OPUS : a Formal Approach to Object-Orientation

accepted at FME'94
Tom Mens, Kim Mens, Patrick Steyaert

Department of Computer Science, Faculty of Sciences
Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium
e-mail: {tommens@is1|we34154@is1|prsteyae@vnet3}.vub.ac.be

Abstract. OPUS is an elementary calculus that models object-orientation. It expresses in a direct way the crucial features of object-oriented programming such as objects, encapsulation, message passing and incremental modification. This is illustrated by numerous examples. Thanks to the way objects are constructed in this calculus, we can deal with self-reference, recursion and even mutual recursion in a straightforward way. We also illustrate that it is relatively easy to model different kinds of inheritance mechanisms. Finally, we argue to which extent our calculus can be used for modeling and investigating object-oriented concepts.

1 Introduction

Until now there has not been given a satisfactory formal model for the object-oriented programming paradigm. Therefore we think it is meaningful to construct a calculus that provides a formal foundation for a whole range of object-oriented programming languages (OOPLs), just like λ-calculus represents the theoretical backbone of functional languages. More specifically, we would like to find those key-features that are essential for object-oriented programming and build a calculus that allows us to model a wide range of object-oriented concepts using only a very restricted set of syntactic constructs. In section 6 we will discuss how such a formal calculus can be used as a tool for investigating and relating object-oriented concepts.

Our calculus is based on the extended abstract [18] presented at an ECOOP workshop. An early version of this calculus can be found in the Ph.D.-thesis of Patrick Steyaert [19]. Because the calculus provides a formal foundation in which object-oriented features can be expressed, we will call it OPUS, which is an acronym for Object-oriented Programming calculUS. The main features of OPUS are:

• OPUS explicitly employs names for message passing. As already discussed in numerous papers, the use of names greatly simplifies the modeling of object-oriented systems. Examples of this are: the λ-calculus augmented with records (cf. [4, 5]), Milner's π-calculus for describing concurrent computations [11] and λ-calculus augmented with names, combinations and alternations [7].

• In [12], Oscar Nierstrasz argues that encapsulation is one of the most fundamental concepts of object-orientedness, and that all object-oriented mechanisms and approaches exploit this idea to various ends. For this reason, we provide an explicit encapsulation mechanism in our calculus, allowing to create strongly encapsulated objects containing a public part and an encapsulated private part.

• Because we want our calculus to model basic object-oriented features such as inheritance and subclassing in a straightforward way, we introduce an incremental modification mechanism similar to the one proposed in [20].
• As opposed to [1], we believe that no explicit syntactic provisions should be made for recursion (e.g. "self" sends). We will show how recursive objects can be modeled in a straightforward way using only the basic syntax of the calculus.
• With [6, 19], we advocate the use of 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.
• Last but not least, with [9] we agree that our object-oriented model should satisfy the property of homogeneity: "any entity is an object, and the unique control structure is message passing".

In the following section we define the syntax and reduction rules of our calculus, discuss the intuitive meaning of the various syntactical constructs, and explain them by means of examples. Section 3 shows how we can deal with recursion by introducing a notion of self-reference, and presents some illustrative examples such as updatable objects, mutual recursion, recursive data structures and an example of iteration. In the fourth section we illustrate how class-based as well as mixin-based inheritance can be modeled in our calculus. The section that follows provides some pointers to related work. Section 6 indicates several topics for further research, and discusses the use of OPUS as a formal foundation for OO. Finally, the last section summarizes the results achieved in this paper, and draws a conclusion.

2 Syntax of OPUS

2.1 Context Free Grammar

Definition: An OPUS-expression is an element of the language generated by the following context free grammar (in EBNF-notation) where the start symbol is Expression, all terminal symbols are denoted between double quotes and Name is also a terminal symbol.

Expression	⇒	Object | Name | MessageSend | Composition
Object	⇒	Compound | Simple
MessageSend	⇒	Expression Name
Composition	⇒	"**(**" Expression "**+**" Expression "**)**"
Compound	⇒	"**<**" Expression "**/**" Expression "**>**"
Simple	⇒	"**[**" List "**]**"
List	⇒	[AssocList]
AssocList	⇒	[AssocList "**,**"] Assoc
Assoc	⇒	VarAssoc | MethodAssoc
VarAssoc	⇒	Name "**=**" Expression
MethodAssoc	⇒	Name "**#**" Expression

Notation:
1) In the rest of this paper, we will assume that N denotes a meta-variable of type Name, E and E_i are meta-variables of type Expression, L and L_i denote meta-variables of type List, and A is a meta-variable of type Assoc.

2) We introduce two notations for reducing expressions. → means "...reduces in one step to...", while **Error!**

Convention: In all the examples in this paper, a Name will always be written in lowercase. (Words written in uppercase will be used as macro-definitions for predefined objects.)

Definition: An *N-association* is an expression of the form N=E or N#E.

In the following subsections we give an intuitive explanation of the syntax described by the context free grammar, enumerate the reduction rules that will be used to reduce expressions to normal form, and illustrate them by means of examples.

2.2 Objects and Encapsulation

In our calculus, we have chosen for an object in its most general form to consist of a public part and a private part. These objects will be called *compound objects*, and are denoted as < Public / Private >. Both public and private parts can contain methods as well as instance variables. The *public part* contains the public methods and instance variables visible for other objects. The *private (or encapsulated) part* on the other hand consists of variables and methods that are only used to *implement* public methods.

