A Layered Calculus for Encapsulated Object Modification

Kim Mens, Kris De Volder, Tom Mens

Department of Computer Science, Faculty of Sciences

Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium

E-mail: { kimmens | kdvolder | tommens }@vub.ac.be

Abstract. *Current prototype-based languages suffer from an inherent conflict between inheritance and encapsulation. Whereas encapsulation tries to hide implementation details from the user, inheritance depends at least to some extent on exposing these implementation details. We propose a powerful calculus with dynamic object modification which solves this conflict. This calculus constitutes a formal foundation of prototype-based languages with a clean interaction between encapsulation and inheritance.*

1 Introduction

Two essential concepts in object-oriented programming are inheritance and encapsulation. Inheritance allows building new objects or classes by incrementally modifying existing ones. Encapsulation provides objects with abstraction barriers behind which implementation details are hidden from the user. As inheritance depends at least to some extent on these implementation details [7], there is an inherent conflict between inheritance and encapsulation.

Class-based languages solve this conflict by introducing the notion of "encapsulated inheritance" [7]: inheriting clients have no direct access to the private attributes of their parents. However class-based languages are sometimes considered too rigid, e.g. when dynamic evolution of an object's behaviour is required [5]. Prototype-based languages are more flexible and expressive and therefore well-suited for rapid prototyping and exploratory programming. Unfortunately this flexibility comes with a serious loss of safety. Since prototype-based languages define inheritance directly on objects rather than classes, encapsulated inheritance cannot prevent all violations of an object's encapsulation barrier, as observed in [9] and confirmed in [4] and [5].

To illustrate the problem, consider the following example in a C++ like syntax, but featuring inheritance on objects rather than classes. A bank account is implemented as an object containing methods for withdrawing and depositing money. Withdrawal is secured with a password.

```
Object Account {
   Private:
      amount = 5000;
      password = "007";
   Protected:
      verify(pwd) { return (password==pwd); }
   Public:
      deposit(val) { amount=amount+val; }
      withdraw(val,pwd) { if (this.verify(pwd)) amount=amount-val; }
};
```

Inheritors can use the protected `verify` method to create specialised versions of password verification. Although it is important to hide this information from message sending clients, current prototype-based languages cannot enforce this. Fraudulent "message sending" clients can gain access to implementation details by temporarily turning into inheritors and creating their own specialised version of an object. The example below illustrates how this technique can be used to steal money from the `Account` object.

```
Object Fraud {
      Private:
         Object ForgedAccount : Inherits Account {
            Protected:
               verify(pwd) { return true };
         };
      Public:
         steal(amount) { ForgedAccount.withdraw(amount,"?"); }
};
Fraud.steal(5000);         // steal some money
```

Such malicious practice cannot occur in class-based languages, where inheritance is not defined on objects directly, but on distinct inheritable entities called classes. Classes can be instantiated to form objects that can only be sent messages and cannot be further specialised.

The challenge with prototype-based languages is to remove the inheritance-encapsulation conflict without sacrificing their flexibility. We tackle this problem by providing a formal calculus with dynamic object modification and a clean interaction between encapsulation and inheritance. To highlight the essence of the model features such as typing, object identity, state and private attributes have not been included.

2 A Layered Calculus

Essentially the problem in prototype-based languages is that inheritance and message sending are both performed on objects. Analogous to class-based languages, this can be solved by making a distinction between objects for message sending and "inheritable entities" —called *generators* [3]— for specialisation. We show that this distinction does not necessarily sacrifice flexibility.

The proposed calculus has a two layered syntax, clearly distinguishing generator expressions from object expressions. The top layer deals with objects and message sending. The second layer deals with generators and inheritance with late binding of self. Due to the layering and a careful scoping of generator names, the use of generators is restricted to the inside of an object so that encapsulation cannot be breached. In spite of this restriction it is still possible to model several mechanisms for (encapsulated) dynamic object modification.

2.1 Syntax

The top layer of the syntax is given by the following production rules. Terminal symbols are written in boldface. Identifiers (`Ident`) are also considered to be terminals.

