Vrije Universiteit Brussel
Faculteit Wetenschappen

Self sends as primary construct: a criterion for object-oriented language design

Tom Mens Marc Van Limberghen

Techreport vub-prog-tr-95-08

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Title:

Self sends as primary construct: a criterion for object-oriented language design

Authors:

Tom Mens
Department of Mathematics (DWIS)
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: tommens@vub.ac.be
Phone: +32.2.6293474
Fax: +32.2.6293495

Marc Van Limberghen
Programming Lab (DINF)
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: mvlimber@vub.ac.be
Phone: +32.2.6293492
Fax: +32.2.6293495

Keywords:

language design, OO software engineering, formal model, denotational semantics

Self sends as primary construct: a criterion for object-oriented language design

Tom Mens
Department of Mathematics (DWIS)
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
tommens@vub.ac.be

Marc Van Limberghen
Programming Lab (DINF)
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
mvlimber@vub.ac.be

ABSTRACT:

Most object-oriented languages provide a self pseudo variable that serves two purposes: invoking methods through self sends and returning a self reference. Current formal models capture this behaviour by introducing a self parameter that fulfils both roles. We will show that this double functionality is incompatible with object-based encapsulation. Moreover we will point out that the separation of both functionalities is needed in type systems. We propose a formal model based on a primitive self send construct, and present its denotational semantics. A self reference is implemented through a self send instead of the other way around. This model is extended formally with an object-based encapsulation mechanism. It allows us to keep inheritance, encapsulation and typing orthogonal. This way we obtain a flexible and incremental software development scheme.

KEYWORDS:
language design, OO software engineering, formal model, denotational semantics

1. Introduction

In object-oriented programs self sends are not merely a kind of recursive behaviour since they contain important design information. [Dodani&Tsai92], [Johnson&Foote88] stressed the design importance of abstract classes and observed that self sends of abstract methods are hereby essential. In a similar vein [Lamping93] indicated the importance of self sends for users who specialise a class and introduced therefore the notion of a specialisation interface, containing all the self sends performed by a class.

Most object-oriented languages offer a kind of pseudo variable to return a self reference. In Smalltalk [Goldberg&Robson83] this pseudo variable is called `self`. In C++ [Ellis&Stroustrup91] it is called `this`, in Eiffel [Meyer88] it's name is `Current`. Many formal models follow a similar approach, e.g. [Cook89], [Cook&Palsberg89] gave the denotational semantics of inheritance through generators of the form λ`self.Object`. The `self` parameter constitutes the formal equivalent of the pseudo variable. [Abadi&Cardelli94] defined a calculus in which each method contains a special `self` parameter denoting the object the method belongs to.[1]

All these systems implement self sends by sending a message to the `self` reference.[2] Unfortunately, these kinds of message sends cannot be strictly distinguished from other message sends. Conditional expressions for instance can prevent us from determining statically whether a self reference is returned or not. Hence not all self sends are recognisable. As a consequence the underlying design information is lost, and software engineering tools that have to do with self sends cannot be realised. These tools include very fundamental ones like typing and encapsulation, indispensable for practical programming languages.

- The Smalltalk language did not succeed in effectively hiding the attributes of the so-called "private protocol". In C++ the encapsulation mechanism is based on the (presence of a) type system. As a consequence the private members of a C++ object are not hidden from another object of the same type. This way objects are not free in the implementation of their behaviour, as opposed to pure

[1] The latter approach slightly differs from the first, in that each method has its own `self` parameter. This difference is however irrelevant to the current discussion.

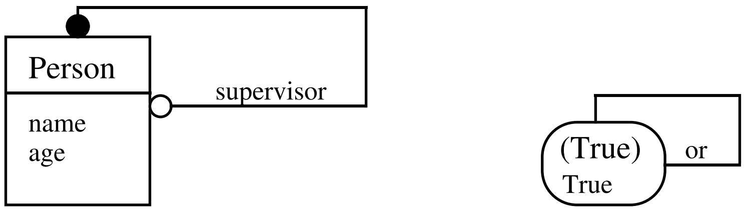
[2] In C++ and Eiffel, the pseudo variable name may be omitted when performing self sends, but this is only a syntactic convention.

object-based encapsulation [Steyaert94]. Eiffel provides object-based encapsulation by giving a context sensitive semantics to the `Current` pseudo variable, thus violating the compositionality criterion for programming languages [Tennent91].

- Type systems for object-oriented languages have to deal with the covariance-contravariance dilemma [Dodani&Tsai92]. Self sends are often used in a context where the self type of the subclass is not type safely substitutable for that of the superclass. To solve this problem state of the art type systems need the recognition of self sends.

The mixture of typing and inheritance has been criticised in [Canning&al.89], the mixture of encapsulation and inheritance in [Bracha92] and [Szyperski92]. We will show that, in order to keep inheritance, typing and encapsulation as orthogonal language concepts, self sends can *not* be expressed using a `self` reference. Therefore self sends should be made explicit.

Knowing that a `self` pseudo variable is no longer used to perform self sends, the question naturally arises if it is still useful. Besides performing self sends, the pseudo variable is used to return self references that are automatically adapted via late binding. We will argue later how self references can be modelled using explicit self sends. A self reference can be regarded as a data cycle of length one. Do not confuse "recursive data structures" with "data cycles" (see figure 1). Explicit cycles of length one - for example a True object that returns itself on invocation of the or message - occur less frequently than self sends.[3] It is debatable whether such specific cycles should be explicitly present as only data modelling concept in an object-oriented programming language.



