A Layered Calculus for Encapsulated Object Modification

Theoretical Results

— Draft version - Do not distribute —

Kim Mens, Kris De Volder, Tom Mens

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

Abstract. *In this paper we formally present a layered calculus for encapsulated modification of objects. Its denotational as well as operational semantics are given. The confluency of the calculus is proven, and a translation of λ-calculus into our calculus is presented.*

1 Motivation

This paper is a theoretical version of an extended abstract [Mens et al. 96], submitted to the third Foundations of Object-Oriented Languages Workshop. The extended abstract explains why current prototype-based languages suffer from an inherent conflict between inheritance and encapsulation, and presents a calculus for dynamic object modification in which the conflict is resolved.

Essentially the problem in prototype-based languages stems from the fact that inheritance and message sending are both performed on objects. Therfore, in analogy to class-based languages, the problem can be solved by making a distinction between objects for message sending and "inheritable entities" — called *generators* [Cook 89] — for specialisation. For this reason, the proposed calculus has a two layered syntax, clearly distinguishing generator expressions from object expressions. The top layer deals with objects and message sending. The second layer deals with generators and inheritance with late binding of self. Due to the layering and a careful scoping of generator names, the use of generators is restricted to the inside of the object so that encapsulation cannot be breached. Due to this clean interaction between encapsulation and inheritance, the calculus can be used as a foundation for prototype-based languages without the inheritance-encapsulation conflict.

To highlight the essence of the model features such as typing, object identity, state and private attributes have not been included. Another important concept, the ability to perform super sends, is not explicitly present in the syntax, but can be straightforwardly modelled using the other syntactic constructs.

2 Syntax

2.1 Syntactic Domains

There are only two kind of expressions in the syntax: object expressions and generator expressions. Identifiers can either be object expressions or generator expressions, depending on the context in which they occur.

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

2.2 Production Rules

In the following grammar in BNF, terminal symbols are printed in bold. Identifiers `I [ Ident` are also considered to be terminal symbols.

Object	→	Object.Ident(Object)	*message send*
	|	[Generator]	*object creation*
	|	[Generator Δ Generator]	*delegation*
	|	Ident	*argument reference*
Generator	→	Generator ; Generator	*generator composition*
	|	Ident (Ident) = Ident # Object	*method*
	|	> Object <	*object to generator conversion*
	|	Ident	*self reference*
	|	ε	*empty generator*

2.3 Super Calls

We will not go into details of the syntax, since this has already been done in [Mens et al. 96]. However there is one extra production rule that introduces a delegation operator. The rationale behind and the use of this operator will be explained here.

Since generators represent templates for objects with a still undetermined self[1], it is natural to define an operator that explicitly binds the self of a generator. We used the symbol Δ to denote this operator. The self of a generator is again a generator in our model, thus the Δ-operator is a binary operator on generators returning an object. Basically, the Δ-operator is a delegation operator. Its first argument is a generator in which messages sent to the resulting object will be looked up. The second argument represents the generator that is used for internal self references.

Using the Δ-operator, it is possible to model super sends to invoke parent operations in an inheritance chain. Consider the example of a two dimensional point which can be extended to a three dimensional point. The three dimensional point is created by overriding the parent's `sumOfSquares` method. In the implementation of the new `sumOfSquares` method a super call is performed. Super calls are looked up in the parent's generator (`Super`), while internal self references of the parent are redirected to the specialisation (`Self`), so messages to `[SuperΔSelf]` behave as super sends in Smalltalk.

```
[ x = 1;
  setx(newx) = Self # [Self ; x = newx];
  y = 2;
  sety(newy) = Self # [Self ; y = newy];
  isOrigin = Self # [Self].distance.isZero;
  distance = Self # [Self].sumOfSquares.sqrt;
  sumOfSquares = Self # [Self].getx.sqr.add([Self].gety.sqr);
  thirdDimension = Super # [
    Super;
    z = 3;
    setz(newz) = Self # [Self ; z = newz];
     sumOfSquares = Self # [SuperΔSelf].sumOfSquares.add([Self].getz.sqr) ]
]
```

3 Semantics

A formal model can be defined by giving a formal description of both its syntax and semantics. In general, there are several ways to define semantics. In the denotational approach a semantics is given as an interpretation function that directly maps expressions to their meaning in some domain of semantical values. In the operational approach a set of reduction rules is given (or alternatively an equational theory based on axioms and inference rules). In the following subsections these two types of semantics are presented.