Object	→	Object**.**Ident**(**Object**)**	*message sending*
	|	**[** Generator **]**	*object creation*
	|	Ident	*argument reference*

Objects are created from "generators". Upon creation of an object its generator is encapsulated inside the object, hiding information only important for inheritors behind the object's message sending interface.

The second layer of the syntax deals with generator expressions and specialisation:

Generator	→	Generator **;** Generator	*composition*
	|	Ident**(**Ident**)=**Ident**#**Object	*method*
	|	**>** Object **<**	*object to generator conversion*
	|	Ident	*self reference*
	|	ε	*empty generator*

Generators are specialisable entities from which objects can be created. The most primitive non-empty generators are single method descriptions. They can refer to a late-bound "self" generator via the name assigned by the `#` binding operator[1]. Informally, `m(a)=Self#`*`body`* means that the name `Self` can be used inside the body of `m` to denote self references. Inheritance can be accomplished through generator composition. Composing generators redirects their late bound self to the resulting composed generator.

The "**>…<**" operator "converts" an object into a generator. This provides some extra flexibility in dynamic object modification. Care must be taken however in defining the semantics of this operator. For example, a semantics that returns the encapsulated generator would reintroduce the encapsulation problems we are trying to avoid.

Here is a simple example[2] of a well-formed object expression. It denotes an object containing a method `letterhead` that performs a self send of the `title` message:

---

[1] Neglecting syntactic differences, this binding operator serves the same purpose as the ς() operator defined in [1].

[2] Although not explicitly present in the syntax, we will use predefined strings (understanding `add` and `equal`), numerals (understanding `equal`, `add` and `subtract`) and booleans (understanding `if` and `ifTrue`) to make more meaningful examples.

```
[     title(dummy) = Self # "Mr. ";
      letterhead(name) = Self # [Self].title([ε]).add(name)       ]
```

In what follows, we assume some conventions to make examples more concise and readable. To avoid confusion we start argument references with lowercase and references to generators with uppercase letters. We omit the # binding operator when no reference to the self generator is made inside the method body. A message send without actual argument is considered equivalent to a message with the empty object [ε] as argument. We omit the formal argument from a method definition when it is not referred to in the method body. Using these conventions the example can be written more concisely:

```
[     title = "Mr. ";
      letterhead(name) = Self # [Self].title.add(name)     ]
```

2.2 Denotational Semantics

In this section we use a denotational semantics to validate our claim that an object's encapsulation boundaries cannot be breached through inheritance. We use the notation of [6], extended with square brackets for parameterised domains.

Syntactic Domains

```
ObjExpr        =      set of all syntactic object expressions
GenExpr        =      set of all syntactic generator expressions
Ident          =      set of all syntactic identifiers
```

Semantic Domains

An `Object` is represented as a record of methods. Each `Method` expects an `Object` as argument and returns an `Object` after evaluation.

```
Record[α]      =      Ident → α ⊕ Unit
Object         =      Record[Method]
Method         =      Object → Object
```

The above representation of objects guarantees object-based encapsulation because properties of an object that are not accessible through its message sending interface are not manifest in the representation either. Another consequence of the object representation is that inheritance is not possible on objects. Instead inheritance is accomplished indirectly via generators.

```
Generator      =      Generator → Object
```

To allow late binding a generator is a template for an object with a still undetermined (late bound) self. It is a function mapping a self `Generator` onto an `Object`.

Wrapping a generator transforms it into an object by self applying the generator. The resulting object can internally manipulate its self generator. This generator represents an unencapsulated version of the object on which inheritance is still possible. Externally however, the object is encapsulated.

```
wrap : Generator → Object
wrap g = g g
```

Scoping of generator names and argument names

To obtain lexical scoping of argument and generator names, both the semantics of object and generator expressions pass around two records containing the bindings for argument and generator names in their lexical environment. In the semantic equations the names `a` and `g` will be used for these environments of argument objects and self generators respectively.

Also in the semantics below, `{}` denotes the empty record, `{key→val}` a single slot record, $r_1 +_r r_2$ right preferential record concatenation, and `lookup r I` denotes the selection of identifier `I` in record `r` (which is undefined $\perp$ when `I` does not occur in `r`).