A recursive data structure.

A data cycle.

Figure 1: OMT [Rumbaugh&al.91] diagrams.

The section that follows describes a formal model containing explicit self sends but no `self` pseudo variable, and presents its denotational semantics. Section 3 shows that expressions in our model conform to the Factory Method design pattern [Pree94] in a natural way. Consequently, all kind of referencing structures can be made adaptable to inheritance. As an example of adding software engineering concepts that need explicit self sends, section 4 orthogonally extends our model with an encapsulation mechanism. Section 5 informally discusses why state of the art type systems need an explicit self send construct. Section 6 illustrates how encapsulation and typing can be orthogonally combined to yield an incremental and powerful software engineering mechanism.

2. The formal model

Our model is stateless: side effects are absent. The expressions in our model can be interpreted as classes (actually mixin-classes as in [Bracha92], [Bracha&Lindstrom92]) or as (specialisable) objects. As such, the statements we make are applicable to the class-based as well as to the object-based paradigm. We start by giving the syntax and some examples. The denotational semantics is presented in subsection 2.3.

2.1. Syntax

The most simple kind of object that can be expressed in our model is an object containing only one method, that can make use of arguments, e.g. "inc(#a) = #a+1". In this expression #a is a formal argument name that is bound to the actual argument on invocation of the method inc. In case of name conflicts between formal argument names, the innermost argument name has precedence.

More complicated objects can be obtained through right preferential inheritance, by combining two objects parent and modifier into a composed object parent & modifier.

---

3 In Smalltalk imperative methods implicitly return self. But these cycles of length one are only intended to shorten a sequence of imperative method invocations on the same receiver.

Our model contains a primitive self send construct. Therefore will make a distinction between two kinds of message sends: object sends and self sends.

- Object sends are denoted by writing the receiver between parentheses followed by a message-selector that we will call *label*. Arguments can be passed by putting them behind the label between parentheses. E.g. "(p)move(1)" means sending the message move to p with argument 1.
- Receiverless messages represent self sends, e.g. "inc(2)" means a self send of label inc with argument 2.

To be able to perform code reuse, we will also allow super calls that are invoked through the reserved keyword `super`. We use super calls only to refer to the method we are currently overriding. In presence of an encapsulation mechanism this restriction does not reduce inheritance expressivity, as discussed in [Van Limberghen&Mens95]. Furthermore it provides the ability to normalise class components [Van Limberghen95]. In this approach it is redundant to specify which super method is activated. For example when overriding a move method, we will no longer write "(`super`)move(1)" to call the super variant of the move method, but simply "`super`(1)".

The following context free grammar rules with Expression as start symbol describe the syntax in EBNF notation.

Expression	→	Method | Compose | SelfSend | ObjectSend | SuperCall | ArgRef
Method	→	Label "(" ArgRef ")" "=" Expression
Compose	→	"(" Expression "&" Expression ")"
SelfSend	→	Label Argument
ObjectSend	→	"(" Expression ")" Label Argument
SuperCall	→	"`super`" Argument
Argument	→	"(" Expression ")" | "(" ε ")"
ArgRef	→	"#" Label

ε corresponds to the empty expression. A Label is a finite sequence of letters. `super` is a reserved keyword.

<u>Conventions</u>:

- From now on, let E_i denote meta-variables of type Expression, and L_i meta-variables of type Label. The indices can also be omitted.
- To simplify the examples we adopt the convention to omit the parentheses in absence of an argument. In other words, `super`, L and (E) L are shortcuts for `super`(), L() and (E)L() respectively. Similarly we will write L_1=E instead of L_1(#L_2)=E for methods that don't make use of arguments.
- We also adopt the convention to omit parentheses whenever appropriate. The denotational semantics will be defined in such a way that composition is associative, so we may write E_1&E_2&E_3 instead of ((E_1&E_2)&E_3) or (E_1&(E_2&E_3)). We also agree that "=" binds stronger than "&", so that L_1=E_1&L_2=E_2 corresponds to (L_1=E_1)&(L_2=E_2) instead of L_1=(E_1&L_2=E_2).

2.2. *Examples*

To make the examples in this paper more readable, we provide a kind of macro expansion mechanism. Words written in uppercase refer to predefined expressions (through the := construct) that have to be replaced "in place". Sometimes we add line comments behind the % symbol.

As a first example, consider a POINT with an x and y-attribute, and an angle method that calculates the angle of the point with the x-axis.[4]

```
POINT :=      x=1
          &   y=1
          &   angle=  ((y)/(x))  arctan          % a self send of y and a self send of x
```

If we want to reflect this POINT about the y-axis, we simply need to extend it by overriding the x-attribute with its additive inverse through a super call. Calculating the angle of this new MIRROREDPOINT will yield a different result.