3.1 Denotational semantics

The denotational semantics will be presented following the notation of [Schmidt 86], and square brackets are used for parameterised domains.

[1] This is explained in [Mens et al. 96], and can also be seen from the denotational semantics presented further.

Semantic Domains

An `Object` is represented as a record of methods. Each `Method` expects an `Object` as argument and returns an `Object` after evaluation. To allow late binding a generator is a template for an object with a still undetermined (late bound) self. It is a function mapping a self `Generator` onto an `Object`.

Record[α]	≡	Ident → Maybe[α]
Maybe[α]	≡	α ⊕ Unit
Object	≡	Record[Method]
Method	≡	Object → Object
Generator	≡	Generator → Object

Auxiliary Functions

Before presenting the semantic functions we need some auxiliary functions to create and manipulate records.

An empty record

```
{}            :    Record[α]
{}            ≡    λx.inMaybe[α]()
```

A single slot record

```
{…→…}         :    Ident → α → Record[α]
{key→val}     ≡    λx.(x=key → inMaybe[α](val) o inMaybe[α]())
```

Right preferential record concatenation

```
$… +_r …$     :    Record[α] → Record[α] → Record[α]
$f_1 +_r f_2$ ≡    λx.cases ($f_2$ x) of
                     isUnit() → $f_1$ x o
                     isα()    → $f_2$ x
                   end
```

Method lookup

```
lookup        :    Record[α] → Ident → α
lookup        ≡    λr.λk.cases (r k) of
                     isα(v)   → v o
                     isUnit() → ⊥
                   end
```

Scoping of generator names and argument names

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

Semantics of an Object Expression

The semantics of an object expression is a function parameterised with the two lexical environments and returning an object.

$$[\![\mathbf{ObjExpr}]\!]_O : \mathtt{Record[Object]} \rightarrow \mathtt{Record[Generator]} \rightarrow \mathtt{Object}$$

The semantics of a message send $O_r.I(O_a)$ is defined by looking up the message I in the receiver object O_r, while providing the object O_a as argument.

$$[\![O_r.I(O_a)]\!]_O\ a\ g \quad \equiv \quad (\mathtt{lookup}\ ([\![O_r]\!]_O\ a\ g)\ I)\ ([\![O_a]\!]_O\ a\ g)$$

The delegation expression $[G_1 \Delta G_2]$ creates an object from G_1 and redirects all self sends to G_2.

$$[\![[G_1 \boldsymbol{\Delta} G_2]]\!]_O\ a\ g \quad \equiv \quad ([\![G_1]\!]_G\ a\ g)\ ([\![G_2]\!]_G\ a\ g)$$

The object creation expression $[G]$ creates an object from a generator by making the generator refer to itself.

$$[\![[G]]\!]_O\ a\ g \quad \equiv \quad [\![[G \boldsymbol{\Delta} G]]\!]_O\ a\ g = \quad ([\![G]\!]_G\ a\ g)\ ([\![G]\!]_G\ a\ g)$$

Evaluation of an identifier on object level occurs by looking up this identifier in the record of actual arguments.

$$\llbracket \mathtt{I} \rrbracket_{O}\ \mathtt{a}\ \mathtt{g} \qquad \equiv \qquad \mathtt{lookup\ a\ I}$$

Semantics of a Generator Expression

The semantics of a generator expression is similar to that of an object expression. It is again a function requiring two lexical environment parameters but it returns a generator rather than an object.

$$\llbracket \mathbf{GenExpr} \rrbracket_{G} : \mathtt{Record[Object]} \rightarrow \mathtt{Record[Generator]} \rightarrow \mathtt{Generator}$$

The semantics of a composition of generators is a new generator of which the self is distributed over its constituents.

$$\llbracket \mathtt{G}_1 \mathbf{;} \mathtt{G}_2 \rrbracket_{G}\ \mathtt{a}\ \mathtt{g} \qquad \equiv \qquad \lambda\mathtt{self.}(\llbracket \mathtt{G}_1 \rrbracket_{G}\ \mathtt{a}\ \mathtt{g})\ \mathtt{self} +_{r} (\llbracket \mathtt{G}_2 \rrbracket_{G}\ \mathtt{a}\ \mathtt{g})\ \mathtt{self}$$