When invoking a public method in a compound object, unbound names in this method are looked up in the encapsulated part of the object (this is called *private method invocation*). Instance variables on the other hand are not bound in the private part. Because of this difference between methods and instance variables, we need a way to distinguish them. Method associations will be represented by a name, followed by a #, followed by an expression (e.g. getx#x), while variable associations are represented by a name, followed by an equality symbol, followed by an expression (e.g. x=xval). This difference will not be visible to the sender of a message, because the principle of homogeneity demands that both methods and instance variables are invoked using exactly the same syntax (message passing is an atomic operation).

Public and private parts in their simplest forms are just association lists (records) of instance variables and methods. These records are called *simple objects* and are denoted between square brackets, while the associations are separated by commas, as for example in [x#a,y=b,z=c].

2.3 Message Passing

For the moment we will only work with messages without arguments. Later we will see how message passing with arguments can be simulated by private attributes, or by means of our composition operator. Passing a message N to an expression E is denoted by E N. The way this message N is evaluated depends on the form of E.

Simple objects.

Passing a message to a simple object is similar to record selection. If the association corresponding to the message is an instance variable, message passing coincides with selecting the value of this variable. E.g. [x#a,y=b,z=c] y reduces to b. There is a problem however if the association corresponding to the message is a

method. Normally we would "execute" this method by looking up all unbound names in the private part of the object. But since a simple object has no private part, this is impossible. Therefore, when a name is sent to a simple object, and the association corresponding to this name is of the form N#E, we do not reduce the expression any further. E.g. [x#a,y=b,z=c] x reduces to [x#a] x which is an expression in normal form. All of this can be formalized in the following rule:

Rule 1: Message passing to a simple object

[L , N=E] N	→	E	*(variable selection)*
[L , A] N	→	[L] N	if A is no N-association

Compound objects.

Message passing to a compound object selects an attribute in the public part of the object. Again there will be a major difference between method selection and instance variable selection. In the case of instance variables, we simply select the value of the corresponding association. E.g. < [x#a,y=b,z=c] / [a=d] > y will reduce to b. In the case where the attribute is a method however, the body of the method has to be evaluated in the context of the private part. For evaluating an expression E_1 in a context E we will use the notation $\{E\}(E_1)$.[1] E.g. < [x#a,y=b,z=c] / [a=d] > x reduces to {[a=d]}(a), which can be further reduced to d, using the definition of evaluation in a context that will be given later.

Rule 2: Message passing to a compound object

< [L , N=E_1] / E > N	→	E_1	*(variable selection)*
< [L , N#E_1] / E > N	→	$\{E\}(E_1)$	*(method execution)*
< [L , A] / E > N	→	< [L] / E > N	if A is no N-association

In this rule we observe that passing a message to an object results in searching the association list in the public part *from right to left* for an attribute corresponding to the message name, and then executing the corresponding method or fetching the corresponding value. If no attribute is found, the expression cannot be reduced any further. For example, <[x=a,y#y]/[y=b]> z reduces to <[]/[y=b]> z by applying rule 2 twice. This new expression is in normal form, and intuitively corresponds to a "message not understood" error[2].

Convention: If two different attributes corresponding to the same name occur in an association list, only the last one is significant (due to the right-to-left strategy for searching attributes). This automatically solves any problems concerning double use of names: only the last occurrence of a name is relevant. For this reason we introduce the convention that, if more than one attribute corresponding to a given name occurs in a simple object, we only write the rightmost occurrence[3]. E.g. [x=a,z=b,y#y,y#z,x#x] will be abbreviated to [z=b,y#z,x#x].

---

[1] This notation is not part of the syntax, but can be considered as some kind of meta-level reduction scheme (similar to the substitution mechanism in lambda-calculus).

[2] In contrast to [9], we don't make explicit use of error messages. In a future version of this calculus we are planning to deal with error messages at a semantic level.

[3] The main argument for introducing this shorthand notation is that it makes the examples easier to read and understand.

It is important to note that *rule 2 provides for both method selection and method application in one single derivation step*! In most models based on λ-calculus with records this is not the case because method selection and method application are considered distinct operations. This however compromises object-based encapsulation, since 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 parts without passing through its interface.

Next, we will show how evaluation in a context can be defined formally. Intuitively, evaluating an expression in some context distributes over all sub-expressions, except for method associations (because those are evaluated in the encapsulated part of the object to which they belong, according to the previous rule). Evaluating a name in a context coincides with sending the name to that context. Note that the context in which an expression is evaluated can be an arbitrary object. The exact definition of evaluation of an expression in a context is given below:

{E}(E_1 N)	equals	{E}(E_1) N
{E}((E_1 + E_2))	equals	({E}(E_1) + {E}(E_2))
{E}(< E_1 / E_2 >)	equals	< {E}(E_1) / {E}(E_2) >
{E}([L])	equals	[{E}(L)] if L is not empty
{E}([])	equals	[ ]
{E}(L,A)	equals	{E}(L),{E}(A)
{E}(N=E_1)	equals	N={E}(E_1)
{E}(N#E_1)	equals	N#E_1
{E}(N)	equals	E N

Evaluating a simple object in a context E yields a new simple object where *only the instance variables are evaluated in the context*, while the methods remain unaltered (because they are not evaluated in the context, but in the private part of the object of which they are part). Indeed {E}(N=E_1) is defined as N={E}(E_1), whereas {E}(N#E_1) simply yields N#E_1.

To illustrate the above definition, consider the following example, where E denotes an arbitrary expression:

```
{E}( < [x=a,y=b,z#d] / [a=e] > x )
=  {E}( < [x=a,y=b,z#d] / [a=e] > ) x
=  < {E}( [x=a,y=b,z#d] ) / {E}( [a=e] ) > x
=  < [x={E}(a),y={E}(b),z#d] / [a={E}(e)] > x
=  < [x=E a,y=E b,z#d] / [a=E e] > x
```

In the rest of the examples presented in this paper, we will always immediately give the resulting expression when evaluating an expression in a context, and omit all the intermediate steps.