---

[4] For reasons of brevity we have not included numerals and arithmetic operations as basic primitives in our calculus, since they can be expressed using the other syntactic constructs in a way similar to [Menset.al. 95].

```
MIRRORX :=         x= -super                       % a super call to x, the label is omitted

MIRROREDPOINT :=      POINT
                   &  MIRRORX                     % POINT is the parent and MIRRORX the modifier
```

When dealing with arguments, things get more complicated. As a first example, we could have defined MIRRORX by means of an implementation method f that mirrors the x argument:

```
MIRRORX2 :=        f(#a) = -#a                     % #a is the formal argument of f.
                 & x= f(super)
```

In the second, more complicated example we define a LIBRARY consisting of a circle and a point. The circle understands the messages centre and radius, where centre refers to the point object of the library. Since the method-body of centre can only perform self sends to attributes of circle (in casu centre and radius), point needs to be passed to circle as argument of the initialisation method init.

```
LIBRARY :=     circle=    (init(#p)=(      centre= #p     % #p denotes the actual argument point
                                       &   radius= 3 )
                          ) init(point)                   % a self send of point
             & point=     POINT
```

To fetch the x-value of the centre of the circle, one simply needs to perform the following sequence of message sends:

```
( ( (LIBRARY) circle) centre) x
```

2.3. Denotational semantics

In this subsection we will present the denotational semantics of our model. Just like in [Cook89], [Cook&Palsberg89] we make use of *generators* (i.e. a function expecting a `self` parameter and returning an environment). There are however three important differences.

- The first difference is that generators are the only first class entities in our model. Generators can be seen as classes as well as specialisable objects. This way our ideas apply to the class-based as well as the object-based paradigm. A second consequence is that every client can be inheritor as well as user client.
- The second, most important difference is that the `self` parameter cannot be returned and is only used to perform self sends. As a result, the `self` parameter need not be first class, and hence can be modelled with an environment instead of a generator.
- The last difference is that super calls are defined at method level instead of object level.

Semantic domains:

ArgEnv	=	**Label**	$\rightarrow$	**Generator**
MethodEnv	=	**Label**	$\rightarrow$	**MethodBody**
Generator	=	**MethodEnv**	$\rightarrow$	**MethodEnv**
MethodBody	=	**ArgGen**	$\rightarrow$	**ArgGen**
ArgGen	=	**Generator**	$\rightarrow$	**Generator**

A **Generator** expects a self environment and returns an environment of method-bodies. An **ArgGen** is a **Generator** expecting another **Generator** as argument. The fact that super calls are defined at method level is reflected in the semantic domains: a **MethodBody** expects a super-version of type **ArgGen** (since a super call can have an argument), and returns an **ArgGen** in which the method argument is filled in on invocation of the method.

Before defining the semantic function, we introduce two operations on environments. $e_1 +_r e_2$ corresponds to the right preferential concatenation of environments, and $\{key \rightarrow val\}$ creates a new environment containing only one association.

$$e_1 +_r e_2 \quad = \quad \lambda x.\ \textit{if}\ (e_2\ x) = \perp\ \textit{then}\ (e_1\ x)\ \textit{else}\ (e_2\ x)$$

$$\{key \rightarrow val\} \quad = \quad \lambda x.\ \textit{if}\ \ x = key\ \textit{then}\ val\ \textit{else}\ \perp$$

Semantic function:

$$\mathcal{E}[\![\ ]\!] : \text{Expression} \rightarrow \textbf{Generator}$$

$$\mathcal{E}[\![E]\!] = [\![E]\!]\ \perp\ \perp\ \perp$$

The semantic function $\mathcal{E}[\![\ ]\!]$ tells us how an expression can be denoted. This function makes use of an auxiliary function $[\![\ ]\!]$ that expects a syntactic expression, a self environment, an environment of local arguments, and a super argument. It returns a generator. $[\![\ ]\!]$ is inductively defined based on the structure of the expression.

$$[\![\ ]\!] : \text{Expression} \rightarrow \textbf{MethodEnv} \rightarrow \textbf{ArgEnv} \rightarrow \textbf{ArgGen} \rightarrow \textbf{Generator}$$

$$\begin{array}{ll}
[\![(E_1)L(E_2)]\!] = & \lambda self.\lambda args.\lambda super.\ \mathit{send}\ \mathbf{Y}([\![E_1]\!]\ self\ args\ super)\ L\ ([\![E_2]\!]\ self\ args\ super) \\
[\![L_1(\#L_2)=E]\!] = & \lambda self.\lambda args.\lambda super.\ \lambda self2.\{ L_1 \rightarrow \lambda super2.\lambda arg.[\![E]\!]\ self2\ (args +_r \{L_2 \rightarrow arg\})\ super2\ \} \\
[\![E_1\ \&\ E_2]\!] = & \lambda self.\lambda args.\lambda super.\ ([\![E_1]\!]\ self\ args\ super)\ 7\ ([\![E_2]\!]\ self\ args\ super) \\
[\![\#L]\!] = & \lambda self.\lambda args.\lambda super.\ args\ L \\
[\![L(E_2)]\!] = & \lambda self.\lambda args.\lambda super.\ \mathit{send}\ self\ L\ ([\![E_2]\!]\ self\ args\ super) \\
[\![super(E_2)]\!] = & \lambda self.\lambda args.\lambda super.\ super\ ([\![E_2]\!]\ self\ args\ super) \\
[\![\varepsilon]\!] = & \lambda self.\lambda args.\lambda super.\ \perp
\end{array}$$

