

1. Object Calculus

In this section we will introduce a

calculus of objects

that gives a simple but powerful mathematical model to study object based languages.

Object calculus is to OO languages what lambda calculus is to functional languages

1.1. (Type-free) Lambda calculus revisited

Recall the syntax of lambda terms

$$t ::= x \mid (t_1 t_2) \mid (\lambda x . t)$$

$(t_1 t_2)$ is an application term

$(\lambda x . t)$ is a lambda abstraction term, and x is said to be bound in $\lambda x . t$

Beta reduction

The main computation step is a one step
beta reduction

$$\text{redex} \blacktriangleleft (\lambda x . M) N \Rightarrow^\beta M\{x \rightarrow N\} \blacktriangleright \text{contractum}$$

where $M\{x \rightarrow N\}$ means replace all free
occurrences of x in M by N .

Lambda computability

We can define the Church numerals

$$n \equiv (\lambda x . \lambda y . x^n y)$$

e.g. $3 \equiv (\lambda x . \lambda y . x x x y)$

A partial function $f : \mathbb{N} \rightarrow \mathbb{N}$ is lambda definable iff there exists a lambda term t_f such that

$$f(m) = n \iff t_f (\lambda x . \lambda y . x^m y) \Rightarrow^{\beta^*} (\lambda x . \lambda y . x^n y)$$

where \Rightarrow^{β^*} is many step beta reduction.

Turing Completeness of Lambda

The functions on \mathbb{N} that are Lambda definable are all and only the Turing machine computable functions.

So lambda calculus is a Turing complete model of functional programming

1.2. Type-free Object Calculus

(From Abadi and Cardelli, *A Theory of Objects*, Springer Verlag, 1996)

Syntax of terms

$a, b ::=$

x | variable

$[l_i = \zeta(x_i)b_i \quad i \in 1..n]$ | object formation

$a.l$ | method invocation

$a.l \leftarrow \zeta(x)b$ | method update

Intuitively an object with n methods labelled l_1, \dots, l_n is represented by the term

$$[l_1 = \zeta(x_1)b_1, \dots, l_n = \zeta(x_n)b_n]$$

which we write in short form as

$$[l_i = \zeta(x_i)b_i \quad i \in 1..n]$$

The letter ζ (sigma) is a binder for the self parameter x in the body b of the expression

$$\zeta(x)b$$

Compare the term $\zeta(\mathbf{x})\mathbf{b}$ with $(\lambda \mathbf{x} \cdot \mathbf{t})$.

Notice that an object has a fixed number of labelled attributes.

We identify objects that differ only in the order of their components, e.g.

$$\begin{aligned} [l_1 = \zeta(\mathbf{x}_1)\mathbf{b}_1 , l_2 = \zeta(\mathbf{x}_2)\mathbf{b}_2] = \\ [l_2 = \zeta(\mathbf{x}_2)\mathbf{b}_2 , l_1 = \zeta(\mathbf{x}_1)\mathbf{b}_1] \end{aligned}$$

The term $o.l$ means *invoke the method labelled l in the object o .*

The term $o.l \Leftarrow \zeta(x)b$ means *update the method l of object o with the method $\zeta(x)b$*

1.2.1. Simple Example

Here is an object that contains 2 methods

$$[l_1 = \zeta(x_1)[] , l_2 = \zeta(x_2) x_2.l_1]$$

Method labelled l_1 returns the *empty object* [].

Method labelled l_2 invokes the first method through the self parameter x_2 .

1.2.2. Data attributes

Q: So far objects only seem to have method attributes, what about data?

A: A method body with an occurrence of the self variable is termed a proper method, otherwise it is termed a (data) field, since it behaves like a constant

Simple Example

The method

$$\zeta(x_1)[]$$

returns a constant value (the empty object)
since it doesn't contain the self parameter x_1 .

So $l_1 = \zeta(x_1)[]$ behaves like a data field.

1.2.3. Notation for Data Fields

We write

$[\dots l = b \dots]$ for $[\dots l = \zeta(x)b \dots]$ with unused x .

