CSC547: Type Systems for OO Languages

Untyped Imperative Calculi
Abadi and Cardelli, Chapter 10

Syntax

Fields

Functions

One-step reduction semantics

Operational semantics
Syntax

So far, all objects are *immutable*.

For real examples we want *mutable* objects such as:

\[\text{Point} \triangleq [\]
\text{getY}^+ : \text{int}, \text{getX}^+ : \text{int},
\text{move} : \text{int} \rightarrow \text{int} \rightarrow \text{void}
\]

\[\text{prototypePoint} \triangleq [\]
\text{x} = 0, \text{y} = 0,
\text{getX} = \varsigma(\text{this}) (\text{this}\.x),
\text{getY} = \varsigma(\text{this}) (\text{this}\.y),
\text{move} = \varsigma(\text{this}) \lambda(dX) \lambda(dY) (\]
\text{this}\.x := \text{this}\.x + dX;
\text{this}\.y := \text{this}\.y + dY;
\text{return}
\)
\]

\[\text{newPoint} \triangleq \lambda(x) \lambda(y) (\]
\text{clone (prototypePoint)}.\text{move} (x, y)
\)

What does \text{newPoint} (5) (37) do?

What would a mutable \text{Cell} example look like now?
Syntax

The new primitives we need are:

- *clone* $(a)$ makes a copy of $a$.
- *let* $x = a$ in $b$ evaluates $a$, binds the result to $x$ in $b$.

and we change the semantics of method update:

- $a.l \leftarrow M$ updates $a$'s method $l$.

Some shorthand:

- `void` $\triangleq [ ]$
- `return` $\triangleq [ ]$
- $a; b \triangleq \text{let unused} = a$ in $b$

This is the $\text{imp}_\varsigma$-calculus.

Why do we need *let* now?
Fields

We can code fields up as before (but being a bit careful about evaluation order!)

An object with fields:

\[
\begin{aligned}
&[ \\
&\quad l_1 = a_1, ..., l_m = a_m, \\
&\quad l'_1 = M_1, ..., l'_n = M_n \\
&\end{aligned}
\]

is shorthand for:

\[
\begin{aligned}
&\text{let } x_1 = a_1 \text{ in } ... \text{ let } x_m = a_m \text{ in } [ \\
&\quad l_1 = \varsigma(x)x_1, ..., l_m = \varsigma(x)x_m, \\
&\quad l'_1 = M_1, ..., l'_n = M_n \\
&\end{aligned}
\]

A field update \( a.l := b \) becomes:

\[
\begin{aligned}
&\text{let } x = a \text{ in } \\
&\text{let } y = b \text{ in } \\
&x.l \leftarrow \varsigma(z)y
\end{aligned}
\]

Why are we worrying about evaluation order?
Functions

We can do (almost) the same translation of functions into objects as before.

We code up $\lambda(x)b$ as:

\[
\begin{array}{l}
\text{arg} = \varsigma(this)(
  \text{this.arg,}
),
\text{val} = \varsigma(this)(
  \text{let } x = \text{this.arg in}
  b
) \\
\end{array}
\]

and function application $b(a)$ as:

\[\text{clone } (b).\text{arg} := a.\text{val}\]

(Abadi and Cardelli’s coding is ‘cleverer’ and allows assignment to function parameters.)

What does $(\lambda(x)(x+x))(5)$ do?

Why do we need the cloning?
One-step reduction semantics

Abadi and Cardelli provide an operational semantics using a stack and a heap (or store).

The resulting semantics is quite difficult to read!

Here is a simpler semantics which will hopefully be easier to use than Abadi and Cardelli’s....

To model Java-style mutable objects, we need a model of the heap (Abadi and Cardelli call this a store):

\[ H ::= p_1 \mapsto O_1, \ldots, p_n \mapsto O_n \]

for example:

fred \mapsto [ \text{first} = "Fred", \text{last} = "Flintstone" ],
wilma \mapsto [ \text{first} = "Wilma", \text{last} = "Flintstone" ],
both \mapsto [ \text{fst} = \text{fred}, \text{snd} = \text{wilma} ]

The semantics of objects can now read and write the heap.