Sending a message to a generator is performed by taking the fixed point **Y** of the receiver (to recursively bind all self sends in this generator), selecting the label corresponding to the message in the resulting environment, and binding the actual argument to the formal argument name in the corresponding method-body. Performing a self send on the other hand is a lot easier, since we only have to select the method corresponding to the message-label in the self environment, and replace all argument references by the actual argument. The formal definition of *send* used in both cases is given below:

$$\mathit{send} : \textbf{MethodEnv} \rightarrow \textbf{Label} \rightarrow \textbf{Generator} \rightarrow \textbf{Generator}$$

$$\mathit{send}\ receiver\ L\ arg = receiver\ L\ \perp\ arg$$

Combining two expressions E_1 and E_2 via the incremental modification mechanism & is performed by using the auxiliary semantic function 7. Intuitively, parent 7 modifier combines the two generators parent and modifier into a larger generator, while binding super calls in modifier to the associated method-bodies in parent.

$$7 : \textbf{Generator} \rightarrow \textbf{Generator} \rightarrow \textbf{Generator}$$

$$g_1\ 7\ g_2 = \lambda self.\ [\ (g_1\ self) +_r \lambda label.\lambda super.\ g_2\ self\ label\ ((\{label \rightarrow \lambda super.super\} +_r g_1\ self)\ label\ super)\ ]$$

Note that in the syntax and semantics, only one argument at a time can be passed with message sends and super calls. Multiple arguments can be added in a straightforward way, but this would only clutter the formal model.

3. References are subject to late binding.

3.1. Cyclic references

In current object-oriented languages, data cycle of length one can easily be obtained without using a pseudo variable `self`, namely by providing an initialisation method. After having created an object, one has to call the initialisation method with itself as argument. As a matter of fact this is in most languages the *only* way in current object-oriented languages to deal with more general cyclic data, like for instance two mutually referring objects.

Consider for example a circular list (cycle) of length one. cycle contains an info attribute, and a next attribute that returns the cycle itself (through the formal argument #c that is bound to cycle).

```
CYCLE :=      cycle=   (init(#c)=(     next=#c
                                    &  info=1)
                        )  init(cycle)
```

Repeatedly sending next to this cycle indeed returns itself:

```
⟦ ((((CYCLE) cycle) next) next) info ⟧   =   1
```

3.2. Specialisation of references

Reconsider the LIBRARY example of the previous section. Suppose we want to extend this library so that points and circles now understand a distance message, returning the distance to the origin. The distance of a circle is defined as the distance of its centre to the origin. To create this extended library, we employ super calls to reuse the circle and point code of the previous library.

EXTLIB := LIBRARY
& point= (super % a super call of point, the label is omitted
& distance= $\sqrt{\mathbf{x}^2 + \mathbf{y}^2}$) % a self send of x and a self send of y
& circle= (super % a super call of circle, the label is omitted
& distance= (centre) distance) % a self send of centre

The distance of the circle to the origin can be calculated as follows:

⟦ ((EXTLIB) circle) distance ⟧ = $\sqrt{2}$

Class libraries that internally create objects cannot always easily be specialised. The reason is that class references are usually fixed so the internally referred classes cannot be updated with late binding. In our model this problem disappeared since all references are subject to late binding, including references to the most global name space. This is for example not the case in Smalltalk where references to global variables (including class-references) are not subject to late binding. Our approach accords to the Factory Method design pattern (also called flexible object creation in [Pree94]), and to the static part of the "Make Isn't Generic" problem [Kiczales93]. The self send of point performed by circle (see LIBRARY in previous section) can indeed be considered as making a new, specialised point! An object-based interpretation is that circle is an object referring to the specialised point object.

In general the owner, circle, and the part, point, are different. Let's have a closer look at the case where owner and part are equal, as in the CYCLE example. We can extend cycle with an extra info field labelled extraInfo.

```
EXTCYCLE :=   CYCLE
              & cycle=  (   super
                         &  extraInfo= 2 )
```

```
⟦ ((((EXTCYCLE ) cycle) next) next) extraInfo ⟧   =   2
```

The expression above, again with an arbitrary number of consecutive next messages, yields 2. Late binded self references turn out to be only a particular case of a more general requirement for object-oriented languages.

Note that an object gets its self reference from the outside. This way we lose the explicitness of self references. Indeed, cycle could have been provided with another actual argument for init. We prefer the explicitness of self sends to the explicitness of self references. Moreover the initialisation method can be made transparent in most languages. In Smalltalk for instance the initialisation method can be called from within the class method new. In object-based languages the propagation of the *copy* operator [Shah&al.89] should appropriately take care of cyclic data. Adding a `self` pseudo variable only to deal with explicit cycles of length one seems like a poor addition to our model.