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

$$\llbracket \mathtt{I}_m \mathbf{(} \mathtt{I}_a \mathbf{)=} \mathtt{I}_s \mathbf{\#} \mathtt{O} \rrbracket_{G}\ \mathtt{a}\ \mathtt{g} \qquad \equiv \qquad \lambda\mathtt{self.}\{\mathtt{I}_m \rightarrow \mathtt{body}\}$$
$$\textit{where} \qquad \mathtt{body} = \lambda\mathtt{arg.}\llbracket \mathtt{O} \rrbracket_{O}\ (\mathtt{a} +_{r} \{\mathtt{I}_a \rightarrow \mathtt{arg}\})\ (\mathtt{g} +_{r} \{\mathtt{I}_s \rightarrow \mathtt{self}\})$$

The semantics of a self reference and an empty generator are straightforward.

$$\llbracket \mathtt{I} \rrbracket_{G}\ \mathtt{a}\ \mathtt{g} \qquad \equiv \qquad \mathtt{lookup\ g\ I}$$
$$\llbracket \varepsilon \rrbracket_{G}\ \mathtt{a}\ \mathtt{g} \qquad \equiv \qquad \lambda\mathtt{self.}\{\}$$

The "$>\ldots<$" operator converts an object into a generator of which the self argument is ignored. An object expression $\mathtt{O}$ can be extended by turning it into a generator $\mathtt{>O<}$ and subsequently composing it with some other generator. This is not really inheritance and does not breach encapsulation because it does not involve late binding of self in the object under extension.

$$\llbracket \mathtt{>O<} \rrbracket_{G}\ \mathtt{a}\ \mathtt{g} \qquad \equiv \qquad \lambda\mathtt{self.}\llbracket \mathtt{O} \rrbracket_{O}\ \mathtt{a}\ \mathtt{g}$$

3.2 Operational Semantics

The operational semantics is given by a set of reduction rules, transforming expressions to their normal form. The following notations are used:

Æ	one-step reduction
	reflexive and transitive closure of Æ
$\mathtt{I,J,}\ldots$	elements of $\mathtt{Ident}$
$\mathtt{O,O}_i$	elements of $\mathtt{ObjExpr}$
$\mathtt{G,G}_i$	elements of $\mathtt{GenExpr}$

Actually, there is only one reduction rule in our operational semantics:

$\ldots\mathtt{O}_r\mathbf{.}\mathtt{I}\mathbf{(}\mathtt{O}_a\mathbf{)}\ldots$ Æ $\ldots \mathit{apply}(\mathtt{O}_r,\mathtt{I},\mathtt{O}_a)\ldots$ if $\mathit{apply}(\mathtt{O}_r,\mathtt{I},\mathtt{O}_a)$ is defined

where *apply* is a partial function defined as follows:

$$\mathit{apply} : (\mathbf{ObjExpr} \times \mathbf{Ident} \times \mathbf{ObjExpr}) \rightarrow \mathbf{ObjExpr}$$
$$\mathit{apply}(\mathbf{[}\mathtt{G}_1\mathbf{\Delta}\mathtt{G}_2\mathbf{]}\ ,\mathtt{I},\mathtt{O}_a) \qquad \equiv \mathit{lookup}(\mathtt{G}_1,\mathtt{I},\mathtt{G}_2,\mathtt{O}_a)$$
$$\mathit{apply}(\mathbf{[}\mathtt{G}\mathbf{]}\ ,\mathtt{I},\mathtt{O}_a) \qquad \equiv \mathit{lookup}(\mathtt{G},\mathtt{I},\mathtt{G},\mathtt{O}_a)$$

apply is undefined for all other cases

and *lookup*[2] is a partial function defined by induction on its first argument $\mathtt{G}_1$ as follows:

[2] Do not confuse this *lookup* function with the one we used in the denotational semantics.

$\mathit{lookup} : (\mathbf{GenExpr} \times \mathbf{Ident} \times \mathbf{GenExpr} \times \mathbf{ObjExpr}) \rightarrow \mathbf{ObjExpr}$

$\mathit{lookup}(G_1 ; G_r \ , I, G_2, O_a)$	$\equiv \mathit{lookup}(G_r, I, G_2, O_a)$	if $\mathit{lookup}(G_r, I, G_2, O_a)$ is defined
	$\equiv \mathit{lookup}(G_1, I, G_2, O_a)$	otherwise
$\mathit{lookup}(I_m(I_a)=I_s\#O \ , I, G_2, O_a)$	$\equiv [O_a \mid I_a] \ ([G_2 \mid I_s] \ O)$	if $I = I_m$
	is undefined	otherwise
$\mathit{lookup}(>O< \ , I, G_s, O_a)$	$\equiv \mathit{apply}(O, I, O_a)$	
$\mathit{lookup}(J \ , I, G_s, O_a)$	is undefined	
$\mathit{lookup}(\varepsilon \ , I_n, G_s, O_a)$	is undefined	

