OPUS: a Calculus for Modelling Object-Oriented Concepts

Tom Mens, Kim Mens, Patrick Steyaert

Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: { tommens@is1 | we34154@is1 | prsteyae@vnet3 }.vub.ac.be
Fax: +32 2 629 3495

ABSTRACT. *We propose a very concise calculus for modelling object-oriented concepts such as objects, message sending, encapsulation and incremental modification. We show how to deal with recursion and provide some object-oriented examples. State and encapsulated inheritance are modelled by means of an incremental modification operator.*

KEY WORDS: *calculus, encapsulated objects, message sending, incremental modification, encapsulated inheritance.*

1. Introduction

This paper proposes a foundation for modelling the essential concepts of object-orientedness (OO) by means of an elementary calculus called OPUS, which is an acronym for "Object-oriented Programming calculUS". Our calculus is based on the model of *substitutable objects*. This kind of object model was introduced in (Steyaert, 1994) to denote objects with explicit interfaces, a notion of encapsulation and polymorphism, and atomic message sending.

First of all, it is obvious that we need a notion of objects. Moreover, our object-oriented model is homogeneous since the only data structures are objects, while the only control structure is message sending. This makes our model very simple. For reasons of simplicity we have also chosen not to include state into our model. It is sufficient to provide only methods (behaviour), because state can be modelled by constant methods (i.e. methods that always return the same result). We will show that updating state can be modelled by an *incremental modification mechanism*. This mechanism is similar to the one described in (Wegner et al., 1988), and can also be used to model a form of *encapsulated inheritance*, as proposed by (Snyder, 1987) for reasons of reusability of software components.

To this extent we also need to provide an *encapsulation mechanism* that allows us to create objects with a public part and an encapsulated private part. This encapsulation operator will play an essential role in modelling recursive objects in our calculus *without needing explicit provisions for recursion*. Recursion can be dealt with by two alternative operators: a first one similar to the Y-operator of lambda-calculus, giving rise to infinite recursion; and a second one unfolding only one level of recursion, needed for updating state.

OPUS *explicitly uses names* in the method lookup mechanism. There is ample evidence that names simplify the modelling of object-oriented systems. In order to make the examples more concise and understandable, we will allow arguments to be passed with messages.

In the following section we will motivate from different perspectives why a formal model for OO can be useful. Section 3 describes the syntax and derivation mechanism of OPUS, gives an informal explanation, and shows how to deal with recursion and updating state using only this basic syntax. In the fourth section we present some object-oriented examples. More specifically we show how Booleans, numerals and class-based inheritance can be expressed. The subsequent section discusses some related work. In the last section we draw some conclusions, and discuss a few topics that might be worthwhile investigating in the future.

2. Motivation

In order to gain a consistent understanding of OO and its descriptions one first needs to find the key features for object-oriented programming languages (OOPLs). For obvious reasons we want these features to be as orthogonal as possible. Once the key concepts are found, a formalisation of OO can be developed in the form of a calculus with only a very restricted set of syntactic constructs. This calculus can then be used as a theoretical foundation of object-oriented languages, just as is the case for lambda-calculus in the functional programming paradigm. Such a formal framework is useful for many reasons.

First of all, many people lack a firm and rigorous ground when trying to prove general properties for OOPLs, partly because most concepts of OO have no generally accepted definition, and because there is still a lot of controversy about which features - classes, inheritance, delegation, encapsulation, message sending - are at the heart of OO, and how they relate to one another. By modelling these features in our calculus we will be able to get more insight in this matter.

An OO-calculus is also interesting from a more pragmatic standpoint. In analogy with the design of functional languages based on lambda-calculus, a concise OO-calculus might lead to development of new OOPLs with a more simple and orthogonal design but with the same expressivity as currently available OOPLs.

Most of the current research on formal models of OO consists of attempts to generalise lambda-calculus. A lot of these models suffer from difficulties in expressing the essential features of object-orientedness in a satisfying way. For this reason we feel that when developing a calculus one should stay closer to real OO languages, and only at a later stage one could investigate if the proposed model subsumes lambda-calculus. This is more or less the same methodology as followed by Laurent Dami, who first developed the HOP-calculus and later proved that this model can be regarded as an extension of de Bruijn lambda-calculus (de Bruijn, 1972) with names, combinations and alternations (Dami, 1994).

3. The calculus

3.1. Informal discussion of the syntax

In OPUS, an object consists of a *public part* and an encapsulated *private part* that is only accessible through the object's operations. Such an object is denoted as [*Public* | *Private*]. The public and private part of an object can contain a list of methods. The public part contains the methods visible for other objects, while the private part contains methods that are only used to implement public methods; hence they are not externally visible.

Methods will be invoked using the following message sending syntax: sending a message N with arguments A to an expression E is denoted by (E N:A). Apart from ordinary methods we need a special kind of methods to model state, namely constant methods. The difference with ordinary methods lies in the way they are executed. Furthermore, with constant methods the arguments in A will always be ignored.

An ordinary method will be represented by the symbol λ followed by a name, an equality sign and an expression (e.g. λgetx=x). Method execution is a combination of method selection and method application, and occurs as follows: if a message corresponding to a method is sent to an object, this method is looked up (method selection), unbound variables in the body of the method are bound to the arguments of the message (argument passing) or to values of corresponding attributes in the private part (static scoping), the method is evaluated (method invocation), and the result is returned. This result is typically again an object.

