A Layered Calculus for Encapsulated Object Modification

Extended Abstract

Kim Mens, Kris De Volder, Tom Mens, Patrick Steyaert

Department of Computer Science, Faculty of Sciences

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

E-mail: { kimmens@is1 | kdvolder@vnet3 | tommens@is1 | prsteyae@vnet3 }.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 does not have 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 the object-oriented programming paradigm 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 can be hidden from the user. However — as recognised by [7] — inheritance depends at least to some extent on these implementation details. Hence there is an inherent conflict between inheritance and encapsulation.

[7] introduced a notion of "encapsulated inheritance": inheriting clients have no direct access to the private attributes of their parents. Although this solves the conflict in class-based languages, [9] showed that in current prototype-based languages where inheritance is defined directly on objects this solution still does not prevent all violations of an object's encapsulation barrier. The same observation was made by [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) { if (val>0) amount=amount+val; }
      withdraw(val,pwd) { if (this.verify(pwd)) amount=amount-val; }
};
```

The protected `verify` method can be used by inheritors to create specialised versions of password verification. Although it is important to hide this information from message sending clients, this cannot be enforced in current prototype-based languages. Fraudulent "message sending" clients can gain access to implementation details by temporarily becoming 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
```

In class-based languages inheritance cannot be abused to breach an object's encapsulation because it is not defined on objects directly but rather on distinct inheritable entities called classes. Classes can be instantiated to form objects that can only be sent messages and cannot be further specialised. However class-based languages are sometimes considered too rigid e.g. when dynamic evolution of an object's behaviour is required [5] [8]. Prototype-based languages are more flexible, but fail to secure an object's encapsulation boundaries.

This paper proposes a calculus with dynamic object modification and a clean interaction between encapsulation and inheritance. This calculus can be used as a foundation for prototype-based languages without the inheritance-encapsulation conflict. 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 will 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 will be restricted to the inside of the object so that encapsulation cannot be breached. In spite of this restriction it will still be possible to model several mechanisms for (encapsulated) dynamic object modification.

2.1 Syntax

The top layer of the syntax dealing with objects and message sending is given by the following grammar. Terminal symbols are printed in bold. 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 dealing with generator expressions and inheritance is given by the following production rules:

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

Generators are inheritable entities from which objects can be created. The most primitive generators (apart from the empty generator ε) 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)=S#`*body* means that the name `S` can be used to denote self references inside the body of `m`. Inheritance can be accomplished through generator composition, because when generators are composed their late bound self will refer to the resulting composed generator.

Finally, the "`>…<`" operator allows to "convert" 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. An implementation that returns the encapsulated generator for example would reintroduce the encapsulation problems we are trying to avoid.

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

The following is a simple example[2] of an object expression in the calculus. It denotes an object representing a person named Joe, containing a method `isThatYou` that performs a self send of the `name` message:

```
[     name(dummy) = Self # "Joe";
      isThatYou(who) = Self # [Self].name([ε]).equal(who)          ]
```

In what follows, we assume some conventions to make examples more concise and readable. To avoid confusion between identifiers denoting arguments and identifiers denoting self references we start argument names with lowercase letters and self names with upper case letters. When no reference to the self generator is made inside the body of a method, we agree to omit the `#` binding operator. A message send which supplies no actual argument is considered equivalent to a message with the empty object `[ε]` as argument. Also, whenever a formal argument is not referred to in the body of a method we will omit it from the method definition. Using these conventions the example can be written more concisely as follows.

```
[     name = "Joe";
      isThatYou(who) = Self # [Self].name.equal(who)               ]
```