4. Orthogonally extending the model with encapsulation

4.1. Encapsulation should be orthogonal to inheritance

In most object-oriented languages, encapsulation of methods is only considered with regard to user clients. However encapsulation towards inheritors is equally important: it makes the development of a subclass more independent of the implementation of the superclass. This has been indicated in the literature. [Szyperski92] for example states that modules and classes are totally different concepts. They should both be supported as orthogonal features instead of being unified into one and the same construct. A similar idea can be found in [Bracha92]: "*The main problem with encapsulation concerns allowing only a restricted set of interfaces to clients (mostly only a user client interface and an inheriting client interface).*"

In C++ [Ellis&Stroustrup91] this principle is not satisfied, since there are only two different interfaces: the user client interface consists of public attributes only, while the inheriting client interface also incorporates the so-called protected attributes[5]. Eiffel [Meyer88] suffers from the same problem. The export mechanism does not restrict visibility towards inheritors.

4.2. Encapsulation in presence of self references

In this subsection we make clear that performing self sends and returning a self reference with one and the same `self` construct is incompatible with object-based encapsulation!

The "*hide* " operator of [Bracha92] and [Bracha&Lindstrom92] provides a simple and elegant encapsulation mechanism. It can be used to incrementally hide parts of a class and it remains orthogonal to inheritance. Unfortunately it was added to the inheritance model of [Cook89] where the `self` parameter also serves to return a self reference. Within the visibility boundaries that hide some attribute from the outside world, the `self` parameter must be a non-encapsulated version of the object. Otherwise we cannot even perform internal self sends to the hidden attribute, so the attribute would be completely useless. Returning a self reference from within the visibility boundaries reveals the problem: a version of the object is returned in which the private attribute is not hidden. This unexpected result is in contrast with the encapsulation spirit of all existing languages.

Consider for example the following Eiffel code. Object-based encapsulation is obtained, but at the cost of a context sensitive semantics for `Current`. As opposed to Bracha's solution, returning `Current` (line 4 and 10) renders an encapsulated version of the self reference. Directly using `Current` as receiver (line 9) renders a non-encapsulated version.

```
(1)  class THISCLASS
(2)  creation create
(3)     feature {NONE} my_hidden : INTEGER is 3;
(4)     feature me : THISCLASS is do Result:=Current end;
(5)  create is
(6)     local i : INTEGER,
(7)     another : THISCLASS;
(8)  do
(9)     i:=Current.my_hidden;        %valid
(10)    i:=me.my_hidden;             %invalid as opposed to (9)
(11)    i:=another.my_hidden;        %invalid: Eiffel supports object-based encapsulation
(12) end
(13) end;
```

4.3. Extending the model with encapsulation

To extend our model with encapsulation, we first extend the syntax:

Expression	→	Encaps
Encaps	→	"[" Expression "]" "(" Label ")"

Secondly we give the denotational semantics of this new expression:

[5] The C++ private attributes are never subject to late binding: they are ordinary procedures rather than methods.

⟦ [E](L) ⟧ =	λself.λargs.λsuper. *encaps* (⟦ E ⟧ self args super) L
where *encaps* is defined as follows:	
encaps : **Generator** → **Label** →	**Generator**
encaps g L =	λself. [**Y**(λfix.λself2. g (self2 $+_r$ { L → λsuper. fix self2 L ⊥ })) self \ L]
where	e \ L = λx. *if* x = L *then* ⊥ *else* (e x)

encaps accords to the "hide" operator of [Bracha&Lindstrom92] (except for our super treatment). To encapsulate a label L in a generator g, first all self sends to L occurring in g are recursively replaced by the actual body of L. To this extent the receiver g gets a self in which the attribute L is not hidden (note that in our model, g cannot return this non-encapsulated value of self). Next the label L is removed from the environment (via \ ) in order to make it invisible to further users. Note that an attribute that is to be encapsulated should not perform any super sends. All super sends still occurring are simply replaced by ⊥.

The introduced encapsulation operator is orthogonal to the already existing composition operator, since only additions had to be made to the existing semantics.

4.4. Use of encapsulation

There are three main reasons for a programmer to use encapsulation:
- To hide implementation details that are not of interest to the user (ordinary client or inheritor);
- To deal with unintended name collisions occurring when we compose two objects that accidentally have used methods with the same name;
- To gain restricted views onto one and the same object.

[Van Limberghen&Mens95] states that these three functionalities are in fact one and the same and that it provides an adequate way to solve multiple inheritance problems. Below we just give an example of each.

As an example of the first usage, reconsider the POINT and MIRRORX2 example of section 2. Since f is only an implementation method, it should not be visible in the interface of the point after having mirrored the x-attribute. Hence f should be encapsulated:

POINT & [MIRRORX2](f)

A second reason for encapsulating f is to deal with unintended name collisions occurring when different subclasses are implemented independently. Suppose we also need a function to double the x attribute of the point:

DOUBLEX := f(#a)=(#a+#a) & x=f(super)

If we now consecutively want to double and mirror the x-attribute (or vice versa) we get into trouble, since both operations are defined in terms of an implementation method f, so one of those two methods will unintentionally become overridden by the other one. To solve this, both implementation methods should be encapsulated before composing with POINT:

POINT & [MIRRORX2](f) & [DOUBLEX](f)

A final purpose of encapsulation is to gain a more restricted view on classes or objects. For example, a one-dimensional point obtained by projecting a two-dimensional one on the x-axis can be achieved by simply encapsulating the y and angle method (in an arbitrary order):

PROJECTPOINT := [[POINT](angle)](y)

5. Typing

By means of an example we first briefly recall the covariance-contravariance controversy [Dodani&Tsai92]. Then we enumerate some type systems that coped with this problem and observe that they can only be realised on languages that contain self sends as primary construct. Using the same example, the following section will deal with another obstacle in the covariance-contravariance controversy, namely binary methods [Bruce&al.95].