A constant method is represented by a name, followed by an equality sign, followed by an expression (e.g. x=xval). Constant methods have no arguments, and do not call attributes in the private part. If a message corresponding to a constant method is sent to an object, the body of this method is simply returned.

As a basic structure to create more complicated objects we use records, which is a common approach followed in literature (Cook, 1989), (Cardelli et al., 1989) and (Wegner et al., 1988). Objects in their most simple form have an empty private part and a public part which is a record of methods, for example [λgetx=x λgety=y |]. A more general form of objects are those containing a private part which is a record as well, e.g. [λgetx=x λgety=y | x=1 y=2]. In the most general case, the public and private parts of an object need not be records, but can be objects themselves. This is an important advantage of our calculus over lambda-calculus with records (Cardelli, 1988) and (Cardelli et al., 1989). It is essential on the one hand to model private methods, and on the other hand to have some form of binding of constant methods in different stages.

In many cases it is useful to have some kind of incremental modification mechanism. The incremental modification of an expression P (parent) with an expression M (modifier) is denoted by (P+M). This operator will prove useful when dealing with inheritance, because inheritance can be characterised as incremental modification with deferred binding of self-reference (Wegner et al., 1988). A further investigation of inheritance is delayed until section 4.3.

There is however a problem when defining an inheritance operator, because "inheritance may be considered a fundamental violation of encapsulation because of the change of self-reference" (Cook, 1989), or stated otherwise "the introduction of inheritance severely compromises encapsulation, by exposing implementation details to inheriting clients" (Snyder, 1987). An extensive discussion on the interaction between inheritance and encapsulation can be found in (Snyder, 1987). In our paper the incremental modification operator + is defined in such a way that it gives rise to a notion of *encapsulated inheritance* (each object can only refer to its own private methods and not to those of its parents, and vice versa). It is also possible to give an alternative definition of this operator so that it yields a form of *non-encapsulated inheritance*, but that is beyond the scope of this paper.

The operational semantics[1] of method lookup will be defined in a similar way as in object-oriented languages like Smalltalk. Sending a message to a modified object (E_1+E_2) is implemented by first sending the message to the rightmost object E_2, and if no corresponding attribute is found we continue searching in the object E_1 on the left. Hence the incremental modification operator is not commutative: if we compose two objects that both contain attributes corresponding to the same name, the attributes of the rightmost object will "overwrite" those of the leftmost object.

The following example illustrates the use of the incremental modification operator as well as the method lookup mechanism. Suppose we have a POINT-object with private constant methods x=1 and y=2, that defines the methods getx and gety as follows[2]:

```
POINT := [ λgetx=x λgety=y | x=1 y=2 ]
```

We can modify this object to obtain a three-dimensional point object that inherits all methods of POINT and supplementary adds a public getz method, needed to access the private constant method z=3, by extending the POINT-object with the following modifier:

```
MODIFIER := [ λgetz=z | z=3 ]
3DPOINT := ( POINT + MODIFIER )
```

[1] A denotational semantics for the method lookup mechanism can be found in (Cook, 1989) or (Cook et al., 1989).

[2] The operator := that binds expressions to variables, is not present in the syntax of the calculus. It is only used to make the examples more readable. Words written in uppercase refer to "predefined" objects that have to be replaced "in place".

Sending the parameterless message[3] gety to this 3DPOINT will result in first looking in the MODIFIER-object for an attribute corresponding to gety, but because no attribute will be found, we continue searching for this attribute in the POINT-object. The method gety found in POINT is then executed by evaluating its body in the context of the private part.

```
3DPOINT gety:[|]
=     ( POINT + MODIFIER ) gety:[|]
⇒     POINT gety:[|]
⇒     [ x=1 y=2 | ] y:[|]
⇒     2
```

Note that methods in the MODIFIER-object can never directly access the private attributes x and y of POINT, because the incremental modification operator + is defined in such a way that it gives rise to a notion of *encapsulated inheritance*. Similarly, the public methods of POINT can only access their own private attributes, and not those of MODIFIER.

3.2. Context free grammar

Formally, expressions in our calculus are defined by providing a context free grammar in EBNF-notation. The start symbol of this grammar is Expression, and all terminal symbols are written in bold between double quotes.

```
Expression      ::=   Object | Message | Modification | Name
Object          ::=   "[" ExtExpression "|" ExtExpression "]"
ExtExpression   ::=   Expression | Record
Record          ::=   { Attribute }
Message         ::=   "(" Expression Name ":" Expression ")"
Modification    ::=   "(" Expression "+" Expression ")"
Attribute       ::=   Constant | Method
Constant        ::=   Name "=" Expression
Method          ::=   "λ" Name "=" Expression
Name            ::=   Character { Character }
Character       ::=   "a" | "b" | "c" | ... | "z"
```

Notice that this syntax allows a Record to be empty. A Record can also contain different attributes corresponding to the same name, but only the last occurrence of this name will be significant, as will become clear in the reduction rules.

Message and Modification both require parentheses. To make the notation somewhat lighter, we will adopt the convention to omit parentheses whenever confusion is impossible. Furthermore, in case of ambiguity an expression will be parsed from left to right. E.g. A+B+C denotes (A+B)+C instead of A+(B+C), and similarly for message sending.

3.3. Reduction rules

For the rest of this paper, let N denote a meta-variable of type Name, E and E_i meta-variables of type Expression, F and F_i meta-variables of type ExtExpression, R and R_i meta-variables of type Record, and A a meta-variable of type Attribute.

Definition: An attribute A is called an **N-attribute** if A is of the form **N**=E or λ**N**=E.

