07—The Language rePL

CS 4215: Programming Language Implementation

Martin Henz

March 2, 2012

Generated on Friday 2 March, 2012, 12:57

Pairs as Functions

Constucting a pair using a conditional

```
let p = fun i ->
           if i=1 then 10 else 20 end
        end
in ...
end
```

Accessing Pairs

Accessing pairs by application

```
let p = fun i ->
           if i=1 then 10 else 20 end
        end
in ... (p 1) ... (p 2) ...
end
```

Constructing Pairs

```
let pair =
   fun x y ->
      fun i ->
         if i=1 then x else y end
      end
   end
in ...
   let p = (pair 10 20)
   in ...
   end
end
```

Motivation

Disadvantages of data structures as functions:

- difficult to distinguish functions from data structures
- definition of data structures with many components gives rise to large nested conditionals
- the only values that we can use to access data structures are integers
- inefficient, due to the function closures created, and linear execution of nested conditionals.

Motivation

With data structures, we will introduce another partial function. Thus we will need to take another look at error handling, giving rise to **exception handling**.

rePL Inherits From simPL

$$\frac{}{x} \qquad \frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{E_1 \qquad E_2}{p[E_1, E_2]} \text{ where } p \in \{\texttt{|}, \texttt{\&}, \texttt{+}, \texttt{-}, \texttt{*}\}$$

$$\frac{E}{p[E]} \text{ where } p \in \{/\}$$

rePL Inherits From simPL

$$\frac{E \quad E_1 \quad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}} \qquad \frac{E \quad E_1 \quad \cdots \quad E_n}{(E\ E_1 \cdots E_n)}$$

$$\frac{E}{\texttt{fun } x_1 \cdots x_n \texttt{ -> } E \texttt{ end}} \; x_1, \ldots, x_n \text{ pairwise distinct}$$

rePL Inherits From simPL

$$\frac{E}{\texttt{recfun}\ f\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}} \quad f, x_1, \cdots, x_n \text{ pairwise distinct}$$

$$\frac{E_1 \qquad \cdots \qquad E_n \qquad E}{\texttt{let}\ x_1 = E_1 \cdots\ x_n = E_n\ \texttt{in}\ E\ \texttt{end}}$$

Records in rePL

Records in rePL are bracket-enclosed sequences of property-value associations.

In rePL we can represent a pair containing 10 and 20 as a record of the form

```
[First:10, Second:20]
```

Accessing Records Using “Dot”

```
let p = [First:10, Second:20]
in p.First + p.Second
end
```

Nested Records

Records can appear inside of other records, as in the following record representing a color point on the screen.

```
[X:100, Y:200, Color:[Red:255, Green:127, Blue:0]]
```

Syntax of Records in rePL

$$\frac{E_1 \qquad E_n}{[q_1 : E_1, \ \ldots, q_n : E_n]} \qquad\qquad \frac{E}{E \,.\, q}$$

where $q, q_1, \ldots, q_n$ denote properties

More Record Operators

The operator

$$E \ \texttt{hasproperty} \ q$$

returns *true*, if the record resulting from E E has property q, and *false* otherwise.

Examples:

```
[Red:0, Blue:127, Green:255] hasproperty Green
```

```
[Red:0, Blue:127, Green:255] hasproperty Yellow
```

The Empty Record

The empty record `[]` does not have any properties.

The operator `empty` checks whether its argument is the empty record.

Examples:

```
empty []
```

```
empty [SomeProperty: 1]
```

Syntax

$$\frac{E}{p[E]} \quad \text{where } p \in \{\backslash, \texttt{empty}\}$$

Syntactic Sugar for rePL

We write E_1`::`E_2 as abbreviation for `[First:`E_1`, Second:`E_2`]`.

The operator `::` is right-associative.

We can write `10 :: 20 :: 30`
instead of `10 :: (20 :: 30)`.

Lists

A list is either empty—in which case it is represented by the empty record [] —or a pair, whose second component is also a list.

The elements of the list are the first components of the pairs that make up the list.

Example:

```
10 :: 20 :: 30 :: 40 :: []
```

Constructing Lists

```
let even = recfun even i counter done ->
              if counter=done then []
              else i :: (even i+2 counter+1 done)
              end
           end
in let evennumbers = fun n -> (even 2 0 n) end
   in ...
   end
end
```

The expression `(evennumbers 3)` returns the list `2 :: 4 :: 6 :: []`.

Length of Lists

The following function computes the length of a given list.

```
recfun length xs ->
   if empty xs then 0
   else 1+(length xs.Second)
   end
end
```

Mapping Lists

```
recfun map xs f -> if empty xs then []
                   else (f xs.First) :: (map xs.Second f)
                   end
end
```

Example:

```
(map 1 :: 2 :: 3 :: [] fun x -> x * x end)
```

returns `1 :: 4 :: 9 :: []`.

Folding Lists

```
recfun fold xs f start -> if empty xs then start
                          else (f xs.First
                                (fold xs.Second f start))
                          end
end
```

Example:

```
(fold 1 :: 4 :: 9 :: [] (fun x y -> x + y end) 0)
```

returns 14.

Motivation

Errors arise from

- Division by zero
- Invalid record access
- ...