5.1 The covariance-contravariance controversy

We will only informally illustrate the main idea and appeal to the intuition of the reader to interpret the following example. Please neglect the print and the twice methods for the first part of our argumentation.

```
Class ListElement
        withInstanceType  ListElementType
        withAttributes
            info: String
            next: ListElementType
            equalContents(#e: ListElementType): Boolean
                    = { info == (#e)info}
            identity: String
                    = { ("I contain ") concat(info) }
            print = { (me identity) print }
            twice: Boolean
                    = { me  equalContents(next) }

Class ExtListElement
        withInstanceType  ExtListElementType
        inherits from ListElement
        withAttributes
            extraInfo: String
            next: ExtListElementType
            equalContents(#e: ExtListElementType): Boolean
                    = { (extraInfo == (#e)extraInfo) and (super(#e)) }
            identity: String
                    = { ((super) concat("and ")) concat(extraInfo) }
```

ExtListElement violates the contravariance rule because the overridden version of the equalContents method expects an argument with a bigger interface [Canning&al.89]. So, ExtListElement cannot be a subtype of ListElement. Nevertheless, an object-oriented programmer would still like to reuse ListElement in his construction for ExtListElement. ExtListElement can be a subclass of ListElement without being a subtype. According to [Canning&al.89], subclassing and subtyping should indeed be separated.

But even in the case where subclasses do not have to be in subtype relationship with their parent, a problem arises when the print method-body is type-checked. Suppose that me is a message send returning a self reference or an instance variable assigned to a self reference. By using me as receiver, the print method performs an unrecognisable self send of identity. A naïve type system will not allow the expression me to return an instance of ExtListElement since ExtListElement is not in subtype relationship with ListElement. In this case, the presence of the print method will even forbid to build the subclass ExtListElement. Nevertheless such extensions occur frequently and should be allowed by the type-checker.

5.2 State of the art type systems need recognisable self sends

Recent type systems coped with the above problem. But, since they were built to type existing languages (based on a `self` pseudo variable), they had to artificially add some self send recognition clause to their model. In fact these type systems are only applicable to languages that comply with our model.

[Abadi&Cardelli94] tackled the problem by introducing a self-type. The `self` parameter is then declared to have this self-type. But they recognised that this mechanism only works for complete classes or objects. Incremental typing seemed to be impossible due to the overriding of methods that return `self`. Our model is better suited to capture self-types since it is impossible to return the `self` parameter that is also used to perform self sends.

Moreover incrementality cannot be neglected in object-oriented software engineering as indicated by [Dodani&Tsai92]: "*An important issue that must be tackled by any type system for OOP is that of incremental development and compilation.*" Dodani and Tsai developed a type system that satisfies this requirement by distinguishing two kinds of inheritance relationships: the substitutable is-a relationship, and abstraction of common behaviour occurring in abstract classes. But in order to be able to make this distinction, the object-oriented system must fulfil some requirements, among others: "*All the messages sent to the variable self must be defined*".[6]

According to this requirement of Dodani and Tsai, [Lamping93] introduced the notion of *specialisation interface* of a class, containing all the self sends performed by that class. He stressed the importance of the specialisation interface for documentation and typing purposes. In our model the specialisation interface can be automatically generated. [Lucas&al.95] formalised specialisation interfaces in a type system that clearly trades off substitutability and extendibility[7]. Note that they had to syntactically restrict the use of the `self` pseudo variable in such a way that self sends actually became explicit.

6. Orthogonally combining typing and encapsulation

The encapsulation mechanism proposed in section 4 is still too coarse-grained, since it always hides an attribute from inheriting and user clients simultaneously. For this reason, it is useful to split the encapsulation operator into two more primitive orthogonal operators: a *preventinvocation* and a *preventspecialisation* operator (still needing self sends to be recognisable). The *preventinvocation* operator prevents further clients to invoke an attribute, whereas specialisation is still allowed. The *preventspecialisation* operator prevents to specialise an attribute, while it can still be invoked. This kind of encapsulation does not make a distinction between two kinds of clients (inheritors and user clients), but differentiates between what can be done with an attribute: invocation and/or specialisation. If we prevent for instance specialisation of all attributes of an object, the object becomes a traditional non-specialisable object as appearing as the result of instantiation in class-based systems. The encapsulation operator defined in section 4 actually is the combination of *preventinvocation* and *preventspecialisation*.

In the previous section the covariance-contravariance dilemma was only solved partially. Even if the twice method performs a recognisable self send of equalContents, the extension ExtListElement still poses a problem. The specialised version of equalContents expects an ExtListElement but receives next (in the parent only known as ListElement) as actual argument. Such methods are commonly referred to as "binary methods" [Bruce&al.95]. A type system has to re-check the twice method-body in order to allow the extension ExtListElement.