[3] We will see that the syntax of OPUS requires us always to send messages with arguments. Sending a message without arguments can be done by sending a message with the empty object [|] as argument.

Using this definition, the reduction rules can be expressed as follows:

<u>Rule 1</u>: Message sending to an object

$$\begin{array}{llll}
a) & [\ R \mid F\ ]\ N{:}E & \rightarrow & (\ [|] + [\ R \mid F\ ]\ )\ N{:}E & \text{if } R \text{ is not empty} \\
 & & & [\ |\ ] & \text{if } R \text{ is empty} \\
b) & [\ (\ E_1 + E_2\ ) \mid F\ ]\ N{:}E & \rightarrow & (\ [\ E_1 \mid F\ ] + [\ E_2 \mid F\ ]\ )\ N{:}E &
\end{array}$$

<u>Rule 2</u>: Message sending to an incrementally modified object

a) Constant method selection

$$(\ E_1 + [\ R\ \ N{=}E_2 \mid F\ ]\ )\ N{:}E \ \rightarrow\ E_2$$

b) Method execution

$$\begin{array}{llll}
(\ E_1 + [\ R\ \ \lambda N{=}E_2 \mid F\ ]\ )\ N{:}E & \rightarrow & \{\ (\ F + E\ )\ \}(E_2) & \text{if } F \text{ is no Record} \\
 & & \{\ (\ [\ F \mid\ ] + E\ )\ \}(E_2) & \text{if } F \text{ is a Record}
\end{array}$$

c) Attribute lookup

$$\begin{array}{llll}
(\ E_1 + [\ R\ \ A \mid F\ ]\ )\ N{:}E & \rightarrow & (\ E_1 + [\ R \mid F\ ]\ )\ N{:}E & \text{if } A \text{ is no } N\text{-attribute} \\
(\ E_1 + [\ \mid F\ ]\ )\ N{:}E & \rightarrow & E_1\ N{:}E &
\end{array}$$

<u>Rule 3</u>: Currying

$$\begin{array}{llll}
[\ [\ F_1 \mid F_2\ ] \mid F_3\ ] & \rightarrow & [\ F_1 \mid [\ F_2 \mid F_3\ ]\ ] & \text{if } F_2 \text{ is not empty} \\
 & & [\ F_1 \mid F_3\ ] & \text{if } F_2 \text{ is empty}
\end{array}$$

The notation → used in the reduction rules means "...reduces in one step to...", while ⇒ ("...reduces in zero or more steps to...") will be used in the subsequent examples to denote the transitive closure of →. An expression is in *normal form* if it cannot be reduced any further.

In rule 1, message sending to an object occurs by sending the message to the empty object [|] incrementally modified with the given object. If the message is not found in the object, the expression will automatically reduce to the empty object. For example:

$$\begin{array}{lll}
\multicolumn{2}{l}{[\ \lambda getx{=}x \mid x{=}1\ ]\ sety{:}[\ y{=}1 \mid\ ]} & \\
\rightarrow & (\ [|] + [\ \lambda getx{=}x \mid x{=}1\ ]\ )\ sety{:}[\ y{=}1 \mid\ ] & \text{(rule 1a)} \\
\rightarrow & (\ [|] + [\ \mid x{=}1\ ]\ )\ sety{:}[\ y{=}1 \mid\ ] & \text{(rule 2c)} \\
\rightarrow & [|]\ sety{:}[\ y{=}1 \mid\ ] & \text{(rule 2c)} \\
\rightarrow & [\ |\ ] & \text{(rule 1a)}
\end{array}$$

Rule 2b states that, to execute a method, we first bind all unbound variables in the body E_2 of the method N to the parameters E provided with the message, and next all remaining unbound variables are bound in the context of the private part F of the object of which the method was part of. This can be done by evaluating the body of the method in the context of the object (F+E). In other words, we implicitly use our incremental modification operator to model argument passing! To formalise the notion of evaluation in a context, a new notation {} will be introduced further. This notation does not belong to the syntax but can be seen as a kind of meta-level reduction scheme.

A problem still present in our approach is that the argument passing mechanism as proposed in this paper jeopardises encapsulation. The reason for this is that in rule 2b arguments have precedence over private methods. For example

$$[\ \lambda getx{=}x\ \ \lambda gety{=}y \mid x{=}1\ \ y{=}2\ ]\ getx{:}[\ x{=}2 \mid\ ]$$

yields 2 instead of the expected result 1. This problem can be solved by adding the restriction that argument names and private method names should be disjoint. Because we think that such issues will make the syntax needlessly difficult, we believe this problem should be solved by providing a higher level syntax.

Rule 2 formalises the method lookup mechanism that was explained earlier in the informal discussion of the syntax: to look up an attribute in a composed object, we search the record in the public part of the *rightmost* object *from right to left* for an attribute corresponding to the message name, and then execute the corresponding method. If no attribute is found in this object, we restart the whole process on the leftmost object. This process is illustrated in the following reduction:

```
([ λgetx=x  λgety=y | x=1 y=2 ] + [ λgetz=z | z=3 ]) getx:[|]
→    ([ λgetx=x  λgety=y | x=1 y=2 ] + [ | z=3 ]) getx:[|]            (rule  2c)
→    [ λgetx=x λgety=y | x=1 y=2 ] getx:[|]                           (rule  2c)
→    ([|] + [ λgetx=x  λgety=y | x=1 y=2 ]) getx:[|]                  (rule  1a)
→    ([|] + [ λgetx=x | x=1 y=2 ]) getx:[|]                           (rule  2c)
→    {([ x=1 y=2 | ] + [|])}( x )                                     (rule  2b)
⇒    1
```

The modification operator + is in fact an (implicit) delegation operator. The term *delegation* refers to the fact that an object can delegate responsibility of a message it cannot handle to objects that potentially could. Delegation is *implicit* because all messages that are not understood by the receiving object are automatically (= implicitly) delegated to its parent. Indeed, if a message N with arguments E is sent to an object E_2 with parent E_1 (i.e. (E_1+E_2) N:E) and E_2 does not contain any N-attribute, then rule 2c automatically delegates the message to the parent E_1 (i.e. E_1 N:E).