2.4 Examples

In this section, we will try to illustrate the concepts discussed earlier by expressing conditionals in our calculus in both a functional and an object-oriented way.

Functional Conditionals.
A *functional* IF-object needs three arguments: a condition, a then-part and an else-part. On invocation of the message res it returns the value of the then-part if the condition is true, and the value of the else-part otherwise. If we define TRUE and FALSE as objects that return the values true or false when the message istrue is sent, the definition[4] of IF can be given as follows:

```
TRUE  := [istrue#true]        FALSE := [istrue#false]

IF := [res#< cond / [true=then,false=else] > istrue]
```

To deal with the arguments, we extend this IF-object to a compound object by encapsulating a private part containing the values of the variables cond, then and false. Sending a res message to this extended object yields the expected result, as can be seen in the following derivation that uses the message passing rules described earlier.

```
< IF / [cond=TRUE,then=a,else=b] > res

= <  [res#<cond/[true=then,false=else]>  istrue]  /
     [cond=TRUE,then=a,else=b] > res                          (def. IF)

→,+  {[cond=TRUE,then=a,else=b]}
     (<cond/[true=then,false=else]>  istrue)                  (rule 2)

= <  [cond=TRUE,then=a,else=b] cond /
     [true=[cond=TRUE,then=a,else=b] then,
      false=[cond=TRUE,then=a,else=b] else] > istrue          (def. {})

→,+  < TRUE / [true=a,false=b] > istrue                       (rule 1)

=    < [istrue#true] / [true=a,false=b] > istrue              (def. TRUE)

→,+  {[true=a,false=b]}(true)                                 (rule 2)

=    [true=a,false=b] true                                    (def. {})

→,+  a                                                        (rule 1)
```

We can easily extend this example to create functional boolean operators NOT, AND and OR.

```
NOT   := [res#< arg / [true=FALSE,false=TRUE] > istrue]

AND   := [res#< first / [ true=<second/[true=TRUE,false=FALSE]> istrue,
                          false=FALSE ] > istrue]

OR    := [res#< first / [ false=<second/[true=TRUE,false=FALSE]> istrue,
                          true=TRUE ] > istrue]
```

One can check that the following reductions are valid:

---

[4] 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" (cf. macro-definitions).

< NOT / [arg=TRUE] > res	$\rightarrow,^{+}$ FALSE
< AND / [first=TRUE,second=FALSE] > res	$\rightarrow,^{+}$ FALSE
< OR / [first=FALSE,second=TRUE] > res	$\rightarrow,^{+}$ TRUE

Object-oriented Conditionals.
Following the object-oriented approach, TRUE- and FALSE-objects are defined as simple objects that return then (respectively else) on invocation of the method if.

TRUE := [if#then] FALSE := [if#else]

Using these boolean objects, a conditional expression will look as follows:

< TRUE / [then=a,else=b] > if	
= < [if#then] / [then=a,else=b] > if	*(def. TRUE)*
$\rightarrow,^{+}$ {[then=a,else=b]}(then)	*(rule 2)*
= [then=a,else=b] then	*(def. {})*
$\rightarrow,^{+}$ a	*(rule 1)*

We will now try to generalize these boolean objects, so that they understand not only if-messages, but also messages not, and and or:

TRUE := [if#then, and#arg, or#TRUE, not#FALSE]

FALSE := [if#else, and#FALSE, or#arg, not#TRUE]

Here a problem arises, because the definition of both objects is defined in terms of themselves and the other object. This is not allowed, because we are not (yet) able to deal with recursive (and even mutual recursive!) definitions. Later we will explain how to solve this problem.

2.5 Currying of Private Attributes

For modeling objects, the most important advantage over λ-calculus with records (cf. [4, 5]) is that the public and private parts of a compound object need not be simple objects, but can be compound objects themselves! This is essential on the one hand to model private methods, and on the other hand to have some form of curried binding of instance variables (i.e. private attributes that are bound in different stages). The rule needed to express this currying principle is very simple.

Rule 3: Currying

$$< < E_1 \ / \ E_2 > / \ E_3 > \ \rightarrow \ < E_1 \ / < E_2 \ / \ E_3 > >$$

Using the currying principle, it is very easy to define a conditional in which the then and else part are already filled in, while the condition has to be provided for later on. This can be achieved by extending the previously defined IF-object as follows:

< IF / [then=a,else=b,cond#cond] >

Now the only thing left to do is providing a condition for this extended object.

< < IF / [then=a,else=b,cond#cond] > / [cond=TRUE] >

Finally, if we send the message res to this compound object, derivation leads to the expected result:

< < IF / [then=a,else=b,cond#cond] > / [cond=TRUE] > res

= < < [res#<cond/[true=then,false=else]> istrue] /
[then=a,else=b,cond#cond] > / [cond=TRUE] > res *(def. IF)*

$\rightarrow,^{+}$ < [res#<cond/[true=then,false=else]> istrue] /
< [then=a,else=b,cond#cond] / [cond=TRUE] > > res *(rule 3)*

$\rightarrow,^{+}$ {< [then=a,else=b,cond#cond] / [cond=TRUE] >}
(<cond/[true=then,false=else]> istrue) *(rule 2)*

$\rightarrow,^{+}$ < [cond=TRUE] cond / [true=a,false=b] > istrue *(def. {}, rule 1&2)*

$\rightarrow,^{+}$ < TRUE / [true=a,false=b] > istrue *(rule 1)*

$\rightarrow,^{+}$ a *(see example of section 2.4)*

This example clearly illustrates that it is possible to provide the required arguments for a given method in different stages. First the then and else part were given, and next the condition was added. Argument passing can be modeled using exactly this mechanism: arguments are bound to an object in supplement to the already bound instance variables. In section 2.7 we will present an alternative approach.