To illustrate the flexibility resulting from the orthogonality between encapsulation and typing, we will show that ExtListElement can even be a subtype of ListElement. equalContents is likely to be an auxiliary method, i.e. a method not called by user clients. But the inheritor ExtListElement must be able to specialise it. In other words ExtListElement should receive as parent a version of ListElement where *preventinvocation* is applied on the method equalContents. As a result the invocations of equalContents are limited to the self sends in (the new version of) ListElement. ExtListElement would become a subtype of ListElement, since there is no a longer a covariantly overridden *invocable* method. As such, we obtain an alternative solution for typing binary methods.

We aim to formally integrate our two encapsulation operators and a type system in our model. Since our objects (or mixin-classes [Bracha92]) do not have a fixed parent, the flexibility will increase even more.

7. Conclusion

Current object-oriented languages and models are heavily based on a `self` pseudo variable, a primary construct to obtain self references. Self sends are modelled as ordinary messages with this pseudo variable as receiver. However self sends are frequently used in object-oriented programs and contain important design information. Mere self references on the contrary occur not that often and happen to be less significant concerning software design.

Therefore we turned around the traditional viewpoint. In our model the self send was the primary construct. Implicit self references turned out to be only a particular case of references subject to late binding through this self send construct. Explicit self references were considered to be superfluous.

---

[6] As noticed by [Hürsch94], Dodani and Tsai actually went a little bit too far in their positivism by allowing different concrete classes to be in subtype relationship with their common abstract superclass. Nevertheless, allowing covariant extensions remains useful.

[7] [Lucaset al.95] tackles type problems in a context of dynamic object-based inheritance. With our example we showed that the same problems appear in clas-based inheritance when incremental compilation is involved.

This switch in the basis of object-oriented concepts was necessary to orthogonally integrate fundamental software engineering aspects like encapsulation and typing. We discussed why performing self sends and returning a self reference with one and the same construct (a `self` pseudo variable) is incompatible with object-based encapsulation. The primitive self send construct allowed us to add an incremental object-based encapsulation mechanism to our model. We also indicated that incremental type systems can only be realised in languages that comply with our model. The orthogonality of inheritance, typing and encapsulation (and possibly other concepts) promises to provide a flexible, comprehensible and incremental software engineering scheme.

Acknowledgements.

We express our gratitude to our promoter Theo D'Hondt, and to Kris De Volder, Carine Lucas, Thomas Kühne, Kim Mens, Niels Boyen and Kurt Verschaeve for their useful comments on earlier versions of this paper.

References.

[Abadi&Cardelli94] M. Abadi & L. Cardelli, "A Theory of Primitive Objects: Second-Order Systems", ESOP '94 Conference Proceedings, pp. 1-25, LNCS 788, Springer-Verlag, 1994.

[Bracha92] G. Bracha, "The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance", PhD thesis, Department of Computer Science, University of Utah, 1992.

[Bracha&Lindstrom92] G. Bracha & G. Lindstrom, "Modularity meets inheritance", International Conference on Computer Languages '92 Proceedings, pp. 282-290, IEEE Computer Society, 1992.

[Bruce&al.95] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G.T Leavens & B. Pierce, "On Binary Methods", Technical report, 1995.

[Canning&al.89] P. S. Canning, W. R. Cook, W. L. Hill & W. G. Olthoff, "Interfaces for Strongly-Typed Object-Oriented Programming", OOPSLA '89 Proceedings, pp. 457-467, ACM Press, 1989.

[Cook89] W. Cook, "A Denotational Semantics of Inheritance", PhD Thesis, Brown University, 1989.

[Cook&Palsberg89] W. Cook & J. Palsberg, "A Denotational Semantics of Inheritance and its Correctness", OOPSLA '89 Conference Proceedings, pp. 433-443, ACM Press, 1989.

[Dodani&Tsai92] M. Dodani & C.-S. Tsai, "A Type System For Object-Oriented Programming Based on Abstract and Concrete Classes", ECOOP '92 Conference Proceedings, pp. 309-328, LNCS 615, Springer-Verlag, 1992.

[Ellis&Stroustrup91] M. Ellis & B. Stroustrup, "The annotated C++ reference manual", Addison-Wesley, 1991.

[Goldberg&Robson83] A. Goldberg & D. Robson, "Smalltalk-80: The language and its implementation", Addison-Wesley, 1983.

[Hürsch94] W. L. Hürsch , "Should Superclasses be Abstract?", ECOOP '94 Conference Proceedings, pp. 12-31, LNCS 788, Springer-Verlag, 1994.

[Johnson&Foote88] R. E. Johnson & B. Foote, "Designing Reusable Classes", Journal of Object-Oriented Programming 1(2), pp. 22-35, SIGS Publications, 1988.

[Kiczales93] G. Kiczales, "Traces: A cut at the 'make isn't generic' problem", ISOTAS '93 Conference Proceedings, LNCS 742, Springer-Verlag, 1993.

[Lamping93] J. Lamping, "Typing the specialisation interface", OOPSLA '93 Conference Proceedings, pp. 201-214, ACM Press, 1993.

[Lucas&al.95] C. Lucas, K. Mens & P. Steyaert, "Typing Dynamic Inheritance. A Trade-Off between Substitutability and Extensibility", submitted to TAPOS, special issue on typing.

[Mens&al.95] T. Mens, K. Mens & P. Steyaert, "OPUS: A calculus for Modelling Object-Oriented Concepts", OOIS'94 Conference Proceedings, pp. 152-165, Springer-Verlag, 1995.