Rule 3 is needed because it is possible for a public part to be again an object, while the message sending mechanism only works with objects where the public part is a record.

3.4. Evaluation in a context

In rule 2b we used the meta-level reduction scheme $\{E\}(E_2)$ for the *evaluation of an expression E_2 in a context E*. The context E in which an expression is evaluated can be an arbitrary expression. The definition of evaluation in a context can be given inductively as follows[4]:

Inductive case:

$\{E\}(\ (E_1\ N{:}E_2)\ )$	equals	$(\ \{E\}(E_1)\ \ N{:}\{E\}(E_2)\ )$
$\{E\}(\ [\ F_1 \mid F_2\ ]\ )$	equals	$[\ \{E\}(F_1) \mid \{E\}(F_2)\ ]$
$\{E\}(\ (E_1 + E_2)\ )$	equals	$(\ \{E\}(E_1) + \{E\}(E_2)\ )$
$\{E\}(\ L\ \ A\ )$	equals	$\{E\}(L)\ \ \{E\}(A)$
$\{E\}(\ N=E_1\ )$	equals	$N=\{E\}(E_1)$

The inductive case captures the intuition that evaluation of an expression in a context corresponds to evaluation of the components of the expression in that context.

Base case:

$\{E\}(\ N\ )$	equals	$E\ \ N{:}[\mid]$
$\{E\}(\ \lambda N=E_1\ )$	equals	$\lambda N=E_1$
$\{E\}(\ \varepsilon\ )$	equals	ε

Intuitively the evaluation of an expression in a context corresponds to replacing all unbound names in that expression by their meaning in that context. Therefore evaluation of a name in a context simply corresponds to looking up that name, i.e. by sending the name as a message to the context. Evaluation of a method $\lambda N=E_1$ in a context leaves the method body unchanged. One might say that all free names in E_1 are bound by the λ.

3.5. Dealing with recursion

In most object-oriented languages it is very common that an object can call its own (public) methods using a recursive reference. Although we did not explicitly introduce recursion in our calculus we

[4] In this definition, ε is used to denote the empty expression.

will show how it can be implemented in a straightforward way. Our approach will be similar to the one followed by (Dami, 1994) for introducing recursion into λN-calculus.

Due to the special treatment of names in OPUS, we need a fixpoint operator μ_S for each name s, satisfying the property that taking the fixpoint over an expression yields a new expression that is exactly the same as the original one, except that every occurrence of s should be replaced by this new expression. In other words, the expression can refer to itself using the name s. Informally, the operator μ_S should behave as follows:

μ_S fix:[par=E |] ⇒ [E | s = [E | s = [E | s = ...]]]

It is possible to define such an operator in the OPUS-syntax, using a rather difficult expression:

μ_S := [λfix=[par | s=([Z_S | self=Z_S e=par] res:[|])] |]
where Z_S = [λres=[e | s=([self | self=self e=e] res:[|])] |]

The fixpoint operator μ_S can for example be used to calculate the factorial of an integer in a functional way. Due to space limitations we will only give an informal description of what this factorial should look like:

FACBODY := [λres=*if* n=*0 then 1 else* n*(fac res:[n=n-*1* |]) |]
FAC := μ_{fac} fix:[par= FACBODY |]

In the approach followed above we observe that once the fixpoint μ_S fix:[par=E |] is defined for a given expression E, every reference to s will always return the same object [E | s=[E | s=[E | s=...]]]. This is not necessarily what we want in an OO approach. If the "internal state" of an object is changed, we want s to refer to this *updated* object, not to the original object. This problem can be solved by defining an operator that unfolds only one level of recursion. I.e. instead of using μ_S we will introduce an alternative operator σ_S with the property

σ_S unfold:[par=E |] ⇒ [E | s = E]

It is very easy to check that the expression below satisfies this property.

σ_S := [λunfold=[par | s=par] |]

To simplify the examples in the rest of this paper we will abbreviate expressions of the form
σ_S unfold:[par=E |] to σE.

The essential difference between μ_S and σ_S is that the former corresponds to an infinite recursive unfolding, whereas the latter expands only one level. The advantage of infinite recursion is that we only need to apply the fixpoint operator once, and that any future reference to s will yield the same object. However, this implies that the object cannot be updated. On the other hand, if we only unfold one level we have the disadvantage that the operator σ_S needs to be rewritten each time we want to unfold another level. But the advantage is that we can update the object at each level because every time we use the operator, s will refer to the latest version of the object instead of the original object.

3.6. Generator objects

When dealing with self-reference, things are quite similar to the generator functions of (Cook, 1989). A *generator* is an object that doesn't have a bound self-reference yet. To create objects that can refer to themselves, the self-reference needs to be bound. In our notation this is exactly done by the σ-notation.