2.6 Object Composition

In many cases it will be useful to have some kind of mechanism that allows us to compose two simple objects into a resulting object. This "composition" of objects operationally corresponds to the concatenation of records, where all the attributes of the second record are added to the first one, and if there is an attribute that already occurred in the first record, its value will simply be "overwritten" by the corresponding value in the second record (due to the right-to-left strategy for attribute lookup). This mechanism will prove useful when dealing with inheritance.

Rule 4: Composition of objects

$[\ L_1\ ] + [\ L_2\ ] \quad \rightarrow \quad [\ L_1\ ,\ L_2\ ]$

Notice that rule 4 is only defined for simple objects, although the syntax allows the components E_1 and E_2 of a composition (E_1+E_2) to be arbitrary expressions. This is because it might be possible to eventually reduce these expressions to simple objects by means of the reduction rules. If this is not possible, rule 4 cannot be applied.

This composition operator can be used as an incremental modification mechanism. For example, suppose we have a POINT-object that understands the messages getx, gety and set.[5]

```
POINT := [ getx#x, gety#y, set#<self/[self=self,x=x,y=y]> ]
```

If we want to modify this object to obtain a CIRCLE-object, by adding a getr message, and modifying the set message, then this can be done by composing the POINT-object with the following MODIFIER:

```
MODIFIER := [ getr#r, set#<self/[self=self,x=x,y=y,r=r]> ]
```

The resulting object is:

```
CIRCLE := ( POINT + MODIFIER )
       = [  getx#x,
            gety#y,
            set#<self/[self=self,x=x,y=y]>,
            getr#r,
            set#<self/[self=self,x=x,y=y,r=r]>  ]
```

This new object contains two set-messages, of which only the last one is relevant, because it "overwrites" the first one. Using the convention that we only write those methods and instance variables that are relevant (in case of double use of names), the previous object can be written somewhat simpler:

```
CIRCLE := [ getx#x, gety#y, getr#r, set#<self/[self=self,x=x,y=y,r=r]> ]
```

2.7 Message Passing with Arguments

Until now, we have only dealt with message passing without arguments for reasons of simplicity. Nevertheless we have seen that it is possible to simulate arguments by means of private attributes. This approach compromises to some extent the encapsulation of objects, because private attributes of an object can be overwritten by arguments with the same name. In a future version of our calculus we will try to solve this problem.

To make it easier to understand the examples, we introduce a new syntactic construction for dealing with *message passing with arguments*, and give a corresponding derivation rule for message passing. To allow argument passing, the syntax of the context free grammar needs to be modified as follows:

MessageSend	⇒	WithoutArguments | WithArguments
WithoutArguments	⇒	Expression Name
WithArguments	⇒	(Expression Name ":" Expression)

For message passing with arguments, parentheses are needed to avoid ambiguity. However, we will adopt the convention that message passing associates to the left. In this way we can drop most of the parentheses when dealing with messages with

[5] This example will be explained more into detail in section 3.2.

arguments. E.g. $E_1\ N_2{:}E_2\ N_3{:}E_3$ means $(E_1\ N_2{:}E_2)\ N_3{:}E_3$ instead of $E_1\ N_2{:}(E_2\ N_3{:}E_3)$.

A new rule for passing messages with arguments needs to be introduced. It basically corresponds to an "extend-then-send" construction, i.e. first the private part of the object is extended with the arguments of the message, and next the message is sent to this extended object.

Rule 5: Message passing with arguments
$< E_1 / E_2 > N{:}E \rightarrow < E_1 / (E_2 + E) > N$

Furthermore, in order to deal with this new syntactic construction, the definition of evaluation in a context needs to be extended with:

$\{E\}(E_1\ N{:}E_2)$	equals	$\{E\}(E_1)\ N{:}\{E\}(E_2)$

3 Recursion

3.1 Dealing with self-reference

In most object-oriented languages (e.g. Smalltalk) it is common that an object can modify its own (public) methods and instance variables using a self-reference that returns the object itself. We will show that it is very easy to implement a notion of self-reference in our calculus: the self instance variable will be a variable that behaves in exactly the same way as all other instance variables.

Before continuing we introduce the following notation for creating a self-referring expression given an initial expression E:

$$\sigma E := < E / [self=E] >$$

Informally this means that a self-referring expression is defined by creating a compound object where the public part contains the initial expression, while the private part contains a variable self that refers to this expression. A special case of this is the object $\sigma self = <self/[self=self]>$, containing an unbound variable self that will be bound later on thanks to the scoping rules.

Notice that the operator σ only unfolds one level of recursion, whereas a fixed-point operator (such as the Y-operator in λ-calculus) corresponds to an infinite recursive unfolding[6]. The advantage of the use of infinite recursion is that we only have to apply the fixed-point operator once, and that any future reference to the object will yield the same object. However, this implies that the object cannot be updated. Therefore we prefer to use our operator σ, and explicitly rewrite the operator each time we want to expand another level.

---

[6] It is possible to define such a fixed-point operator in our calculus, but that is beyond the scope of this paper.

An example of the use of self-reference is given below, where we construct a compound object with only one method s in its public part, such that invocation of this method returns the object itself.