Semantics of an Object Expression

The semantics $[\![\ldots]\!]_O$ of an object expression is a function parameterised with the two lexical environments and returning an object. The semantics of message sends or references to formal arguments is straightforward. An encapsulated object is created from a generator by wrapping this generator.

$[\![\textbf{ObjExpr}]\!]_O$: Record[Object] → Record[Generator] → Object

$[\![O_r.I(O_a)]\!]_O\ a\ g$ = (lookup ($[\![O_r]\!]_O\ a\ g$) I) ($[\![O_a]\!]_O\ a\ g$)
$[\![I]\!]_O\ a\ g$ = lookup a I
$[\![[G]]\!]_O\ a\ g$ = wrap ($[\![G]\!]_G\ a\ g$)

Semantics of a Generator Expression

The semantics $[\![\ldots]\!]_G$ of a generator expression is similar to that of an object expression. It is again a function requiring two lexical environment parameters but it returns a generator rather than an object.

$[\![\textbf{GenExpr}]\!]_G$: Record[Object] → Record[Generator] → Generator

The semantics of a composition of generators is a new generator of which the self is distributed over its constituents. The semantics of a self reference and an empty generator are straightforward.

$[\![G_1;G_2]\!]_G\ a\ g$ = λself.($[\![G_1]\!]_G\ a\ g$) self $+_r$ ($[\![G_2]\!]_G\ a\ g$) self
$[\![I]\!]_G\ a\ g$ = lookup g I
$[\![\varepsilon]\!]_G\ a\ g$ = λself.{}

A method generator augments the lexical environments with bindings of the actual argument and late bound self to the appropriate identifiers. Upon invocation, the method is evaluated in these environments.

$[\![I_m(I_a)=I_s\#O_{body}]\!]_G\ a\ g$ = λself.$\{I_m \rightarrow method\}$
where $method = \lambda arg.[\![O_{body}]\!]_O\ (a +_r \{I_a \rightarrow arg\})\ (g +_r \{I_s \rightarrow self\})$

The "`>…<`" operator turns an object into a generator of which the self argument is ignored. A message sender can extend an object `O` by turning it into a generator `>O<` and subsequently composing it with some other generator. This is not really inheritance and does not breach encapsulation because it does not involve late binding of self in the object under extension.

$[\![>O<]\!]_G\ a\ g$ = λself.$[\![O]\!]_O\ a\ g$

Summarising

From the above semantics follows that the calculus indeed respects encapsulation boundaries of objects. The representation of an object as a record of methods exposes no more than the functionality of a message send. More specifically, `Object` and `Method` have no provisions for late binding. Inheritance with late binding can only be achieved by composing `Generator`s. Although objects can be converted to generators and vice versa (using `>…<` and `[…]` respectively), care has been taken that these conversions do not compromise encapsulation.

3 (Encapsulated) Dynamic Object Modification

In this section we illustrate that the proposed calculus is still expressive enough to model several mechanisms for dynamic object modification.

3.1 Encapsulated Inheritance on Objects

Inheritance with late binding of self can be modelled by adding (composing) methods directly to an object's self generator. Since self generators are only visible to code inside the object, an object can only specialise itself or any of its surrounding objects (due to the lexical scoping of generators). Other objects do not have access to any of its implementation details.

As a concrete example, consider a person object with attributes `name`, `sex` and `title`. When the message `letterhead` is sent to the object, the name is returned with the correct title prefixed to it. The message `newPerson` is used for modifying the original object to create a new person with a similar behaviour.

```
[ name    = "Ann Ticipate";
  sex     = "Female";
  title   = Self # [Self].sex.equal("Female").if([then="Miss ";else="Mr. "]);
  letterhead = Self # [Self].title.add([Self].name);
  newPerson(init) = Self # [ Self; name = init.name; sex = init.sex ]
]
```

The `title` method anticipates the overriding of the `sex` attribute by performing a self send to it. From the viewpoint of a message sender this is only an implementation detail. For inheritors however it is important information. Since objects only contain information important for message senders, inheritance must be performed on an object's generator which is only accessible inside the object. The `newPerson` method for example uses inheritance on its receiver's self generator to override the `name` and `sex` attributes. In doing so it actually depends on the `title` and `letterhead` method's self sending behaviour.