Two remarks need to be made in the above definition:

- In $[O_a \mid I_a] \ ([G_2 \mid I_s] \ O)$ there is one difficulty that does not occur in the denotational semantics (where argument names and self references are stored in different records), namely that self reference names can collide with argument reference names. This problem can be solved by using two disjoint sets of names. (In the examples self references will always start with a capital letter, while argument references begin with a lower case letter.)
- We use a notion of substitution $[E \mid I] F$ of an identifier I by an expression E in an expression F, which is defined inductively on the form of F:

Definition (Substitution)

$[E \mid I] \ O_r . I_m(O_a)$	$\equiv$	$[E \mid I] O_r . I_m([E \mid I] O_a)$	
$[E \mid I] \ [G_1 \Delta G_2]$	$\equiv$	$[[E \mid I] G_1 \Delta [E \mid I] G_2]$	
$[E \mid I] \ [G]$	$\equiv$	$[[E \mid I] G]$	
$[E \mid I] \ G_1 ; G_2$	$\equiv$	$[E \mid I] G_1 ; [E \mid I] G_2$	
$[E \mid I] \ I_m(I_a)=I_s\#O_r$	$\equiv$	$I_m(I_a)=I_s\#O_r$	if ($E \in \mathtt{ObjExpr}$ and $I_a = I$) or ($E \in \mathtt{GenExpr}$ and $I_s = I$)
	$\equiv$	$I_m(I_a)=I_s\#[E \mid I] O_r$	if ($E \in \mathtt{ObjExpr}$ and $I_a \neq I$) or ($E \in \mathtt{GenExpr}$ and $I_s \neq I$)
$[E \mid I] \ >O<$	$\equiv$	$>[E \mid I] O<$	
$[E \mid I] \ \varepsilon$	$\equiv$	ε	
$[E \mid I] \ J$	$\equiv$	E	if $J = I$ and E and I are both object expressions or both generator expressions
	$\equiv$	J	in all other cases

4 Confluency

One of the most important theoretical properties of a calculus is that it should be confluent, i.e. that different reduction paths lead to a unique result. This property is also commonly known as the *diamond property*.

Theorem 1 (Confluency or Diamond Property for)
If $E \quad E_1$ and $E \quad E_2$ then there is some E' such that $E_1 \quad E'$ and $E_2 \quad E'$.

The proof follows a rather elegant variation of the Tait and Martin-Löf proof of the Church-Rosser theorem for λ-calculus, as presented in [Takahashi 95]. The key notion of the proof is parallel reduction which intuitively corresponds to reducing a number of redexes (possibly overlapping each other) simultaneously.

Definition (Parallel Reduction)

(P1) $O_r.I(O_a)$ $O_r'.I(O_a')$ if O_r O_r' and O_a O_a'

(P2) $O_r.I(O_a)$ $\mathit{apply}(O_r',I,O_a')$ if $\mathit{apply}(O_r,I,O_a)$ is defined, O_r O_r' and O_a O_a'

(P3) $[G_1\Delta G_2]$ $[G_1'\Delta G_2']$ if G_1 G_1' and G_2 G_2'

(P4) $[G]$ $[G']$ if G G'

(P5) $G_1;G_2$ $G_1';G_2'$ if G_1 G_1' and G_2 G_2'

(P6) $I_m(I_a)=I_s\#O$ $I_m(I_a)=I_s\#O'$ if O O'

(P7) $>O<$ $>O'<$ if O O'

(P8) I I

(P9) ε ε

Based on the inductive definition of   , we can easily verify the following properties. The first property states that a single Æ reduction is a special case of a parallel reduction, whereas the second property intuitively states that all reductions that are done in parallel can also be done step by step.

Property 1

If E Æ E' then E E'.

Proof:

By induction on the context of the redex in E.

Property 2

If E E' then E E'.

Proof:

By induction on the structure of E. Furthermore, in the inductive case where E is of the form $O_r.I(O_a)$ with $\mathit{apply}(O_r,I,O_a)$ defined, we also need to make use of lemma 2 which is given below.

From these two properties we know that   is the reflexive, transitive closure of . Therefore, to prove the confluency theorem for   it suffices to show the confluency property of .

Theorem 1 bis (Confluency or Diamond Property for)

If E E_1 and E E_2 then there is some E' such that E_1 E' and E_2 E'.