Property: σ[s#σself] s $\rightarrow,^{+}$ σ[s#σself]

Proof:

σ[s#σself] s = < [s#σself] / [self=[s#σself]] > s	(def. σ)
$\rightarrow,^{+}$ {[self=[s#σself]]}(σself)	(rule 2)
= {[self=[s#σself]]}(<self/[self=self]>)	(def. σ)
$\rightarrow,^{+}$ < [s#σself] / [self=[s#σself]] >	(def. {} & rule 1)
= σ[s#σself]	(def. σ)

In a similar way, if we reduce the expression σ[s#σself s] s we obtain exactly the same expression again. This means that deriving this expression leads to an infinite reduction sequence without ever reaching a normal form. The proof of this property is similar to the previous one.

Property: σ[s#σself s] s $\rightarrow,^{+}$ σ[s#σself s] s

3.2 Updatable Objects

Now we will show how the σ-operator can be used to deal with updatable objects. We 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). In order to obtain the expected result, the set-method should be invoked by means of a message with two arguments x and y representing the new x- and y-values.

POINT := [getx#x,gety#y,set#<self/[self=self,x=x,y=y]>]

The compound object σPOINT will satisfy our requirements, as we can see from the following reduction:

(σPOINT set:[x=1,y=2]) gety	
= (< POINT / [self=POINT] > set:[x=1,y=2]) gety	(def. σ)
$\rightarrow,^{+}$ < POINT / ([self=POINT] + [x=1,y=2]) > set gety	(rule 5)
$\rightarrow,^{+}$ < POINT / [self=POINT,x=1,y=2] > set gety	(rule 4)
$\rightarrow,^{+}$ {[self=POINT,x=1,y=2]}(<self/[self=self,x=x,y=y]>) gety	(rule 2)
$\rightarrow,^{+}$ < POINT / [self=POINT,x=1,y=2] > gety	(rule 1)
$\rightarrow,^{+}$ {[self=POINT,x=1,y=2]}(y)	(rule 2)
$\rightarrow,^{+}$ 2	(rule 1)

3.3 Mutual Recursion

In an earlier example, we needed a mechanism to deal with mutual recursion. The solution to this is similar to plain recursion, but slightly more complicated, because we have to deal with two different objects interacting with each other. For this reason, we do not only need a self instance variable, referring to the object itself, but also an instance variable other referring to the object with which the given object is mutually recursive. Reviewing the proposed object-oriented definitions of TRUE and FALSE in an example earlier in this paper,

```
TRUE    := [if#then,and#arg,or#TRUE,not#FALSE]

FALSE   := [if#else,and#FALSE,or#arg,not#TRUE]
```

we observe that only minor changes have to be made, by introducing a SELF and OTHER object, replacing them for TRUE and FALSE in the previous definitions, and adding two private instance variables self and other, containing the object itself and its mutually recursive object respectively.

```
TRUE    := < TRUE' / [self=TRUE',other=FALSE'] >

FALSE   := < FALSE' / [self=FALSE',other=TRUE'] >

TRUE'   := [if#then,and#arg,or#SELF,not#OTHER]

FALSE'  := [if#else,and#SELF,or#arg,not#OTHER]

SELF    := < self / [self=self,other=other] >

OTHER   := < other / [self=other,other=self] >
```

It is easy to see that these definitions yield the expected results, as can be verified by means of the following derivations:

$$\text{TRUE not} \rightarrow,^{+} \text{FALSE}$$

$$\text{TRUE and:[arg=FALSE]} \rightarrow,^{+} \text{FALSE}$$

$$\text{TRUE or:[arg=FALSE]} \rightarrow,^{+} \text{TRUE}$$

3.4 Recursive Data Structures

In this section we illustrate how recursive data structures can be simulated in our calculus. More specifically we will show how to deal with circular linked lists and double linked lists.

Circular linked lists.
These are simply lists of records containing a value part and a next-pointer referring to the next record. Moreover, because the list is circular, the next-pointer in the last record should point to the first record. In figure 1, the schematic representation of a linked list with three items (1, 2 and 3) is given.

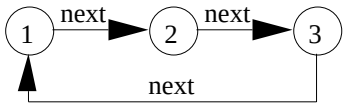

Fig. 1. Circular linked list

The way to simulate this in OPUS is rather straightforward:

```
CIRCULAR := [ val=1,
              next#<[  val=2,
                       next#[  val=3,
                               next#σself]
                       ] / [self=self] > ]
```

One can easily check that this corresponds to a circular linked list, by looking at the following derivations:

σCIRCULAR next next next $\rightarrow,^{+}$ σCIRCULAR

σCIRCULAR next val $\rightarrow,^{+}$ 2

Double linked lists.
These are lists of records where each record contains a value part, a next-pointer referring to the next record, and a prev-pointer referring to the previous record. The next-pointer of the last record and the prev-pointer of the first record both point to nil. In figure 2 we give an example of such a double linked list.

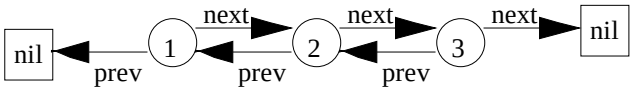

Fig. 2. Double linked list

The OPUS-expression corresponding to this list is the following one:

```
DOUBLE := [  val=1,
             prev#nil,
             next#<[  val=2,
                      prev#σself,
                      next#<[  val=3,
                               prev#σself next,
                               next=nil  ]  /  [self=self]>
                      ] / [self=self] > ]
```

Again one can see that this expression does what is expected:

σDOUBLE next next prev prev $\rightarrow,^{+}$ σDOUBLE

3.5 An Iterative Example

To illustrate that it is possible to deal with iteration in our calculus, we show how to express the classical functional example of calculating the factorial.

```
FAC    := [res#< IF / [  cond=n iszero, then=fac,
                         else=σself res:[n=n-1,fac=fac*n] ] > res]
```