2.2 Denotational Semantics

Due to space limitations, the operational semantics of the calculus and some interesting theoretical properties thereof, such as confluence and a translation of λ-calculus into our calculus are left out from this document.[3] Instead a denotational semantics will be used 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
```

Object-based encapsulation means that an object's message sending interface constitutes an abstraction barrier behind which implementation details can be hidden from the user. The above representation of objects guarantees this 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 as explained above.

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

---

[2] Although not explicitely present in the syntax, we will use predefined strings (understanding `add` and `equal` messages) and numerals (understanding `equal`, `add`, `substract` and `greater` messages) to make more meaningful examples.

[3] A technical report containing these theoretical results is available by anonymous FTP at
progftp.vub.ac.be/FTP/tech_report/1996/vub-prog-tr-96-07.ps.Z

Scoping of generator names and argument names

Generator names and argument names will be lexically scoped. Therefore both the semantics of object expressions and generator expressions pass around two records which contain the bindings for argument names and generator names in their lexical environment. In the semantic equations the names `e` and `c` will be used for the records denoting the environment of argument objects and self generators respectively.

Also in the semantics below, `{}` denotes the empty record, `{key→val}` a single slot record, $e_1 +_r e_2$ right preferential record concatenation (i.e. rightmost occurences of identifiers take precedence), and `lookup r I` is used to denote the selection of identifier `I` in record `r` (which is undefined ⊥ when `I` does not occur in `r`).

Semantics of an Object Expression

The semantics 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

$[\![O_r.I(O_a)]\!]_O\ e\ c = (\text{lookup } ([\![O_r]\!]_O\ e\ c)\ I)\ ([\![O_a]\!]_O\ e\ c)$

$[\![I]\!]_O\ e\ c = \text{lookup } e\ I$

$[\![\texttt{[}G\texttt{]}]\!]_O\ e\ c = \text{wrap } ([\![G]\!]_G\ e\ c)$

Semantics of a Generator Expression

The semantics 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.

$[\![G_1;G_2]\!]_G\ e\ c = \lambda self.([\![G_1]\!]_G\ e\ c)\ self +_r ([\![G_2]\!]_G\ e\ c)\ self$

$[\![I]\!]_G\ e\ c = \text{lookup } c\ I$

$[\![\varepsilon]\!]_G\ e\ c = \lambda self.\{\}$

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.

$[\![I_m(I_a)=I_s\#O_{body}]\!]_G\ e\ c = \lambda self.\{I_m \rightarrow method\}$

where method = $\lambda arg.[\![O_{body}]\!]_O\ (e +_r \{I_a \rightarrow arg\})\ (c +_r \{I_s \rightarrow self\})$

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.

$[\![\texttt{>}O\texttt{<}]\!]_G\ e\ c = \lambda self.[\![O]\!]_O\ e\ c$

Summarising

It is clear from the semantics 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, `Objects`s and `Method`s 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

We have proposed a calculus that respects object-based encapsulation. In this section we will illustrate that it 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 performs a self send to the `sex` attribute. In this way it anticipates the overriding of the `sex` attribute. 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. In Agora the only way to modify an object is through invoking a so called "mixin method". A mixin method [10] is a method that, upon invocation, extends the receiver with methods enumerated in the body of the mixin method.

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

3.2 Conservative Object Modification

Although inheritance is only possible with generators, the "`>…<`" operator provides a way to extend an existing object "from the outside" without breaching encapsulation. This operator 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 is non-intrusive with respect to object-based encapsulation consider a translation of the bank account example into the calculus:

```
[ makeAccount(password) =
    [ amount = 5000;
      verify(pwd) = Self # password.equals(pwd);
      deposit(val) = Self # val.greater(0)
             .ifTrue([ Self; amount=[Self].amount.add(val) ]);
      withdraw(arg) = Self # [Self].verify(arg.pwd)
             .ifTrue([ Self; amount=[Self].amount.substract(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. A malevolent client can nevertheless try to override the `verify` method from the outside through conservative modification. However, this does not affect the `account` object's internal workings. Its `withdraw` method will still refer to the original `verify` method, so the `account` is not compromised.

From a software engineering point the manifestation of these different kinds of dynamic object modification — encapsulated inheritance and conservative modification — seems very natural. Code for an implementation dependent modification of an object has to be nested somewhere inside the object, thus clearly identifying it as belonging to that object and depending on its implementation details. In contrast, a conservative modification can be applied to several objects sharing the same message sending interface and can be encoded separately from the object(s).

While Agora provides mixin methods as a language mechanism for encapsulated inheritance, the ability to perform conservative modifications is not (yet) included. However, based on the similarities between the calculus and Agora, we are convinced that this new feature would be a valuable extension to the language.

4 Conclusions

We have presented a calculus modelling the kernel of a 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 also be performed to extend an object from the outside.

5 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] Stein, L. A.; Lieberman, H. & Ungar, D. - 1989. A Shared View of Sharing: The Treaty of Orlando. Object-Oriented Concepts, Databases, and Applications; (eds.) Kim, W. & Lochovsky, F. H.; pp.31-48; ACM Press.

[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.