But actually we can prove the following stronger statement more easily:

Property 3

For each E, there is some E^* such that for each F holds: if E F, then F E^*.

The important thing to notice in this property is that E^* is a term determined by E, but independent from F. A possible E^* that satisfies the property can be constructed from E by contracting all the redexes existing in E simultaneously. This E^* can be defined by induction on the structure of the expression E.

Definition

($P1^*$) $(O_r.I(O_a))^* \equiv O_r^*.I(O_a^*)$ if $\mathit{apply}(O_r,I,O_a)$ is undefined

($P2^*$) $(O_r.I(O_a))^* \equiv \mathit{apply}(O_r^*,I,O_a^*)$ if $\mathit{apply}(O_r,I,O_a)$ is defined

($P3^*$) $([G_1\Delta G_2])^* \equiv [G_1^*\Delta G_2^*]$

($P4^*$) $([G])^* \equiv [G^*]$

($P5^*$) $(G_1;G_2)^* \equiv G_1^*;G_2^*$

($P6^*$) $(I_m(I_a)=I_s\#O)^* \equiv I_m(I_a)=I_s\#O^*$

($P7^*$) $(>O<)^* \equiv >O^*<$

($P8^*$) $I^* \equiv I$

($P9^*$) $\varepsilon^* \equiv \varepsilon$

Before continuing, we will prove some additional lemmas that are needed to verify property 3.

Lemma 1

For each E holds: $E \quad E^*$

Proof:

By induction on the structure of E, and because of the similarity of the definitions of and E^*.

Lemma 2

If $apply(O_r,I,O_a)$ is defined, $O_r \quad O_r'$ and $O_a \quad O_a'$,
then $apply(O_r',I,O_a')$ is defined and $apply(O_r,I,O_a) \quad apply(O_r',I,O_a')$.

Proof:

Let $O_r=[G_1 \Delta G_2]$ *and* $O_r'=[G_1' \Delta G_2']$

or $O_r=[G_1]$, $O_r'=[G_1']$, $G_2=G_1$ *and* $G_2'=G_1'$

(in all other cases, $apply(O_r,I,O_a)$ *is undefined)*

Then it is easy to see that the above formulation of the lemma is equivalent with:

If $lookup(G_1,I,G_2,O_a)$ *is defined,* $G_1 \quad G_1'$, $G_2 \quad G_2'$ *and* $O_a \quad O_a'$
then $lookup(G_1',I,G_2',O_a')$ *is defined*
and $lookup(G_1,I,G_2,O_a) \quad lookup(G_1',I,G_2',O_a')$.

which can be verified by induction on the structure of G_1. Because $lookup$ is defined in terms of substitution $[E|I]$, somewhere in the proof we need to use lemma 4.

Lemma 3

If $apply(O_r,I,O_a)$ is defined then $apply(O_r^*,I,O_a^*)$ is defined.

Proof:

Follows immediately from lemma 1 and lemma 2.

Lemma 4

If $E \quad E'$ and $F \quad F'$ then $[E|I]F \quad [E'|I]F'$.

Proof:

By induction on the structure of F.

The proof of property 3 is fairly straightforward, and is based on induction on the structure of E. Only when E is of the form $O_r.I(O_a)$ with $apply(O_r,I,O_a)$ defined one has to be a little more careful, because E can either be (parallelly) reduced according to (P_1) or according to (P_2).

Proof of property 3: *(by induction on the structure of E)*

<u>Case 1</u>: $E \equiv O_r.I(O_a)$ *with* $apply(O_r,I,O_a)$ *undefined*

If $E \quad F$ *then* $F \equiv O_r'.I(O_a')$ *with* $O_r \quad O_r'$ *and* $O_a \quad O_a'$ *(P_1)*

Moreover $O_r' \quad O_r^*$ *and* $O_a' \quad O_a^*$ *(because O_r and O_a are subterms of E, and by using the structural induction hypothesis)*

Hence $F \quad O_r^*.I(O_a^*)$ *(P_1)*

$\equiv E^*$ *(P_1^*)*

<u>Case 2</u>: $E \equiv O_r.I(O_a)$ *with* $apply(O_r,I,O_a)$ *defined*

If $E \quad F$ *then there are 2 possibilities, depending on whether (P_1) or (P_2) is applied.*