For example, let G_1 be a generator object that makes use of self. Then σG_1 := [G_1 | self = G_1] corresponds to the same object where every self-reference is bound to the object G_1 itself.

G_1 := [a=2 λdouble=(σself a:[|]) * 2 |]
σG_1 double:[|] ⇒ 4

Generators may also be modified to define a new generator. This kind of use is unique to inheritance due to late binding of self-references: self calls of a parent should refer to the modified object instead of to the parent object. To insure that self-reference is handled properly, first the generator G_1 needs to be modified with a generator G_2 yielding (G_1+G_2), and only then the σ-operator should be applied. The example underneath clearly illustrates that this approach correctly handles late binding of self.

G_2 := [a = 3 |]
$\sigma(G_1+G_2)$ double:[|] ⇒ 6

When the σ-operator is applied before the generators G_1 and G_2 are composed, we do not obtain late binding of self (because self-reference in G_1 refers to the unmodified version of G_1):

$(\sigma G_1+\sigma G_2)$ double:[|] ⇒ 4

3.7. Updating state

Assume we want to define a POINT object with two private variables x and y that can only be accessed by means of the public methods getx and gety (that simply return the value of x and y respectively), and set (that stores a new value in x and y). The set method should be invoked by a message with two parameters x and y representing the new x- and y-values.

```
POINT := [ λgetx=x λgety=y λset=σ(self+[λgetx=x λgety=y | x=x y=y])
           | x=0 y=0 λself=self ]
```

Since state is modelled via incremental modification, we need the above described mechanism of generator combination: when we take a look at the set method, we see that to modify the POINT-object we have to modify self with the new values of x and y, and then apply the σ-operator. Also notice that because we work with encapsulated inheritance the modifier-object should contain in its public part all state-accessor methods, and in its private part the new values of the state.

The object σPOINT with bound self-references will satisfy our requirements, as we can see from the following reduction:

```
(σPOINT set:[x=1 y=2|]) gety:[|]
⇒    [ ( POINT + [ λgetx=x λgety=y | x=1 y=2 ] )
     | self = ( POINT + [ λgetx=x λgety=y | x=1 y=2 ] ) ]  gety:[|]
→    ( [ POINT | self=(POINT+[ λgetx=x λgety=y | x=1 y=2 ]) ]
     + [  [λgetx=x λgety=y | x=1 y=2]
          | self=(POINT+[λgetx=x λgety=y | x=1 y=2]) ]
     )  gety:[|]                                          (rule 1b)
→    ( [ POINT | self=(POINT+[ λgetx=x λgety=y | x=1 y=2 ]) ]
     + [ λgetx=x λgety=y
          | [ x=1 y=2 | self=(POINT+[ λgetx=x λgety=y | x=1 y=2 ]) ] ]
     )  gety:[|]                                          (rule 3)
→    { ( [ x=1 y=2 | self=(POINT+[ λgetx=x λgety=y | x=1 y=2 ]) ]
       + [|] ) } (y)                                      (rule 2b)
⇒    2
```

4. Examples

We will illustrate how some basic object-oriented examples can be expressed in OPUS in a relatively straightforward fashion. More specifically we will define Boolean objects, numerals and classes.

4.1. Booleans

First we will show how to express the Booleans TRUE and FALSE. The standard way to do this is by defining Booleans as objects that understand the basic logical operators if, ifnot, and, or and not. As a first attempt, we simply define TRUE and FALSE as records understanding exactly those messages:

```
TRUE := [ λif=then  λifnot=else  λand=second  λor=TRUE  λnot=FALSE | ]
FALSE := [ λif=else  λifnot=then  λand=FALSE  λor=second  λnot=TRUE | ]
```

The problem is that those objects are mutually recursive, because TRUE is defined in terms of itself and FALSE, and vice versa. To solve this, we need two additional instance variables: a self instance variable referring to the object itself, and an instance variable other referring to the object with which the given object is mutually recursive. The revised versions of TRUE and FALSE are defined below:

```
TRUE     := [ TRUE' | self=TRUE' other=FALSE' ]
FALSE    := [ FALSE' | self=FALSE' other=TRUE' ]
TRUE'    := [ λif=then λifnot=else λand=second λor=SELF λnot=OTHER | ]
FALSE'   := [ λif=else λifnot=then λand=SELF λor=second λnot=OTHER | ]
SELF     := [ self | self=self other=other ]
OTHER    := [ other | self=other other=self ]
```

Note that we cannot simply write TRUE := σTRUE' (or FALSE := σFALSE'), because we do not only need a recursive reference to TRUE itself, but also one to FALSE.
It is easy to see that these definitions yield the expected results, as can be verified by means of the following derivations:

```
TRUE if:[ then=1 else=2 | ]  ⇒  1
FALSE ifnot:[ then=1 else=2 | ]  ⇒  1
FALSE not:[|]  ⇒  TRUE
TRUE and:[ second=FALSE | ]  ⇒  FALSE
TRUE or:[ second=FALSE | ]  ⇒  TRUE
```

4.2. Numerals

In a similar way, numerals can be defined as objects understanding the basic arithmetic operations. For the sake of simplicity, we will only deal with positive integer numbers (although negative numbers can be expressed in a similar way). Once the unary messages iszero, pred and succ are defined, all other operations plus, minus, times and fac can be expressed in terms of those:

```
a plus b =     b                          if a is zero
               (a pred) plus (b succ)     otherwise
a minus b =    a                          if a or b is zero
               (a pred) minus (b pred)    otherwise
a times b =    a                          if a is zero
               ((a pred) times b) plus b  otherwise
a fac =        a succ                     if a is zero
               ((a pred) fac) times a     otherwise
```

