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. 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 languages, 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

\[ \tau_1 \rightarrow \tau_2 \]

values

fun \( (x_1: \tau_1) \Rightarrow e \)

introduction

elimination

fun \( (x_1: \tau_1) \Rightarrow e \)

\[ e_2 e_1 \]
Dynamic semantics

\[
\begin{align*}
\text{(fun } x_1 : \tau_1 \Rightarrow e) & \Downarrow (\text{fun } x_1 : \tau_1 \Rightarrow e) \\
\text{e}_2 & \Downarrow (\text{fun } x_1 : \tau_1 \Rightarrow e) & \text{let val } x_1 = e_1 \text{ in } e \text{ end} & \Downarrow \nu \\
\text{e}_2 & \text{ e}_1 & \Downarrow \nu
\end{align*}
\]
Recursive functions

\[
\text{rec fun } (n : \text{Nat}) \Rightarrow \\
\quad \text{if } n == 0 \text{ then 1 else self } (n - 1) \ast n
\]

\( \text{self} \) refers to the function itself.
Dynamic semantics

\[ \text{(rec fun } x_1 : \tau_1 \Rightarrow e) \downarrow (\text{fun } x_1 : \tau_1 \Rightarrow e)\{ \text{self} \mapsto \text{rec fun } x_1 : \tau_1 \Rightarrow e \} \]

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

values

object $\mu$ end

introduction elimination

object $m$ end $e.x$
Dynamic semantics

\[
m \downarrow \mu
\]

\[
\text{object } m \text{ end} \downarrow \text{object } \mu \text{ end}
\]

\[
e \downarrow \text{object } \mu \text{ end} \quad \mu(x) \downarrow \nu
\]

\[
e.x \downarrow \nu
\]
Recursive objects

```plaintext
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

\[
\begin{align*}
  m \Downarrow \mu \\
  \text{rec object } m \text{ end} \Downarrow \text{rec object } \mu \text{ end}
\end{align*}
\]

\[
\begin{align*}
  e \Downarrow \text{rec object } \mu \text{ end} & \quad \mu(x)\{\text{self} \mapsto \text{rec object } \mu \text{ end}\} \Downarrow \nu \\
  e.x \Downarrow \nu
\end{align*}
\]

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

```ocaml
val math =
rec object
  method factorial (n : Nat) : Nat =
    if n == 0 then 1 else self.factorial' n * n
  method factorial' (n : Nat) : Nat =
    self.factorial (n - 1)
end

val math' =
rec object
  include math
  method factorial' (n : Nat) : Nat =
    self.factorial (n - 1) + 1
end
```
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 : Nat) ⇒ 1
val fac-step =
  rec fun open (n + 1 : Nat) ⇒ self * (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

\[ e ::= \cdots \]
\[ \mid \text{fun open } r \]
\[ \mid \text{rec fun open } r \]
\[ \mid e_1 \text{ or } e_2 \]

\[ r ::= \epsilon \]
\[ \mid p \Rightarrow e \]
\[ \mid r_1 \mid r_2 \]

**open functions:**
- non-recursive open function
- recursive open function
- alternation
- empty rule
- single rule
- sequences of rules
Dynamic semantics

\[
\begin{align*}
\text{(rec fun open } r\text{)} \Downarrow \text{(rec fun open } r\text{)} \\
\text{\hspace{1cm}}
\begin{array}{c}
e_1 \Downarrow \text{(rec fun open } r_1\text{)} \\
\text{or } e_2 \Downarrow \text{rec fun open } (r_1 \mid r_2)
\end{array}
\end{align*}
\]

\[
\begin{align*}
e_2 \Downarrow \text{(rec fun open } r\text{)} \\
\text{case } e_1 \text{ of } r\{ \text{self } \mapsto \text{rec fun open } r \} \text{end } \Downarrow \nu \\
\text{\hspace{1cm}}
\begin{array}{c}
e_2 \downarrow \nu \\
e_1 \downarrow \nu
\end{array}
\end{align*}
\]
Application — generic programming

val sum-nat =
  fun open⟨Nat⟩ ⇒
    fun x ⇒ x

val sum-pair =
  rec sum fun open⟨(a₁, a₂)⟩ ⇒
    fun (x₁, x₂) ⇒ sum⟨a₁⟩x₁ + sum⟨a₂⟩x₂

☞ 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

```
val my-account =
object
  local
    val balance = ref 0
  in
    method deposit (amount : Nat) =
      balance := !balance + amount
    method withdraw (amount : Nat) =
      balance := sub (!balance, amount)
    method balance =
      ! balance
  end
end
```
Objects — abstract syntax

\[
m \leftarrow \text{Method}
\]
\[
m ::= \epsilon
\]
\[
| \text{method } x = e
\]
\[
| m_1 \quad m_2
\]
\[
| \text{local } d \text{ in } m \text{ end}
\]

- **method declaration**
- empty declaration
- method definition
- sequence of declarations
- local declaration
Objects — dynamic semantics

\[ \varepsilon \Downarrow \varepsilon \]

\[ (\text{method } x = e) \Downarrow \{ x \mapsto e \} \]

\[ m_1 \Downarrow \mu_1 \quad m_2 \Downarrow \mu_2 \]

\[ m_1 \; m_2 \Downarrow \mu_1, \mu_2 \]

\[ d \Downarrow \delta \quad m\delta \Downarrow \mu \]

\[ \text{local } d \text{ in } m \text{ end} \Downarrow \mu \]
Finite maps

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

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