It is easy to see that this definition of factorial[7] works as expected, by observing that it corresponds to the following intuitive definition:

```
FAC(n, fac) = if n iszero then fac else FAC(n-1, fac*n)
```

The argument n contains the number of which the factorial has to be computed, while the argument fac is used as an accumulator to iteratively accumulate the value of the factorial. We start with an initial value of fac=1 and at each step this value is multiplied with a new value, as can be seen in the following reduction that calculates the factorial of 3:

```
( σFAC res:[n=3,fac=1] )

→,+ < FAC / [self=FAC,n=3,fac=1] > res

→,+ < FAC / [self=FAC,n=2,fac=3] > res

→,+ < FAC / [self=FAC,n=1,fac=6] > res

→,+ < FAC / [self=FAC,n=0,fac=6] > res

→,+ 6
```

4 Dealing with Inheritance

In this section, we will illustrate that it is possible to model different kinds of inheritance mechanisms in our calculus in a straightforward manner. First we illustrate how to deal with class-based inheritance; secondly we do the same for mixin-based inheritance. Intuitively it should be clear that inheritance can be expressed in our calculus, because in [20] it is shown that inheritance can be modeled as an incremental modification mechanism: a parent P is transformed by means of a modifier M to form a result R = P + M. Now if we assume that R, P and M are simple objects, then this mechanism exactly corresponds to our own composition operator!

4.1 Class-based Inheritance

Intuitively, classes can be defined as templates from which objects are created. In our approach, we will use the public part as template, i.e. two different instances of a given class will have the same public part, but can have different private parts. For example, instance variables x and y will differ between different instances of the same POINT-class, while the getx, gety and set methods will be the same for all copies.

Taking care never to turn away from our basic principle that "everything is an object", we will formally define a class as a compound object. The private part of this compound object will consist of:

---

[7] In this definition it is important to note that the arithmetic operations iszero, +, * and - can be defined as OPUS-expressions, but that is beyond the scope of this paper.

- an inst variable, containing the value of the public part of each instance of a class;
- a self variable, referring to the class itself (to allow recursive class-methods);
- a super variable, referring to the superclass of a given class (to allow code reuse).

For example we will define a POINTCLASS with instances of the form POINT, and a BOUNDEDCLASS which is a subclass of POINTCLASS, and has instances of the form BOUNDEDPOINT.

```
POINTCLASS    := < POINTPUBLIC / [  inst=POINT,
                                    self=POINTPUBLIC,
                                    super=root  ]  >

BOUNDEDCLASS  := < BOUNDEDPUBLIC / [  inst=BOUNDEDPOINT,
                                      self=BOUNDEDPUBLIC,
                                      super=POINTPUBLIC ] >
```

The public part of a class will consist of:
- a new method, needed to create new instances of the given class. As for classes, instances can refer to themselves using a self-variable, but they can also refer to their class using a class-variable.
- possibly some other class-methods

As an example, we define the public parts of the POINTCLASS and BOUNDEDCLASS respectively. Instances of BOUNDEDCLASS are created by delegating the new message to its superclass.

```
POINTPUBLIC :=     [  new#<inst/[self=inst,class=self]>  ]

BOUNDEDPUBLIC :=   [  new#<super/[inst=inst,self=self,super=super]>  new  ]
```

The idea behind class-based inheritance is that a new class is defined by specifying how it differs from an already existing class. This principle of incremental modification is simply implemented by defining the instance of a given subclass as an instance of its parent, modified by a certain modifier:

inst=(PARENTINST + MODIFIER)

For example, POINT-instances understand the messages getx, gety, set and move. BOUNDEDPOINT-instances inherit these methods of POINT, while the set message is overwritten by another one that tests whether the new x-value exceeds the upper bound 5.

```
POINT := [  getx#x,  gety#y,
            set#< self / [self=self,class=class,x=x,y=y] >,
            move#(  <self/[self=self,class=class]>  set:[x=x,y=y]  )  ]

MODIFIER := [  set#< x ≤ 5 / [  then=<self/[self=self,class=class,x=x,y=y]>,
                                else=error  ]  >  if  ]

BOUNDEDPOINT := (POINT + MODIFIER)
```

The correctness of all these definitions will be illustrated by the following two reductions. The first one creates an instance of POINTCLASS, changes the coordinates of the instantiated point, and returns the y-value.

```
POINTCLASS new set:[x=1,y=2] gety

→,+ < POINT / [self=POINT,class=POINTPUBLIC] > set:[x=1,y=2] gety

→,+ < POINT / [self=POINT,class=POINTPUBLIC,x=1,y=2] > gety

→,+ 2
```

The second reduction illustrates the *late binding of self*, one of the essential aspects of inheritance, by creating a BOUNDEDPOINT-instance, and executing its move-method. In the example we can clearly see that this results in invoking the move-method of POINT using the self of BOUNDEDPOINT. In this way the new set-method is used instead of the original one. Hence we obtain late binding.

```
BOUNDEDCLASS new move:[x=1,y=2]

→,+ {[inst=BOUNDEDPOINT,self=BOUNDEDPUBLIC,super=POINTPUBLIC]}
      (<super/[inst=inst,self=self,super=super]> new) move:[x=1,y=2]

= < POINTPUBLIC /
      [inst=BOUNDEDPOINT,self=BOUNDEDPUBLIC,super=POINTPUBLIC]>
      new move:[x=1,y=2]

→,+ < BOUNDEDPOINT / [self=BOUNDEDPOINT,class=BOUNDEDPUBLIC] >
      move:[x=1,y=2]

→,+ < BOUNDEDPOINT / [self=BOUNDEDPOINT,class=BOUNDEDPUBLIC] >
      set:[x=1,y=2]

→,+ < 1 ≤ 5 / [then=<BOUNDEDPOINT/[self=BOUNDEDPOINT,
                                    class=BOUNDEDPUBLIC,x=1,y=2]>,
                else=error ] > if

→,+ < BOUNDEDPOINT /
       [self=BOUNDEDPOINT,class=BOUNDEDPUBLIC,x=1,y=2] >
```

We must admit that there are still some problems with class-based inheritance due to the fact that our argument passing mechanism compromises encapsulation. However, due to space limitations we will not go deeper into this matter.