Note that in the previous definitions we defined zero minus b as zero, because we only work with positive numbers. In all these definitions we had to distinguish the special case where a is zero. For this reason we will need two kinds of objects: ZERO and POSITIVE, where the latter is a kind of template for all positive numbers:

```
ZERO := σ[    iszero=TRUE
              λpred=σself
              λsucc=σ( POSITIVE + [ pred=σself | ] )
              λplus=arg
              λminus=σself
              λtimes=σself
              λfac=σself  succ:[|]
              | ]

POSITIVE := [   iszero=FALSE
                λsucc=σ( self + [ pred=σself | ] )
                λplus=σself  pred:[|]  plus:[ arg=arg  succ:[|]  | ]
                λminus=arg  iszero:[|]
                        if:[     then=σself
                                 else=σself  pred:[|]  min:[arg=arg  pred:[|]  |]
                                 | ]
                λtimes=σself  pred:[|]  times:[ arg=arg  | ]  plus:[ arg=arg  | ]
                λfac=σself  pred:[|]  fac:[|]  times:[ arg=σself | ]
                | ]
```

Using these definitions the numerals can be defined as follows:

0 := *normal form of* ZERO
1 := *normal form of* **0** succ:[|]
2 := *normal form of* **0** succ:[|] succ:[|] = *normal form of* **1** succ:[|]

One can check that these numerals are well defined (i.e. they express the arithmetic operators in a faithful way). A reduction of an expression with numerals is given below.

0 succ:[|] times:[arg=**2**|] minus:[arg=**1**|] ⇒ **1**

4.3. Class-based inheritance

In this section, we show by means of an example how to deal with class-based inheritance. Classes can be defined as templates from which objects are created. Different instances of a given class have the same behaviour, but can have a different internal state. For example we can define a point class where the private constant methods x and y differ between different instances of this class, while the getx, gety and set methods are the same for all instances. Moreover we add two new methods to illustrate late binding of self-reference. The move-method expects two arguments dx and dy, and updates the values x and y by invoking the set-method with x+dx and y+dy as arguments. The double-method doubles the coordinates x and y simply by invoking the move-method with the old values of x and y as arguments.

To simplify the subsequent examples, we will extend the meaning of σ_S:

σ_S := [λunfold=[par | s=par class=class super=super] |]

σ_{self} unfold:[par=E |] ⇒ [E | self=E class=class super=super]

As before we will abbreviate σ_{self} unfold:[par=E |] to σE.

Formally, a point instance without bound self- and class-references looks as follows:

```
POINT := [ λgetx=x
           λgety=y
           λset=σ( self + [ λgetx=x  λgety=y | x=x  y=y ] )
           λmove=σself set:[  x=σself  getx:[|]  plus:[arg=dx|]
                              y=σself  gety:[|]  plus:[arg=dy|]  |  ]
           λdouble=σself move:[  dx=σself  getx:[|]
                                 dy=σself  gety:[|]  |  ]
           | x=0  y=0  λself=self  λclass=class  ]
```

Note that in the private part two methods occur. The self-method is used to be able to deal with recursive self-references, while the class-method is needed to refer to the point class itself. The bodies of both methods will be filled in via curried binding. Point instances can be created by invoking the new-method of the point class

```
POINTCLASS := [ λnew=[ inst | self=inst class=self ] | ]
```

where self and inst are bound to POINTCLASS and POINT respectively:

```
PC := [ POINTCLASS | self=POINTCLASS inst=POINT ]
PC new:[|]  ⇒  [ POINT | self=POINT class=POINTCLASS ]
```

Subclasses of the point class can be created by incrementally modifying them for example with a MODIFIER implementing a move-method that only moves the x-coordinate while keeping the y-value unaltered:

```
MODIFIER := [  λmove=σself set:[  x=σself  getx:[|]  plus:[arg=dx|]
                                  y=σself  gety:[|]  |  ]
               | λself=self  λclass=class  ]
```

Instances of such a modified point class MODCLASS can be obtained by invoking its new-method. To illustrate the use of super, we will implement this new-method as a super call to the new-method of the point class.

```
MODCLASS := [ λnew=[ super | self=self inst=inst ] new:[|] | ]
```

Again self, inst and super need to be bound in order to become a meaningful object.

```
MC := [ MODCLASS | self=MODCLASS inst=(POINT+MODIFIER) super=POINTCLASS ]
```

In the above definition of inst we clearly see the incremental modification: POINT is incrementally modified with MODIFIER by using the + operator. Also note that the super variable in the above definition can only be used by class methods. In order to allow instance methods to perform super calls, another super variable needs to be introduced. We have chosen not to do this to keep the example simple.

Finally one can observe the correctness of all these definitions by looking at the reductions below. The first two create an instance of the point class, set the coordinates of the instantiated point, double these coordinates and return the x- and y-values respectively:

```
PC  new:[|]  set:[x=1  y=2|]  double:[|]  getx:[|]        ⇒       2
PC  new:[|]  set:[x=1  y=2|]  double:[|]  gety:[|]        ⇒       4
```

To illustrate the late binding of self, observe that invoking the double-method on an instance of the modified class exhibits a different behaviour than invoking the same method on an instance of the point class, since the double-method makes use of the move-method, which was overridden in the modifier instance!

```
MC  new:[|]  set:[x=1  y=2|]  double:[|]  getx:[|]        ⇒       2
MC  new:[|]  set:[x=1  y=2|]  double:[|]  gety:[|]        ⇒       2
```

There are still some difficulties with modelling classes in our object-based model. The dilemma is that normal message sending requires objects with an encapsulated self (e.g. PC and MC), while inheritance on objects requires objects with a non-encapsulated self (e.g. POINT+MODIFIER) in order to be able to ensure late binding of self.

