Closed and Open Recursion

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn
Email: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf`

July 2007

(Pick up the slides at . . . /`~ralf/talks.html#56`.)

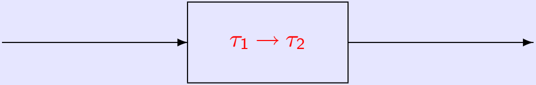
Trinity

- functional core,
- imperative core,
- object-oriented core,
- [module system].

Open recursion

Open recursion. *Another handy feature offered by most languages with objects and classes is the ability for one method body to invoke another method of the same object via a special variable called self or, in some langauges, this. The special behavior of self is that it is late-bound, allowing a method defined in one class to invoke another method that is defined later, in some subclass of the first.*

Functions



values

$$\textbf{fun}\ (x_1 : \tau_1) \Rightarrow e$$

introduction	elimination
$\textbf{fun}\ (x_1 : \tau_1) \Rightarrow e$	$e_2\ e_1$

Dynamic semantics

$$\frac{}{(\textbf{fun}\ (x_1 : \tau_1) \Rightarrow e) \Downarrow (\textbf{fun}\ (x_1 : \tau_1) \Rightarrow e)}$$

$$\frac{e_2 \Downarrow (\textbf{fun}\ (x_1 : \tau_1) \Rightarrow e) \qquad \textbf{let val}\ x_1 = e_1\ \textbf{in}\ e\ \textbf{end} \Downarrow \nu}{e_2\ e_1 \Downarrow \nu}$$

Recursive functions

$\textbf{rec fun}\ (n : \mathit{Nat}) \Rightarrow$
$\quad \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ \mathit{self}\ (n-1) * n$

☞ *self* refers to the function itself.

Dynamic semantics

$$\frac{}{(\textbf{rec fun}\ (x_1 : \tau_1) \Rightarrow e) \Downarrow (\textbf{fun}\ (x_1 : \tau_1) \Rightarrow e)\{\mathit{self} \mapsto \textbf{rec fun}\ (x_1 : \tau_1) \Rightarrow e\}}$$

☞ The recursive knot is tied at the earliest possible point in time: the introduction form.

Objects

$$\longrightarrow \boxed{\textbf{object}\ \mathrm{M}\ \textbf{end}} \longrightarrow$$

values

object μ **end**

introduction	elimination
object *m* **end**	*e.x*

Dynamic semantics

$$\frac{m \Downarrow \mu}{\textbf{object}\ m\ \textbf{end} \Downarrow \textbf{object}\ \mu\ \textbf{end}}$$

$$\frac{e \Downarrow \textbf{object}\ \mu\ \textbf{end} \qquad \mu(x) \Downarrow \nu}{e.x \Downarrow \nu}$$

Recursive objects

rec object
  method *factorial* $(n : Nat) =$
    if $n == 0$ **then** 1 **else** *self* .*factorial* $(n - 1) * n$
end

☞ *self* refers to the object itself.

Dynamic semantics

$$\frac{m \Downarrow \mu}{\textbf{rec object } m \textbf{ end} \Downarrow \textbf{rec object } \mu \textbf{ end}}$$

$$\frac{e \Downarrow \textbf{rec object } \mu \textbf{ end} \qquad \mu(x)\{\mathit{self} \mapsto \textbf{rec object } \mu \textbf{ end}\} \Downarrow \nu}{e.x \Downarrow \nu}$$

☞ The recursive knot is tied at the latest possible point in time: the elimination form.

Stocktaking

- Closed recursion: tie the recursive knot in the introduction form.
- Open recursion: tie the recursive knot in the elimination form.
- Does it make a difference?
- No! If there is only a single introduction and a single elimination form.
- Let's add an additional combining form, for instance, delegation.

Delegation

$$
\begin{array}{l}
\textbf{val}\ \mathit{math} = \\
\quad \textbf{rec object} \\
\quad\quad \textbf{method}\ \mathit{factorial}\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad\quad\quad \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ \mathit{self}.\mathit{factorial}'\ n * n \\
\quad\quad \textbf{method}\ \mathit{factorial}'\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad\quad\quad \mathit{self}.\mathit{factorial}\ (n - 1) \\
\quad \textbf{end} \\
\textbf{val}\ \mathit{math}' = \\
\quad \textbf{rec object} \\
\quad\quad \textbf{include}\ \mathit{math} \\
\quad\quad \textbf{method}\ \mathit{factorial}'\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad\quad\quad \mathit{self}.\mathit{factorial}\ (n - 1) + 1 \\
\quad \textbf{end}
\end{array}
$$

Thoughts

- Open recursion is not limited to objects.
- It is also useful for functions.
- Let's add an additional combining form for functions.

Open functions

val *fac-base* $=$
  fun open $(0 : \mathit{Nat}) \Rightarrow 1$

val *fac-step* $=$
  rec fun open $(n + 1 : \mathit{Nat}) \Rightarrow \mathit{self}\ \ n * (n + 1)$

val *factorial* $=$
  fac-base **or** *fac-step*

☞ An open function is a *partial function*.