4.2 Mixin-based Inheritance

Mixin-based inheritance is a mechanism sometimes used in prototype-based object-oriented languages (in contrast to languages such as Smalltalk, where class-based inheritance is advocated). New objects are created by cloning existing ones. Moreover, objects can be extended using mixins: a mixin-method returns a copy of the receiver extended with the declarations specified in the mixin's body. The order in which mixins are applied is important for the external visibility of public attribute names. In [16], it is shown how mixins can be regarded as named attributes of an object: an object lists as mixin attributes all and only those mixins that are applicable

to it. Precisely those mixin-attributes can be used to extend the object. Mixins can be inherited and nested within each other. In this way we obtain a very powerful inheritance mechanism. In fact, in [2] it is shown that mixin-based inheritance subsumes the inheritance mechanisms provided by Smalltalk, Beta and CLOS.

In this paper, we follow the same approach as [16]. Using the OPUS-syntax, a mixin-method will always have the following form

```
mixin#σ( self + MIXIN )
```

because a mixin extends the current object (self) with new methods and variables described in MIXIN. Remark that the same syntax will also be used for messages that only overwrite variables in an object with new values. Indeed, because overwriting existing attributes is a special case of extending an object with attributes, these methods can be regarded as a special kind of mixin-methods. We will introduce the following shorthand notation for the expression above:

```
mixin@MIXIN
```

As an example, assume that we have three mixin-methods makePoint, addColor and make3D. The first two mixin-methods can be applied to every object, while the third one is nested: it can only be applied to objects that have already been extended with makePoint. Invoking makePoint extends the current object with a new mixin-method make3D, two instance variables x and y, and two methods setx and sety for updating their values. Executing make3D extends this new (already extended) object by adding an instance variable z and a public method setz. Finally, the addColor method can be applied to every object and results in extending the object with an instance variable c and a public setcol method. Using the new notation mentioned above, this example can be expressed as follows:

```
OBJECT := [  makePoint@[  x=x,
                          y=y,
                          setx@[ x=x ],
                          sety@[ y=y ],
                          make3D@[   z=z,
                                     setz@[ z=z ] ] ],
             addColor@[    c=c,
                          setcol@[ c=c ] ] ]
```

Using these definitions, we can for example extend the root-object to a point (1,2), replace its y-coordinate by 3, extend this point-object to a colored point-object by adding the color yellow, extend this colored point-object to a colored 3D-object with z-coordinate 2, and finally ask the color of this object using one single expression:

```
OBJECT makePoint:[x=1,y=2] sety:[y=3] addColor:[c=yellow] make3D:[z=2] c
```

Note that it is also possible to obtain late binding of self using mixin-methods, but since the mechanism is essentially the same as for class-based inheritance, we will not provide an example of this.

5 Related Work

There are lots of calculi being constructed for the purpose of modeling object-orientedness in a suitable way. Each of these calculi has its own specific characteristics and advantages. For this reason, it is useful to find out where our calculus can be situated amongst the others.

Only a small set of people is currently working on OO calculi that are not based on λ-calculus, mostly in the area of concurrent OOPLs. For example, Robin Milner has introduced the π-calculus [11], and Oscar Nierstrasz proposed an "object-calculus" [13]. Most of the current research on formal OO-models consists of attempts to generalize λ-calculus in order to describe OO-concepts more easily. All of these models however have difficulties in expressing some of the essential features of object-orientedness in a satisfying way.

For example, as opposed to OPUS, most of the OO-models based on polymorphic typed λ-calculus (e.g. [10, 14, 15]) do not have atomic message passing since it is composed of record selection and functional application. We explained earlier that this compromises object-based encapsulation.

Dami's λ-calculus with names, combinations and alternations (cf. [7, 8]) essentially has the same shortcomings, although it seems possible to introduce a higher-level syntax in which these problems do not occur. There appear to be some striking similarities with OPUS, such as the use of names for interaction between objects and the way in which objects are modeled.

Another approach based on λ-calculus is λ&-calculus [3]. This calculus does have atomic message passing, and the message passing rules are very similar to ours. However, because λ& adopts a multi-methods approach, object-interfaces are not explicit at all, while OPUS does have explicit interfaces.

There also seem to be many connections between OPUS and the untyped calculus proposed by Abadi and Cardelli in [1], but the latter contains an explicit recursion operator in its basic syntax. We feel that this is not necessary, because it is possible to simulate recursion using more primitive concepts, as we have shown earlier in this paper.

6 Future Perspectives

This section will motivate how our calculus can be used as a tool in the research and understanding of new OO-concepts, the interrelationships between existing concepts, etc.

Using a calculus one can give rigorous definitions of concepts that have no generally accepted formal definition (e.g. what is an object, what is a class?). Furthermore, there is still a lot of discussion going on about which features are truly essential to OO and which are not. A formal model might provide help in solving these problems. For example, in this paper we have shown that recursion is not essential, because it can be expressed using more primitive concepts.