5. Related research

In most formal approaches to OO, objects are modelled as records (Cardelli, 1988), (Cardelli et al., 1989), (Cook, 1989), (Dami, 1994), and (Wegner et al., 1988). In OPUS objects are more general since they are composed of a public and a private part, which are both records, or can even be objects themselves. This is an important advantage of our calculus over the more primitive approaches.

Most of the currently available object-oriented models (Castagna et al., 1992), (Dami, 1994), (Hofmann et al., 1993), (Pierce et al., 1994) are based on some extension or variant of lambda-calculus. OPUS on the other hand is not based on lambda-calculus, because we feel that it is necessary to stay closer to the object-oriented paradigm. Indeed, many of the proposed models such as (Hofmann et al., 1993), (Pierce et al., 1994), and (Dami, 1994) seem to have some problems because they are too functional. Apart from objects, functions are first class entities as well, and apart from message sending, function application is also a control structure. Therefore these models are not homogeneous.

According to (Steyaert, 1994), atomic message sending means that the syntax of message sending is always the same and that it is not composed out of more elementary building stones. Most models based on lambda-calculus with records do not have atomic message sending, since method selection and method application are considered distinct operations[5]. In OPUS on the contrary message sending is an atomic operation, since reduction rule 2 shows that method selection and method application are performed simultaneously.

For reasons advocated by (Canning et al., 1989) and (Steyaert, 1994), OPUS objects have explicit interfaces, i.e. an object's interface is determined totally by the object's definition: the object should always respond to the same messages, independent of the context in which it is used. Explicit interfaces are important in realising data abstraction, data encapsulation and separate compilation. They have many uses which bear on quality and productivity in object-oriented software development. These include compatibility checking, system design and documentation, and software reuse.

One of the calculi that is closest (in thought, if not in form) to ours is Dami's λN-calculus (Dami, 1994). Many examples expressible in λN are expressible in OPUS and vice versa. We even think that it is possible to define our calculus on top of λN. However, we do feel that λN-calculus is too low-level as a model for expressing and investigating object-oriented concepts.

Another calculus that seems close to ours is the one proposed by (Abadi et al., 1994). Just like OPUS, it is not based on lambda-calculus, although it subsumes lambda-calculus. But as opposed

[5] This also compromises object-based encapsulation, because it allows a method to be selected and temporarily stored somewhere, and later on this method can be retrieved and applied in a totally different context, gaining access to the object's encapsulated part without passing through its interface.

to OPUS, it contains a notion of recursion and self-reference in the basic syntax. There seem to be a lot of similarities between both calculi, although a more thorough comparison remains to be done.

6. Conclusions and future work

In this paper, we proposed an object-oriented programming calculus with a very simple syntax and only three reduction rules. Nevertheless we believe it can model all essential features of object-orientedness. We have shown that it is not necessary to include notions like recursion, numerals, inheritance and classes in the basic syntax, because they can be modelled straightforwardly using the basic syntactic constructs: objects, encapsulation, message sending and incremental modification. In a similar way state is not essential, because it can be modelled using constant methods and an incremental modification operator. Nevertheless it could be useful to find out how difficult it is to directly include state into our basic syntax and reduction rules, because this will strongly simplify the examples. However, it seems to be a non-trivial problem.

A problem still present in our calculus is that argument passing compromises object-based encapsulation to a certain extent. We believe however that these difficulties can be resolved in a reasonably straightforward way by providing a higher-level syntax. There are some problems with inheritance as well, due to the implicit interaction of encapsulation and inheritance. For example, in the approach presented in this paper, we were only able to model a form of encapsulated inheritance. Sometimes however non-encapsulated inheritance could also be useful. To solve this, one could uncouple the encapsulation mechanism from the inheritance mechanism by making the encapsulation operator explicit.

Another disadvantage of our calculus is that object-identity cannot be modeled with it because object identity is only meaningful when several distinct objects live together in the same space. This is clearly not the case in OPUS, since our calculus can only reduce one object (possibly containing different nested subobjects) at a time.

In the future we plan to show that OPUS subsumes lambda-calculus, thus proving its computational completeness. We have strong convictions that our calculus is computationally complete, because numerals can be modelled in it without any problems. It is also necessary to prove the consistency of OPUS, by means of a property similar to the Church-Rosser theorem for lambda-calculus. Showing the soundness and completeness of OPUS might be worthwhile too. OPUS might be extended to incorporate typing rules as well, in analogy with the extension of lambda-calculus to typed lambda-calculus.

7. Acknowledgements

We wish to express our gratitude to Niels Boyen and Wolfgang De Meuter for the heavy discussions, interesting suggestions and numerous remarks, and to the whole AGORA-group for their help and support. We would also like to thank our promoter Theo D'hondt and several anonymous referees for their comments.