Handling Exceptions

```
try (evaluate input)
catch e
with if e hasproperty DivisionByZero
     then (evaluate (readNewUserInput))
     else ..
     end
end
```

Syntax of Exception Handling

$$\frac{E_1 \qquad E_2}{\texttt{try } E_1 \texttt{ catch } x \texttt{ with } E_2 \texttt{ end}}$$

Built-in Exceptions

Division by zero leads to an exception of the form `[DivisonByZero:true]`.
Invalid record access leads to an exception of the form `[InvalidRecordAccess:true]`.

Examples:

```
4711 / 0
```

```
[].SomeProperty
```

Let the Programmer Throw Exceptions

$$\frac{E}{\texttt{throw}\ E\ \texttt{end}}$$

Example

```
if percentage > 100
then throw [PercentageExceeds100: true,
             PercentageValue: percentage]
     end
else ... end
```

The `percentageExceeds100` exception can then be caught by a surrounding expression and handled appropriately.

1. Data Structures
2. Exception Handling
3. Denotational Semantics of rePL
   - rePL0: simPL plus Records
   - rePL1: rePL0 plus Exceptions

Semantic Domains for rePL0

Sem. domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \{\bot\} + \mathbf{Fun} + \mathbf{Rec}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun} + \mathbf{Rec}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \leadsto \mathbf{DV}$	environments
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} \leadsto \mathbf{EV}$	function values
Rec	$\mathbf{Id} \leadsto \mathbf{DV}$	records

Rules for rePL0

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \quad \cdots \quad \Delta \Vdash E_n \rightarrowtail v_n}{\Delta \Vdash [q_1 : E_1 \, , \ \ldots \, , \ q_n : E_n] \rightarrowtail f} \text{where } f = \emptyset[q_1 \leftarrow v_1] \cdots [q_n \leftarrow v_n]$$

Rules for rePL0

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E\,.\,q \rightarrowtail v'}$$

where $v' = v(q)$

Rules for rePL0

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \texttt{empty}\ E \rightarrowtail \mathit{true}} \quad \text{if } \mathit{dom}(v) = \emptyset$$

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \texttt{empty}\ E \rightarrowtail \mathit{false}} \quad \text{if } \mathit{dom}(v) \neq \emptyset$$

Rules for rePL0

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E \ \texttt{hasproperty}\ q \rightarrowtail \mathit{true}} \quad \text{if } q \in \mathit{dom}(v)$$

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E \ \texttt{hasproperty}\ q \rightarrowtail \mathit{false}} \quad \text{if } q \notin \mathit{dom}(v)$$

Semantic Domains for rePL1

Sem. domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool** + **Int** + **Exc** + **Fun** + **Rec**	expressible values
DV	**Bool** + **Int** + **Fun** + **Rec**	denotable values
Id	alphanumeric string	identifiers
Env	**Id** $\rightsquigarrow$ **DV**	environments
Fun	**DV** $* \cdots *$ **DV** $\rightsquigarrow$ **EV**	function values
Rec	**Id** $\rightsquigarrow$ **DV**	records
Exc	**Rec**	exceptions

Rules for rePL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail e}{\Delta \Vdash E_1\text{+}E_2 \rightarrowtail e} \quad \text{if } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v \qquad \Delta \Vdash E_2 \rightarrowtail e}{\Delta \Vdash E_1\text{+}E_2 \rightarrowtail e} \quad \text{if } v \notin \textbf{Exc} \text{ and } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\text{+}E_2 \rightarrowtail v_1 + v_2} \quad \text{if } v_1, v_2 \notin \textbf{Exc}$$

Rules for rePL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail e}{\Delta \Vdash E_1 / E_2 \rightarrowtail e} \quad \text{if } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v \qquad \Delta \Vdash E_2 \rightarrowtail e}{\Delta \Vdash E_1 / E_2 \rightarrowtail e} \quad \text{if } v \notin \textbf{Exc} \text{ and } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 / E_2 \rightarrowtail v_1 / v_2} \quad \text{if } v_1, v_2 \notin \textbf{Exc} \text{ and } v_2 \neq 0$$

Rules for rePL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail 0}{\Delta \Vdash E_1 / E_2 \rightarrowtail e}$$

if $v_1 \not\in$ **Exc** and
where $e =$ `[DivisionByZero:true]`, and $e \in$ **Exc**

Rules for rePL1

$$\frac{\Delta \Vdash E \rightarrowtail e}{\Delta \Vdash E\,.\,q \rightarrowtail e} \quad \text{if } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E\,.\,q \rightarrowtail v'} \quad \text{if } q \in dom(v) \text{ and where } v' = v(q)$$

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E\,.\,q \rightarrowtail e}$$

if $q \notin dom(v)$ and where
$e = $ `[InvalidRecordAccess: true]`,
and $e \in \textbf{Exc}$

Rules for rePL1

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \texttt{throw } E \texttt{ end} \rightarrowtail e}$$

if $v \in \textbf{Rec} \cup \textbf{Exc}$ and
where $e = v$, $e \in \textbf{Exc}$

Overview of Next Lecture

Friday 9/3:

- Variants: pass-by-name and pass-by-need
- Implementing rePL and its variants