme programming environment*, see `http://www.drscheme.org`.

[DSL 02] VAN DEURSEN A., KLINT P. , VISSER J., *Domain-Specific Languages: An Annotated Bibliography*, see `http://www.cwi.nl/ arie/papers/dslbib`.

[EMO] *European Master in Object-Oriented Software Engineering* , see `http://www.emn.fr/emoose`.

[EOP 01] FRIEDMAN D. P., WAND M. , HAYNES C. T., *Essentials of Programming Languages*, MIT Press (ISBN: 0-262-06217-8) 2nd edition, 2001.

[JS] ANDERSON K., HICKEY T. , NORVIG P., *The Jscheme Web Programming Project*, see `http://jscheme.sourceforge.net/jscheme/mainwebpage.html`.

[NWS 95] SMITH W. R., *Using a prototype-based language for user interface programming: the Newton project's experience*, ACM SIGPLAN Notices, Proceedings of the tenth annual conference on Object-Oriented Programming Systems, Languages, and Applications Volume 30 Issue 10, October 1995.

[PIC] D'HONDT T., *The Pico Programming Project* , see `http://pico.vub.ac.be`.

[SLF 87] UNGAR D. , SMITH R. B., *Self: The power of simplicity*, SIGPLAN Notices 22(12), December 1987.