Inheritance schemes such as the one above where an object is modified indirectly through a message send thus respecting the object's encapsulation boundaries are called "encapsulated inheritance on objects" in [9]. As far as we know, Agora [2] is the only language featuring such an encapsulated inheritance mechanism. The only way to modify an object there is through invoking a so called "mixin method" which, upon invocation, extends the receiver with attributes enumerated in its body [10].

The example above illustrates that the calculus provides a formal basis for languages with encapsulated inheritance on objects. It allows more flexibility than mixin methods since generators can be explicitly manipulated whereas with mixin methods generators can only be manipulated implicitly at the semantic level. Therefore, the calculus also constitutes a medium for exploring how to extend languages like Agora with new features. An example of such a new feature is the alternative (encapsulated) dynamic modification mechanism given below.

The example in this section might seem somewhat verbose, but this need not be the case in a real programming language built on top of our model. For example, Agora has a very simple Smalltalk-like syntax.

3.2 Conservative Object Modification

Although inheritance is only possible with generators, the "`>…<`" operator allows an existing object to be extended "from the outside" without breaching encapsulation. It casts an object into a generator that can be extended afterwards. Since this generator ignores its self argument, late binding of self does not apply. We call such modifications *conservative* since they embed the object as is, without changing its internal workings. The modifier cannot depend on the self sends performed in the object but only on the abstract functionality offered by the object's message sending interface and therefore cannot breach encapsulation. To illustrate that conservative modification does not breach object-based encapsulation we translate the bank account example of section 1 into the calculus:

```
[ makeAccount(password) =
    [ amount = 5000;
      verify(pwd) = password.equals(pwd);
      deposit(val) = Self # [ Self; amount=[Self].amount.add(val) ];
      withdraw(arg) = Self # [Self].verify(arg.pwd)
             .ifTrue([ Self; amount=[Self].amount.subtract(arg.val) ])
    ];
  account = Env # [Env].makeAccount("007");
  fraud = Env # [
          ForgedAccount = [ >[Env].account<; verify(pwd) = TRUE ];
           steal(amount) = Self#[Self].ForgedAccount.withdraw([val=amount,pwd="?"])]
].fraud.steal(5000)     // Unsuccessful attempt to steal money.
```

The above `account` cannot be modified at will. It can only be modified indirectly by calling its `withdraw` and `deposit` methods. When a malevolent client tries to override the `verify` method from the outside through conservative modification, this does not affect the `account`'s internal workings. Its `withdraw` method will still refer to the original `verify` method, so the `account` is not compromised.

The ability to perform conservative modifications is not (yet) included in Agora, but based on the similarities with the calculus we are convinced that this new feature would be a valuable extension to the language.

4 Related Work

The delegation-based object calculus developed by Fisher and Mitchell [8] contains many similarities with ours. Whereas their calculus adds new syntactic forms to untyped lambda calculus, we prefer a pure object model, containing only objects as first-class entities, not functions. By adding a type system to their basic calculus, Fisher and Mitchell explicitly distinguish objects from inheritable entities called *prototypes*. This approach differs from ours from a pragmatic as well as a theoretical point of view:

- The theoretical difference is that we embed the distinction between objects and inheritable entities in the basic syntax and semantics of the calculus, so we do not need an additional type system for this purpose.
- A pragmatic difference lies in the motivation of the calculus: Fisher and Mitchell developed a new type system to cope with problems in existing type systems (such as [1]), while we have developed a two-layered calculus to solve the encapsulation problems in current prototype-based languages.
- Another difference between both approaches lies in the functionality of objects and inheritable entities. In [8] the functionality of objects forms a subset of the possible operations on prototypes: while prototypes can be used to send messages and to add or redefine methods, an object can only be sent messages to. In contrast, we keep the functionality of both kinds of entities strictly orthogonal. An object can only be sent messages to, while a generator can only be used to add or redefine methods.
- A more essential difference lies in the expressivity of objects. In [8] methods in an object are only allowed to redefine themselves or other methods due to typing problems. They are not allowed to add new methods. Nevertheless, the latter kind of behaviour is sometimes desired. It corresponds to the Agora-concept of mixin-method based inheritance.