☞ The combinator **or** combines two partial functions.

Abstract syntax

$e ::= \cdots$	open functions:
$\mid$ **fun open** r	non-recursive open function
$\mid$ **rec fun open** r	recursive open function
$\mid$ e_1 **or** e_2	alternation
$r ::= \epsilon$	empty rule
$\mid$ $p \Rightarrow e$	single rule
$\mid$ $r_1 \mid r_2$	sequences of rules

Dynamic semantics

$$\frac{}{(\textbf{rec fun open } r) \Downarrow (\textbf{rec fun open } r)}$$

$$\frac{e_1 \Downarrow (\textbf{rec fun open } r_1) \qquad e_2 \Downarrow (\textbf{rec fun open } r_2)}{e_1 \;\textbf{ or }\; e_2 \Downarrow \textbf{rec fun open } (r_1 \mid r_2)}$$

$$\frac{e_2 \Downarrow (\textbf{rec fun open } r) \qquad \textbf{case } e_1 \textbf{ of } r\{\mathit{self} \mapsto \textbf{rec fun open } r\}\textbf{end} \Downarrow \nu}{e_2\ e_1 \Downarrow \nu}$$

Application — generic programming

$$
\begin{aligned}
&\mathbf{val}\ \mathit{sum\text{-}nat} = \\
&\quad \mathbf{fun\ open}\langle \mathit{Nat} \rangle \Rightarrow \\
&\qquad \mathbf{fun}\ x \Rightarrow x \\
&\mathbf{val}\ \mathit{sum\text{-}pair} = \\
&\quad \mathbf{rec}\ \mathit{sum}\ \mathbf{fun\ open}\langle (a_1, a_2) \rangle \Rightarrow \\
&\qquad \mathbf{fun}\ (x_1, x_2) \Rightarrow \mathit{sum}\langle a_1 \rangle x_1 + \mathit{sum}\langle a_2 \rangle x_2
\end{aligned}
$$

☞ Instead of a **case** we have a **typecase**.

Conclusion

- Open functions are useful in conjunction with open data types.
- Vision: replacement for overloading and type classes.
- Open recursive modules?

Objects — example

$$
\begin{array}{l}
\textbf{val}\ \mathit{my\text{-}account} = \\
\quad \textbf{object} \\
\quad\quad \textbf{local} \\
\quad\quad\quad \textbf{val}\ \mathit{balance} = \textbf{ref}\ 0 \\
\quad\quad \textbf{in} \\
\quad\quad\quad \textbf{method}\ \mathit{deposit}\ (\mathit{amount} : \mathit{Nat}) = \\
\quad\quad\quad\quad \mathit{balance} := !\mathit{balance} + \mathit{amount} \\
\quad\quad\quad \textbf{method}\ \mathit{withdraw}\ (\mathit{amount} : \mathit{Nat}) = \\
\quad\quad\quad\quad \mathit{balance} := \mathit{sub}\ (!\mathit{balance}, \mathit{amount}) \\
\quad\quad\quad \textbf{method}\ \mathit{balance} = \\
\quad\quad\quad\quad !\ \mathit{balance} \\
\quad\quad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Objects — abstract syntax

$m \leftarrow$ Method	method declaration
$m ::= \epsilon$	empty declaration
$\mid$ **method** $x = e$	method definition
$\mid$ $m_1\ m_2$	sequence of declarations
$\mid$ **local** d **in** m **end**	local declaration

Objects — dynamic semantics

$$\frac{}{\epsilon \Downarrow \epsilon}$$

$$\frac{}{(\textbf{method}\ x = e) \Downarrow \{x \mapsto e\}}$$

$$\frac{m_1 \Downarrow \mu_1 \qquad m_2 \Downarrow \mu_2}{m_1\ m_2 \Downarrow \mu_1, \mu_2}$$

$$\frac{d \Downarrow \delta \qquad m\delta \Downarrow \mu}{\textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} \Downarrow \mu}$$

Finite maps

When X and Y are sets $X \to_{\text{fin}} Y$ denotes the set of finite maps from X to Y. The domain of a finite map φ is denoted $\mathit{dom}\ \varphi$.

- the singleton map is written $\{x \mapsto y\}$
  - $\mathit{dom}\{x \mapsto y\} = \{x\}$
  - $\{x \mapsto y\}(x) = y$
- when φ_1 and φ_2 are finite maps the map φ_1, φ_2 called φ_1 extended by φ_2 is the finite map with
  - $\mathit{dom}\ (\varphi_1, \varphi_2) = \mathit{dom}\ \varphi_1 \cup \mathit{dom}\ \varphi_2$
  - $(\varphi_1, \varphi_2)(x) = \begin{cases} \varphi_2(x) & \text{if } x \in \mathit{dom}\ \varphi_2 \\ \varphi_1(x) & \text{otherwise} \end{cases}$