In this heap, what does \texttt{clone(both.fst).first="Pebbles"} do?
One-step reduction semantics

Note that heaps can contain cycles:

\[
\begin{align*}
\text{bar} & \mapsto [ \ a = \text{foo} ] \\
\text{foo} & \mapsto [ \ b = \text{fold}(\text{Baz}, \text{bar}) ] \\
\end{align*}
\]

where \( \text{Baz} \triangleq \mu(B)[ \ a : [ \ b : B ] ] \)

We can generate cyclic heaps using assignment:

\[
\begin{align*}
\text{let } x &= [ \ a = \text{rubbish} ] \text{ in } \\
\text{let } y &= [ \ b = \text{fold}(\text{Baz}, x) ] \text{ in } \\
x.a & := y
\end{align*}
\]

How does this program execute?
One-step reduction semantics

The one-step reduction semantics is given by judgements $H \mid a \rightarrow H' \mid a'$.

This means ‘in heap $H$ program $a$ can do one step and become program $a'$ in heap $H'$.’

What are the reductions of $\text{clone}(\text{both}.\text{fst}).\text{first}:="\text{Pebbles}"$ in the heap:

- $\text{fred} \mapsto [ \text{first} = "\text{Fred}" , \text{last} = "\text{Flintstone}" ]$,
- $\text{wilma} \mapsto [ \text{first} = "\text{Wilma}" , \text{last} = "\text{Flintstone}" ]$,
- $\text{both} \mapsto [ \text{fst} = \text{fred} , \text{snd} = \text{wilma} ]$
One-step reduction semantics

(Step Object) \( (p \text{ fresh}) \)
\[
H | O \\
\rightarrow H, p \mapsto O | p
\]

(Step Select)
\[
H, p \mapsto O, H' | p.l_i \\
\rightarrow H, p \mapsto O, H' | b_i \{ x_i \rightarrow p \}
\]

(Step Update)
\[
H, p \mapsto O, H' | p.l_i \leftarrow M \\
\rightarrow H, p \mapsto O', H' | p
\]

(Step Clone) \( (q \text{ fresh}) \)
\[
H, p \mapsto O, H' | \text{clone } (p) \\
\rightarrow H, p \mapsto O, H', q \mapsto O | q
\]

(Step Let)
\[
H | \text{let } x = v \text{ in } a \\
\rightarrow H | a \{ x \rightarrow v \}
\]

where:
\[
O \equiv [l_1=M_1, ..., l_n=M_n] \\
O' \equiv [l_1=M_1, ..., l_i=M, ..., l_n=M_n] \\
M_i \equiv \varsigma(x_i)b_i
\]
One-step reduction semantics

An evaluation context $E[-]$ is given by the grammar:

$$E[-] ::= -
| E[-].l
| E[-].l \leftarrow M
| \text{clone } (E[-])
| \text{let } x = E[-] \text{ in } a$$

We allow reduction inside any evaluation context:

(Step Context)
If $H | a \rightarrow H' | b$
then $H | E[a] \rightarrow H' | E[b]$

What are the reductions of:

$$[x=0].x := 1.x$$

Why are we using evaluation contexts? Why not just let $E[-]$ be any context?
Example

A list object:

\[
\text{nil} \triangleq [
\begin{align*}
\text{isNil} &= \text{true}, \\
\text{hd} &= 0, \\
\text{tl} &= \varsigma (this)(this), \\
\text{cons} &= \varsigma (this) \lambda (h) ( \\
&\quad \text{let result} = \text{clone (this) in} \\
&\quad \text{result}.\text{isNil} := \text{false}; \\
&\quad \text{result}.\text{hd} := h; \\
&\quad \text{result}.\text{tl} := \text{this}; \\
&\quad \text{result}
\end{align*}
\]
\]

What are the reductions of \text{nil}.\text{cons}(1).\text{cons}(2).\text{hd}?
Example

A stack object:

```
empty ≜ [
  contents = nil,  
  isEmpty = λ (this) (this.contents.isNil),  
  push = λ (this) (x) (let oldContents = this.contents in let newContents = oldContents.cons (x) in this.contents := newContents; return),  
  pop = λ (this) (let oldContents = this.contents in let newContents = oldContents.tl in this.contents := newContents; oldContents.hd)
]
```

What are the reductions of:

```
let s = empty in s.push (1); s.push (2); s.pop
```
Operational semantics

Abadi and Cardelli’s semantics has a heap and a stack.

Their semantics is quite complex, since objects need to be stored on the heap together with a stack.

We shall stick with the one-step semantics since it’s simpler!