Case 3.1: $F \equiv o_r'.I(o_a')$ *with* o_r o_r' *and* o_a o_a' *(P1)*

Moreover o_r' o_r^* *and* o_a' o_a^* *(structural induction hypothesis)*

Hence F $o_r^*.I(o_a^*)$ *(P1)*

Æ $apply(o_r^*,I,o_a^*)$ *def. of Æ and lemma 3*

$apply(o_r^*,I,o_a^*)$ *property 1*

$\equiv E^*$ *(P2*)*

Case 3.2: $F \equiv apply(o_r',I,o_a')$ *with* o_r o_r' *and* o_a o_a' *(P2)*

$apply(o_r',I,o_a')$ *is defined because of lemma 2*

Moreover o_r' o_r^* *and* o_a' o_a^* *(structural induction hypothesis)*

Hence F $apply(o_r^*,I,o_a^*)$ *(lemma 2)*

$\equiv E^*$ *(P2*)*

All other cases can be shown in a similar way

5 Translation of λ-calculus

An interpretation ⟦⟧ of expressions in our calculus in terms of λ-expressions with records has been given by our denotational semantics. It is also fairly straightforward to define a function ⩽⩾ translating λ-expressions into expressions in our syntax.

In the following, upper case letters are used to denote meta variables for λ-expressions.

Definition

```
LambdaExpr          ≡      set of all λ-expressions
LambdaIdent         ≡      set of all λ-variables
```

Definition (λ-translation)

Let `A, F ∈ LambdaExpr` and `X,D ∈ LambdaIdent`

```
⩽⩾ : LambdaExpr ⟶ ObjExpr
⩽λX.A⩾      ≡     [eval(⩽X⩾)=⩽D⩾#⩽A⩾]        where D is free in A
⩽F A⩾       ≡     ⩽F⩾.eval(⩽A⩾)
⩽X⩾         ≡     X                          straightforward translation of λ-identifiers
```

To prove that this is a good translation, we show that it respects β-reduction.

Property 4

`⩽(λX.B)A⩾ Æ ⩽[A|X]B⩾ ∀ A,B ∈ LambdaExpr, ∀ X ∈ LambdaIdent`

Proof:

```
  ⩽(λX.B)A⩾    =   ⩽λX.B⩾.eval(⩽A⩾)
               =   [eval(⩽X⩾)=⩽D⩾#⩽B⩾].eval(⩽A⩾)                    with D free in B
               Æ   apply([eval(⩽X⩾)=⩽D⩾#⩽B⩾], eval, ⩽A⩾)
               =   lookup(eval(⩽X⩾)=⩽D⩾#⩽B⩾, eval, eval(⩽X⩾)=⩽D⩾#⩽B⩾, ⩽A⩾)
               =   [⩽A⩾|⩽X⩾] ([eval(⩽X⩾)=⩽D⩾#⩽B⩾|⩽D⩾] ⩽B⩾)
               =   [⩽A⩾|⩽X⩾] ([⩽λX.B⩾|⩽D⩾] ⩽B⩾)
               =   [⩽A⩾|⩽X⩾] ⩽[λX.B|D]B⩾                            (lemma)
               =   [⩽A⩾|⩽X⩾] ⩽B⩾                                    (D free in B)
               =   ⩽[A|X]B⩾                                         (lemma)
```

In the above proof, we used the following lemma:

Lemma 5

`[⩽A⩾|⩽X⩾]⩽B⩾ = ⩽[A|X]B⩾ ∀ A,B ∈ LambdaExpr, ∀ X ∈ LambdaIdent`

where the `[|]` on the left denotes our substitution mechanism, and the one on the right denotes substitution in λ-calculus.

Proof:

The proof of this lemma is left to the reader.

6 Implementations

We have made two implementations of our calculus in the functional programming language Gofer. The first implementation is based on the operational semantics, the second one reflects the denotational semantics.

7 Acknowledgements

We are indebted to Dirk Thierbach and Luca Cardelli for helping us with some proofs.

8 References

[Cook 89] Cook W. - 1989. A Denotational Semantics of Inheritance, Ph.D.-Thesis, Brown University.

[Mens et al. 96] Mens, K.; De Volder, K.; Mens, T. & Steyaert, P. - 1996. A Layered Calculus for Encapsulated Object Modification; Extended Abstract; Submitted to Foundations of Object-Oriented Languages Workshop 3.

[Schmidt 86] Schmidt, D. A. - 1986. Denotational Semantics: A Methodology for Language Development; Allyn and Bacon, Inc.

[Takahashi 95] Takahashi, M. - 1995. Parallel Reductions in λ-Calculus; Information and Computation, Vol. 118; pp. 120-127.