So $l = b$ is a *data field*.

We write

$o.l := b$ for $o.l \Leftarrow \zeta(x)b$ with unused x .

So $o.l := b$ is a *data field update* (assignment).

1.2.4. Operational Semantics

The idea is to copy the substitution semantics of the lambda calculus ... so for

$$o \equiv [l_i = \zeta(x_i)b_i \quad i \in 1..n]$$

$$o.l_j \Rightarrow b_j \{ x_j \rightarrow o \}$$

$$(o.l_j \Leftarrow \zeta(y)b) \Rightarrow [l_j = \zeta(y)b, l_i = \zeta(x_i)b_i \quad i \in \{1..n\} - \{j\}]$$

$o.l_j \Rightarrow b_j \{ x_j \rightarrow o \}$ means reduce $o.l_j$ in one step by replacing the self parameter x_j with the object o in the body b_j .

$(o.l_j \Leftarrow \zeta(y)b) \Rightarrow$

$[l_j = \zeta(y)b , l_i = \zeta(x_i)b_i \quad i \in \{1..n\} - \{j\}]$

means update the method l_j in the object o in one step by replacing the current method body $\zeta(x_j)b_j$ by $\zeta(y)b$.

Simple Example

Let $o_1 \equiv [1 = \zeta(x)[]]$. Then

$$o_1.1 \Rightarrow []\{ x \rightarrow o_1 \} \equiv []$$

and

$$o_1.1 \Leftarrow \zeta(x) o_1 \Rightarrow [1 = \zeta(x) o_1].$$

We could also write $o_1 \equiv [1 = []]$, and

$$o_1.1 \Leftarrow \zeta(x) o_1 \text{ as } o_1.1 := o_1$$

Examples: self substitution

Let $o_2 \equiv [1 = \zeta(x) x.1]$. Then

$o_2.1 \Rightarrow x.1 \{ x \rightarrow o_2 \} \equiv o_2.1 \Rightarrow \dots$ infinite loop!

Self substitution allows a method to return self.

Let $o_3 \equiv [1 = \zeta(x) x]$. Then

$o_3.1 \Rightarrow x \{ x \rightarrow o_3 \} \equiv o_3$

Self substitution also allows a method to modify self.

Let $o_4 \equiv [1 = \zeta(y) (y.l \leftarrow \zeta(x) x)]$.

Then

$$o_4.l \Rightarrow o_4.l \leftarrow \zeta(x) x \Rightarrow o_3$$

1.2.4.1. Inductive Definition of Free Variables

- $FV(\mathbf{x}) = \{\mathbf{x}\}$
- $FV(\zeta(\mathbf{y})\mathbf{b}) = FV(\mathbf{b}) - \{\mathbf{y}\}$
- $FV([\mathbf{l}_i = \zeta(\mathbf{x}_i)\mathbf{b}_i]_{i \in 1..n}) = \bigcup_{i \in 1..n} FV(\zeta(\mathbf{x}_i)\mathbf{b}_i)$
- $FV(\mathbf{a.l}) = FV(\mathbf{a})$
- $FV(\mathbf{a.l} \Leftarrow \zeta(\mathbf{y})\mathbf{b}) = FV(\mathbf{a}) \cup FV(\zeta(\mathbf{y})\mathbf{b})$

1.2.4.2. Definition: Object Substitution

1. $(\zeta(y)b) \{x \rightarrow c\} =$

$$(\zeta(z)(b \{y \rightarrow z\} \{x \rightarrow c\}))$$

For $z \notin FV(\zeta(y)b) \cup FV\{c\} \cup \{x\}$

2. $x \{x \rightarrow c\} = c$

3. $y \{x \rightarrow c\} = y$ for $y \neq x$

Object Substitution (continued)

$$4. \quad [l_i = \zeta(x_i)b_i \quad i \in 1..n] \{ x \rightarrow c \} =$$

$$[l_i = \zeta(x_i)b_i \{ x \rightarrow c \} \quad i \in 1..n]$$

$$5. \quad a.l \{ x \rightarrow c \} = (a \{ x \rightarrow c \}).l$$

$$6. \quad (a.l \Leftarrow \zeta(y)b) \{ x \rightarrow c \} =$$

$$(a \{ x \rightarrow c \}).l \Leftarrow (\zeta(y)b \{ x \rightarrow c \})$$

One-Step Reduction Relation

We write $a \Rightarrow b$ if for some

$o = [l_i = \zeta(x_i)b_i \quad i \in 1..n]$ and $j \in \{ 1, \dots, n \}$ either:

1. $a \equiv o.l_j$ and $b \equiv b_j \{ x_j \rightarrow o \}$, or

2. $a \equiv o.l_j \Leftarrow \zeta(y)c$ and

$b \equiv [l_j = \zeta(y)c , l_i = \zeta(x_i)b_i \quad i \in \{ 1..n \} - \{ j \}]$

1.2.5. Church Rosser Theorem

Let \Rightarrow^* be the reflexive, transitive closure of the binary relation \Rightarrow (i.e. many-step reduction)

Theorem

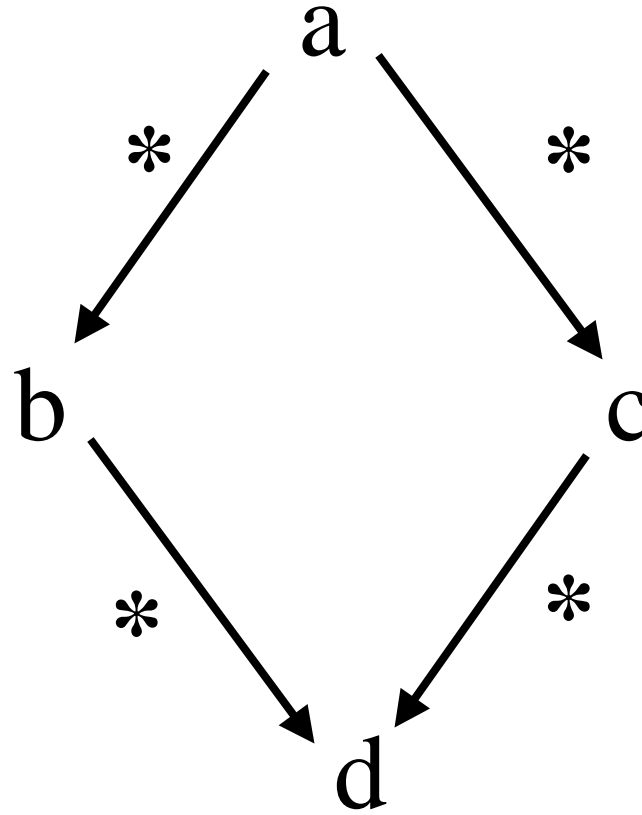
For any objects a, b, c if

$a \Rightarrow^* b$ and $a \Rightarrow^* c$ then

there exists an object d such that

$b \Rightarrow^* d$ and $c \Rightarrow^* d$

AKA. "The Diamond Property"



1.2.6. Reduction Strategy: Weak reduction

Let $v \equiv [l_i = \zeta(x_i)b_i \mid i \in \{1..n\}]$

$$\frac{}{v \supset^{wr} v} \quad \text{Red Object}$$

$a \supset^{wr} v', \quad b_j \{x_j \rightarrow v'\} \supset^{wr} v, \quad j \in \{1, \dots, n\}$

$$\frac{}{a.l_j \supset^{wr} v} \quad \text{Red Select}$$

where $v' \equiv [l'_i = \zeta(x'_i)b'_i \mid i \in \{1..m\}]$

$$\frac{a \supset^{\text{wr}} [l_i = \zeta(x_i)b_i \text{ } i \in \{1..n\}] , j \in \{1, \dots, n\}}{a.l_j \Leftarrow \zeta(y)b \supset^{\text{wr}} [l_j = \zeta(y)b , l_i = \zeta(x_i)b_i \text{ } i \in \{1..n\} - \{j\}]}$$

Red Update

Weak reduction \supset^{wr} for object calculus is analogous to weak reduction for lambda calculus.