A common theory also provides a basis for making comparisons between related concepts, enabling us to learn more about their interrelationships. This can be useful to find out which approach is best fit to solve a certain problem. For example, in the near future we are planning to use our calculus for investigating the differences

between encapsulated and non-encapsulated inheritance (cf. [17]), and to find out how they interact with each other.

Our OO calculus can be interesting from a more *practical point of view* as well. It might be used as a basis for developing new OOPLs based on a very small set of essential features (thus with a more orthogonal design), yet with the same expressiveness as currently available OOPLs. (In analogy with the design of functional languages based on λ-calculus like LISP, Scheme, Gofer, Miranda and Haskell.)

Finally, a lot of people face important difficulties when trying to *prove* general properties of OOPLs. One of the main reasons for their problems is a lack of a solid formal basis. Thus formal methods for OO are necessary for reasoning about the paradigm.

Although our calculus provides some very promising results, further research remains to be done. Below we provide some ideas for improvement of our calculus.

• We plan to show that OPUS is as powerful as λ-calculus, by providing a translation scheme from λ-calculus to OPUS and vice versa.

• We are studying the confluency property for our calculus. We are convinced that our calculus is confluent, but the results are not mature enough to be presented in this paper.

• The derivation rules of OPUS can be considered as a kind of *operational semantics* for the syntactic constructs of the calculus. Similarly, it is worthwhile looking at a suitable denotational semantics for OPUS.

• Just as λ-calculus can be extended to typed λ-calculus, typing could be added to OPUS as well.

7 Conclusion

As a conclusion, we claim that we have succeeded in finding a suitable formal framework in which many object-oriented programming languages can be expressed. Although the calculus in this paper has a very simple syntax and only a few reduction rules, we believe that it can model all key-features of object-orientedness.

Our calculus contains some innovative features such as the use of an explicit encapsulation operator to hide implementation details of objects, and the fact that message passing is treated as an atomic operation. We argued that this approach is more object-oriented than in many other formal models where message passing consists of two distinct operations, namely method selection and method invocation.

Another important feature is that interaction occurs through names (as in Dami's calculus [7]) instead of positions as is the case in λ-calculus. We also introduced an incremental modification operator to model different kinds of inheritance mechanisms, and have shown that it is not necessary to include notions like recursion and classes in the basic syntax, because they can be modeled using the other syntactic constructs. Although the calculus was originally designed to work with object-oriented examples, functional examples can also be expressed easily, making the calculus more or less multi-purpose. Finally, we have argued how our calculus can be used as a tool for describing and relating existing OO-concepts, or investigating new concepts.

8 Acknowledgments

We express our extreme gratitude to Niels Boyen and Wolfgang de Meuter for the heavy discussions and numerous remarks, to Laurent Dami and Kris De Volder for some helpful comments given when proof-reading an early version of this paper, and to several anonymous referees for their useful suggestions.

9 References

1. M. Abadi, L. Cardelli: A Theory of Primitive Objects. Unpublished, 1994
2. G. Bracha, W. Cook: Mixin-based Inheritance. OOPSLA/ECOOP '90 Conference Proceedings, pp. 303-311, ACM Press, 1990
3. G. Castagna, G. Ghelli, G. Longo: A Calculus for Overloaded Functions with Subtyping. Extended Abstract, ACM, 1992
4. L. Cardelli, J. Mitchell: Operations on Records. Proceedings on Mathematical Foundations of Programming Semantics, LNCS 442, 1989
5. L. Cardelli: A semantics of multiple inheritance. Information and Computation 76, pp. 138-164, 1988
6. P. Canning, W. Cook, W. Hill, W. Olthoff: Interfaces for Strongly-Typed Object-Oriented Programming. OOPSLA '89 Conference Proceedings, pp. 457-467, ACM Press, 1989
7. L. Dami: Extensible Lambda Expressions: A Lambda Calculus with Names, Combinations and Alternations. Technical Report, University of Geneva, 1993
8. L. Dami: Named Parameters: A Foundation for Subtyping. Extended Abstract, Submitted to LICS, University of Geneva, 1994
9. L. Dami: Software Composition: Towards an Integration of Functional and Object-Oriented Approaches. Ph.D.-Thesis, University of Geneva, 1994
10. M. Hofmann, B. Pierce: An abstract view of objects and subtyping. Technical Report ECS-LFCS-92-226, University of Edinburgh, 1992
11. R. Milner: The Poliadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991
12. O. Nierstrasz: A survey of object-oriented concepts. Object-oriented concepts, databases and applications, pp. 3-21, ACM Press and Addison-Wesley, 1989
13. O. Nierstrasz: Towards an Object Calculus. ECOOP Workshop on Object-Based Concurrent Computing, LNCS 612, 1992
14. B. Pierce: A Model of Delegation Based on Existential Types. Working Draft, Inria-Roquencourt, 1993
15. B. Pierce, D. Turner: Object-oriented Programming without Recursive Types. Technical Report ECS-LFCS-92-225, University of Edinburgh, 1992
16. P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limbergen: Nested Mixin-Methods in Agora. Technical Report vub-prog-tr-93-01, Vrije Universiteit Brussel, ECOOP' 93 Conference Proceedings, 1993
17. A. Snyder: Inheritance and the Development of Encapsulated Software Components. Research Directions in Object-Oriented Programming, MIT Press, 1987
18. P. Steyaert: Towards a Calculus for Objects and its Reflective Variant. Extended Abstract (unpublished), presented at ECOOP '92 workshop on reflection and metalevel architectures, 1992

19. P. Steyaert: Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks. Ph.D.-Thesis, Vrije Universiteit Brussel, 1994
20. P. Wegner, 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