[Meyer88] B. Meyer, "Object-oriented Software Construction", Prentice Hall, 1988.

[Pree94] W. Pree, "Design Patterns for Object-Oriented Software Development", ACM Press / Addison-Wesley, 1994.

[Rumbaugh&al.91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall, 1991.

[Shah&al.89] A. Shah, J. Rumbaugh, J. Hamel, R. Borsari, "DSM: An Object-Relationship Modeling Language", OOPSLA '89 Conference Proceedings, pp. 191-202, ACM Press, 1989.

[Steyaert94] P. Steyaert, "Open Design of Object-Oriented Languages: A foundation for Specialisable Reflective Language Frameworks", PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1994.

[Szyperski92] C. A. Szyperski, "Import is not inheritance. Why we need both: modules and classes", ECOOP '92 Conference Proceedings, pp. 19-32, LNCS 615, Springer-Verlag, 1992.

[Tennent91] R. D. Tennent, “Semantics of Programming Languages”, Prentice Hall, 1991.

[Van Limberghen&Mens95] M. Van Limberghen & T. Mens, "Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems", Submitted to OOS Journal, 1995

[Van Limberghen95] M. Van Limberghen, "Normalising class components", submitted to OO-ER 95.

A. Appendix

In this appendix we will give the complete syntax and denotational semantics of our formal model extended with encapsulation. A PROLOG implementation can be obtained from the authors by request.

Extended syntax:

Expression	→	Method | Compose | Encaps | SelfSend | ObjectSend | SuperCall | ArgRef
Method	→	Label "(" ArgRef ")" "=" Expression
Compose	→	"(" Expression "&" Expression ")"
Encaps	→	"[" Expression "]" "(" Label ")"
SelfSend	→	Label Argument
ObjectSend	→	"(" Expression ")" Label Argument
SuperCall	→	"`super`" Argument
Argument	→	"(" Expression ")" | "(" ε ")"
ArgRef	→	"#" Label
Label	→	Letter { Letter }
Letter	→	"a" | "b" | ... | "z"

Semantic domains:

ArgEnv	=	**Label**	→	**Generator**
MethodEnv	=	**Label**	→	**MethodBody**
Generator	=	**MethodEnv**	→	**MethodEnv**
MethodBody	=	**ArgGen**	→	**ArgGen**
ArgGen	=	**Generator**	→	**Generator**

Semantic function:

ª º : Expression → **Generator**

ª E º = ⟦ E ⟧ ⊥ ⊥ ⊥

⟦ ⟧ : Expression → **MethodEnv** → **ArgEnv** → **ArgGen** → **Generator**

$⟦ (E_1)L(E_2) ⟧ = λself.λargs.λsuper.\ \mathit{send}\ \mathbf{Y}(⟦ E_1 ⟧\ self\ args\ super)\ L\ (⟦ E_2 ⟧\ self\ args\ super)$

$⟦ L_1(\#L_2)=E ⟧ = λself.λargs.λsuper.\ λself2.\{ L_1 → λsuper2.λarg.⟦ E ⟧\ self2\ (args +_r \{L_2 → arg\})\ super2$
$\}$

$⟦ E_1\ \&\ E_2 ⟧ = λself.λargs.λsuper.\ (⟦ E_1 ⟧\ self\ args\ super)\ 7\ (⟦ E_2 ⟧\ self\ args\ super)$

$⟦ [E](L) ⟧ = λself.λargs.λsuper.\ \mathit{encaps}\ (⟦ E ⟧\ self\ args\ super)\ L$

$⟦ \#L ⟧ = λself.λargs.λsuper.\ args\ L$

$⟦ L(E_2) ⟧ = λself.λargs.λsuper.\ \mathit{send}\ self\ L\ (⟦ E_2 ⟧\ self\ args\ super)$

$⟦ super(E_2) ⟧ = λself.λargs.λsuper.\ super\ (⟦ E_2 ⟧\ self\ args\ super)$

$⟦ ε ⟧ = λself.λargs.λsuper.\ ⊥$

send : **MethodEnv** → **Label** → **Generator** → **Generator**
send receiver L arg = receiver L ⊥ arg

7 : **Generator**→ **Generator**→ **Generator**
$g_1\ 7\ g_2 = λself.\ [\ (g_1\ self) +_r λlabel.λsuper.\ g_2\ self\ label\ ((\{label → λsuper.super\} +_r g_1\ self)\ label\ super)$
$]$

encaps : **Generator**→ **Label** → **Generator**
$\mathit{encaps}\ g\ L = λself.\ [\ \mathbf{Y}(\ λfix.λself2.\ g\ (self2 +_r \{ L → λsuper.\ fix\ self2\ L\ ⊥ \})\ )\ self \setminus L\ ]$

$e_1 +_r e_2 = λx.\ \mathit{if}\ (e_2\ x) = ⊥\ \mathit{then}\ (e_1\ x)\ \mathit{else}\ (e_2\ x)$

$\{key → val\} = λx.\ \mathit{if}\ x = key\ \mathit{then}\ val\ \mathit{else}\ ⊥$

$e \setminus L = λx.\ \mathit{if}\ x = L\ \mathit{then}\ ⊥\ \mathit{else}\ (e\ x)$