1.2.7. An Interpreter based on Weak Reduction \supset^{wr}

We write a recursive algorithm `Execute(o)` on closed object calculus terms. If it converges then it produces a result, otherwise it returns the token *wrong*, representing a computation error.

1. $\text{Execute}([l_i = \zeta(x_i) b_i \quad i \in \{1..n\}]) =$
 $[l_i = \zeta(x_i) b_i \quad i \in \{1..n\}]$

2. $\text{Execute}(a.l_j) = \text{let } o = \text{Execute}(a) \text{ in}$
if o is of the form $[l_i = \zeta(x_i) b_i \{x_i\} \quad i \in \{1..n\}]$
with $j \in \{1, \dots, n\}$ then $\text{Execute}(b_j \{x_j \rightarrow o\})$
else *wrong*

Execute($a.l_j \Leftarrow \zeta(y) b$) =

let $o = \text{Execute}(a)$ in

if o is of the form $[l_i = \zeta(x_i) b_i \{x_i\} \quad i \in \{1..n\}]$

with $j \in \{1, \dots, n\}$

then $[l_j = \zeta(y)b, l_i = \zeta(x_i)b_i \quad i \in \{1..n\} - \{j\}]$

else *wrong*.

Clearly $o \supset^{\text{wr}} v$ iff

Execute(o) = v and $v \neq \textit{wrong}$

1.2.8. Theorems on Weak Reduction

Soundness Theorem

If $a \supset^{\text{wr}} v$ then $a \Rightarrow^* v$

Completeness Theorem

Let a be a closed term and v be a result

If $a \Rightarrow^* v$ then there exists v' such that

$a \supset^{\text{wr}} v'$

1.3. Encoding Lambda Calculus in Object Calculus

Since lambda calculus is a Turing complete functional programming language, it is natural to try to encode it in object calculus to learn more about objects.

First let's encode terms and then look at reduction.

1.3.1. Translating Lambda Terms

Define the translation map

$\langle . \rangle : \text{Lambda terms} \rightarrow \text{Object terms}$

recursively on lambda terms:

1. $\langle x \rangle = x$.
2. $\langle (b a) \rangle = \langle b \rangle . \langle a \rangle$

where $p.q = (p.\text{arg}:=q).\text{val}$

3. $\langle (\lambda x . b\{x\}) \rangle =$
[arg = $\zeta(x)$ x.arg,
val= $\zeta(x) \langle b(x) \rangle \{x \rightarrow x.arg\}$]

Idea is that $\langle (b a) \rangle$ first stores the argument $\langle a \rangle$ in the field arg inside of $\langle b \rangle$ and then invokes a method of $\langle b \rangle$ that can access the argument through self.

1.3.2. Translation Example

$$\langle ((\lambda x . x) y) \rangle =$$
$$([\text{arg}=\zeta(x) \text{ x.arg}, \text{val}=\zeta(x) \text{ x.arg}].\text{arg}:=y).\text{val}$$
$$= y$$

Notice how nested λ 's are translated to nested ζ 's. Although every method has a single self parameter, we can emulate functions with multiple parameters

1.3.3. Lambda Conversions

Does object reduction *preserve* lambda reduction?

Do we get the same results from lambda terms if we translate them into object terms and then execute the objects?

1.3.3.1. Preserving Beta reduction

Let $o =$

$[\text{arg} = \langle a \rangle, \text{val} = \zeta(x) \langle b(x) \rangle \{x \rightarrow x.\text{arg}\}]$

Then

$\langle (\lambda x . b\{x\}) (a) \rangle =$

$([\text{arg} = \zeta(x) x.\text{arg}, \text{val} = \zeta(x) \langle b\{x\} \rangle \{x \rightarrow$
 $x.\text{arg}\}].\text{arg} := \langle a \rangle).\text{val}$

$= o.\text{val}$

$= (\langle b\{x\} \rangle \{x \rightarrow x.\text{arg}\}).\{x \rightarrow o\}$

$$= \langle b\{x\} \rangle \{x \rightarrow o.\text{arg}\}$$

$$= \langle b\{x\} \rangle \{x \rightarrow \langle a \rangle\}$$

$$= \langle (b a) \rangle$$

So OC conversion preserves lambda calculus' beta reduction.