8. References

(Abadi et al., 1994) M. Abadi and L. Cardelli. *A Theory of Primitive Objects: Untyped and First-order Systems*. Theoretical Aspects of Computing Software '94 Proceedings, Springer-Verlag, 1994

(Bracha et al., 1990) G. Bracha and W. Cook. *Mixin-based Inheritance.* Joint OOPSLA/ECOOP '90 Conference Proceedings, pp. 303-311, ACM Press, 1990

(Canning et al., 1989) P. Canning, W. Cook, W. Hill and W. Olthoff. *Interfaces for Strongly-Typed Object-Oriented Programming.* OOPSLA '89 Conference Proceedings, pp. 457-467, ACM Press, 1989

(Cardelli, 1988) L. Cardelli. *A semantics of multiple inheritance.* Information and Computation 76, pp. 138-164, Academic Press, 1988

(Cardelli et al., 1989) L. Cardelli and J. Mitchell. *Operations on Records.* Proceedings on Mathematical Foundations of Programming Semantics, LNCS 442, Springer-Verlag, 1989

(Castagna et al., 1992) G. Castagna, G. Ghelli and G. Longo. *A Calculus for Overloaded Functions with Subtyping.* LISP and Functional Programming '92 Conference Proceedings, pp. 182-192, ACM Press, 1992

(Cook, 1989) W. Cook. *A Denotational Semantics of Inheritance.* Ph.D.-Thesis, Brown University, 1989

(Cook et al., 1989) W. Cook and J. Palsberg. *A Denotational Semantics of Inheritance and its Correctness.* OOPSLA '89 Conference Proceedings, pp. 433-444, ACM Press, 1989.

(Dami, 1994) L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* Ph.D.-Thesis, University of Geneva, 1994

(de Bruijn, 1972) N. de Bruijn. *Lambda-Calculus with Nameless Dummies, a Tool for Automatic Formula Manipulation.* Indag. Mat. 34, pp. 381-392, 1972

(Hofmann et al., 1993) M. Hofmann and B. Pierce. *A Unifying Type-Theoretic Framework for Objects.* Journal of Functional Programming 1, Cambridge University Press, 1993

(Milner, 1991) R. Milner. *The Poliadic π-calculus: A tutorial.* Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991

(Nierstrasz, 1992) O. Nierstrasz. *Towards an Object Calculus.* ECOOP Workshop on Object-Based Concurrent Computing, LNCS 612, Springer-Verlag, 1992

(Pierce et al., 1994) B. Pierce and D. Turner. *Simple Type-Theoretic Foundations for Object-Oriented Programming.* Journal of Functional Programming, 1994

(Steyaert, 1994) P. Steyaert. *Open Design of Object-Oriented Languages, a Foundation for Specialisable Reflective Language Frameworks.* Ph.D.-Thesis, Vrije Universiteit Brussel, 1994

(Steyaert et al., 1993) P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas and M. Van Limbergen. *Nested Mixin-Methods in Agora.* ECOOP '93 Conference Proceedings, pp. 197-219, Springer-Verlag, 1993

(Snyder, 1987) A. Snyder. *Inheritance and the Development of Encapsulated Software Components.* Research Directions in Object-Oriented Programming, MIT Press, 1987

(Wegner et al., 1988) P. Wegner and S. Zdonik. *Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like.* ECOOP '88 Conference Proceedings, pp. 55-77, Springer-Verlag, 1988

OPUS: a Calculus for Modelling Object-Oriented Concepts ERRATA

Tom Mens, Kim Mens, Patrick Steyaert

Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: { tommens@is1 | **kimmens@is1** | prsteyae@vnet3 }.vub.ac.be
WWW: http://progwww.vub.ac.be/prog/pools/opus/opus.html
Fax: +32 2 629 3495

This text contains a list of errata still present in our paper in the OOIS'94 Conference Proceedings. The changes made are printed in bold.

3.3. Reduction rules

The notation → used in the reduction rules means "...reduces in one step to...", while ⇒ ("...reduces in **at least one step** to...") will be used in the subsequent examples to denote the transitive closure of →.

3.4. Evaluation in a context

Evaluation of a method $\lambda N=E_1$ in a context leaves the method body unchanged. One might say that all ~~**free**~~ names in E_1 are bound by the λ.

3.5. Dealing with recursion

To simplify the examples in the rest of this paper we will abbreviate expressions of the form $\sigma_{\mathbf{self}}$ unfold:[par=E |] to σE.

4.3. Class-based inheritance

Subclasses of the point class can be created by incrementally modifying **it** for example with a MODIFIER implementing a move-method that only moves the x-coordinate while keeping the y-value unaltered.

In section 3.3 a problem still present in our approach and a possible solution are suggested:

> A problem still present in our approach is that the argument passing mechanism as proposed in this paper jeopardises encapsulation. The reason for this is that in rule 2b arguments have precedence over private methods. For example
>
> [λgetx=x λgety=y | x=1 y=2] getx:[x=2 |]
>
> yields 2 instead of the expected result 1. This problem can be solved by adding the restriction that argument names and private method names should be disjoint.

However, there is a much more simple solution to the above problem. Instead of giving arguments precedence over private methods in rule 2b, we have to do the opposite and give private methods precedence over arguments. So rule 2b becomes:

<u>Rule 2</u>: Message sending to an incrementally modified object

b) Method execution

$$\begin{array}{lll} (\,E_1 + [\,R\ \ \lambda N{=}E_2 \mid F\,]\,)\ N{:}E \ \rightarrow & \{\,(\,\mathbf{E} + \mathbf{F}\,)\,\}(E_2) & \text{if F is no Record} \\ & \{\,(\,\mathbf{E} + \mathbf{[\,F \mid\,]}\,)\,\}(E_2) & \text{if F is a Record} \end{array}$$

Notice that this will introduce some changes in the examples of the paper as well, but all these changes are very straightforward.