In [1] a simple object calculus supporting method override and object subsumption is introduced. As in our approach, instead of struggling with complex encodings of objects as λ-terms or other primitive constructs, objects are taken as primitives themselves. The object calculus however does not have encapsulated modification of objects. An example similar to our "bank account" example of section 1 can be constructed to show that local variables can be exported using a clever overriding of methods. Moreover, as in [8] the mechanisms for incremental modification in this calculus are too restrictive. Only overriding of already defined methods is allowed. It is impossible to add new methods to an existing object.

In [9], a denotational semantics for a subset of AGORA (called MiniMix) is presented. Although this semantics was a major source of inspiration, MiniMix and AGORA only offer a limited form of encapsulated modification of objects. They only offer part of the functionality —inheritance with mixin methods— but offer no mechanism for conservative modifications. Furthermore the ability to explicitly name and manipulate the self of an object as a generator makes our inheritance mechanism more flexible than mixin methods.

5 Conclusions

We have presented a calculus modelling the kernel of a safe prototype-based language. This calculus does not suffer from the conflict between encapsulation and inheritance. This is accomplished by distinguishing objects from generators. Objects only provide message sending interfaces and generators take care of late binding.

The denotational semantics clearly shows that an object's message sending interface serves as an unbreachable abstraction barrier behind which implementation details can be hidden. The representation of objects as records of methods only exhibits how they react to messages. Properties of an object not accessible through its message sending interface are not manifest in the object's semantic representation. This guarantees that encapsulation boundaries cannot be breached.

Despite of the restrictions ensuing from the sober object model it is still possible to model several types of dynamic object modification. *Inheritance* with late binding is only possible by inheritors "from inside" an object. *Conservative modifications* without late binding can be performed to extend an object from the outside.

6 References

[1] Abadi, M. & Cardelli, L. - 1994. A Theory of Primitive Objects: Untyped and First-Order Systems. TACS '94 Proceedings, Springer-Verlag.

[2] Codenie, W; De Hondt, K; D'Hondt, T. & Steyaert, P. - 1994. Agora: Message Passing as a Foundation for Exploring OO Language Concepts. ACM SIGPLAN Notices vol 29 (12); pp. 48-49; ACM Press.

[3] Cook W. - 1989. A Denotational Semantics of Inheritance, Ph.D.-Thesis, Brown University.

[4] Dony, C.; Malenfant, J. & Cointe, P. - 1992. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. OOPSLA '92 Proceedings, pp. 201-217, ACM Press.

[5] Mezini, M. - 1995. Supporting evolving objects without giving up classes. TOOLS '95 Proceedings, pp. 183-197, Prentice Hall.

[6] Schmidt, D. A. - 1986. Denotational Semantics: A Methodology for Language Development; Allyn and Bacon, Inc.

[7] Snyder, A. - 1987. Inheritance and the Development of Encapsulated Software Components. Research Directions in Object-Oriented Programming; (eds.) Shriver, B. & Wegner, P.; pp. 165-188; MIT Press.

[8] Fisher, K. & Mitchell J. - 1995. A Delegation-based Object Calculus with Subtyping. FCT '95, LNCS 965, pp. 42-61, Springer-Verlag.

[9] Steyaert, P. & De Meuter, W. - 1995. A Marriage of Class and Object Based Inheritance Without Unwanted Children. ECOOP '95 Proceedings, LNCS 952, pp. 127-144, Springer-Verlag.

[10] Steyaert, P.; Codenie, W.; D'Hondt, T.; De Hondt, K.; Lucas, C. & Van Limberghen, M. - 1993. Nested Mixin-Methods in Agora. ECOOP '93 Proceedings, LNCS 707; pp. 197-219; Springer-Verlag.