Exercise: show that OC conversion preserves lambda calculus alpha reduction.

1.3.3.2. Eta conversion, \equiv^η

Not every object models a lambda calculus term.

For example $x \equiv^\eta (\lambda y . x y)$ but

$$\langle x \rangle = x$$

$$\langle (\lambda y . x y) \rangle =$$

$$[\text{arg}=\zeta(y) \ y.\text{arg}, \text{val}=\zeta(y) \ \langle x y \rangle \{y \rightarrow y.\text{arg}\}] =$$

$$[\text{arg}=\zeta(y) \ y.\text{arg}, \text{val}=\zeta(y) \ (x.\text{arg} := y.\text{arg}).\text{val}]$$

So $\langle x \rangle$ and $\langle (\lambda y . x y) \rangle$ do not have the same normal form.

1.4. Object Calculus Programming

Let's see some simple examples of programming in the object calculus.

1. Moveable Geometric Points

Let $Origin_1 = [x = 0,$

$mv_x = \zeta(s) (\lambda dx . s.x := s.x + dx)]$

Let $\mathbf{Origin}_2 = [x = 0, y = 0,$
 $mv_x = \zeta(s) (\lambda dx . s.x := s.x + dx),$
 $mv_y = \zeta(s) (\lambda dy . s.x := s.y + dy)]$

Defining

$\mathbf{Unit}_2 = \mathbf{Origin}_2.mx_x(1).mv_y(1)$

then

$\mathbf{Unit}_2.x \Rightarrow^* 1$ and $\mathbf{Unit}_2.y \Rightarrow^* 1$

Since all operation possible on *Origin₁* are also possible on *Origin₂* then we would like in any type system for OC that *a moveable two-dimensional point* (such as *Origin₂*) *can be accepted in any context expecting a moveable one-dimensional point* (such as *Origin₁*).

Thus if *Origin₁ : 1-dim* and *Origin₂ : 2-dim* we would like *1-dim ≥ 2-dim*.

2. Backup methods.

How can we store self for later retrieval?

$o = [\text{retrieve} = \zeta(s_1) s_1, // \text{ initial object!}$

$\text{backup} = \zeta(s_2) s_2. \text{retrieve} \Leftarrow \zeta(s_1) s_2]$

Whenever backup method is called it stores a copy of the self current at invocation time into retrieve.

$o' = o.\text{backup} \Rightarrow^* [\text{retrieve} = \zeta(s_1) o, \dots]$

and $o'.\text{retrieve} \Rightarrow^* o$

3. Object-oriented natural numbers.

How can we model natural (positive) numbers as objects? Supposing the Booleans are available ..

```
zero = [   iszero = true,  
         pred = ζ(x) x,  
         Succ = ζ(x) ( x.iszero := false).pred:=x ]
```

Then zero answers "true" to iszero while all other number objects zero.succ, zero.succ.succ ... etc answer "false".

4. Pocket Calculator

The interface provides 4 operations:

enter, add, sub, equals.

calculator =

[arg = 0.0,

acc = 0.0,

enter = $\zeta(s) \lambda n . s.\text{arg} := n,$

add = $\zeta(s) (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow$

$\zeta(a) a.\text{acc} + a.\text{arg} ,$

sub = $\zeta(s)$ (s.acc := s.equals).equals \Leftarrow
 $\zeta(a)$ a.acc – a.arg ,
equals = $\zeta(s)$ s.arg]

Then

calculator.enter(5.0).equals \Rightarrow^* 5.0

calculator.enter(5.0).sub.enter(3.5).equals
 \Rightarrow^* 1.5

calculator.enter(5.0).add.add.equals \Rightarrow^* 15.0