08—The Language rePL

CS 4215: Programming Language Implementation

Martin Henz

March 9, 2012

Generated on Friday 9 March, 2012, 09:02

1. Review of Data Structures in rePL
    - Records in rePL
    - Syntactic Sugar
2. Exceptions in rePL
3. Denotational Semantics of rePL
4. Pass-by-name and Pass-by-need
5. A Virtual Machine for rePL

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
```

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

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

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 :: []`.

1. Review of Data Structures in rePL
2. Exceptions in rePL
    - Motivation
    - Syntax of Exception Handling
    - Built-in Exceptions
    - Programmer-defined Exceptions
3. Denotational Semantics of rePL
4. Pass-by-name and Pass-by-need
5. A Virtual Machine for rePL

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. Review of Data Structures in rePL
2. Exceptions in rePL
3. Denotational Semantics of rePL
   - rePL0: simPL plus Records
   - rePL1: rePL0 plus Exceptions
4. Pass-by-name and Pass-by-need
5. A Virtual Machine for rePL

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	$\mathbf{Bool} + \mathbf{Int} + \{\perp\} + \mathbf{Fun} + \mathbf{Rec}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun} + \mathbf{Rec}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values
Rec	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	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

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

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

Rules for rePL0

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

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

Semantic Domains for rePL1

Sem. domain	Definition	Explanation
Bool	$\{$*true*, *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** $\leadsto$ **DV**	environments
Fun	**DV** $* \cdots *$ **DV** $\leadsto$ **EV**	function values
Rec	**Id** $\leadsto$ **DV**	records
Exc	**Rec**	exceptions

Rules for rePL1

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

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

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

Rules for rePL1

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

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

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

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 $v_1 \notin \textbf{Exc}$ and
where $e = \texttt{[DivisionByZero:true]}$, and $e \in \textbf{Exc}$

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 $e = v$, $e \in \textbf{Exc}$

1. Review of Data Structures in rePL
2. Exceptions in rePL
3. Denotational Semantics of rePL
4. Pass-by-name and Pass-by-need
    - Pass-by-name
    - Pass-by-need
    - Examples
5. A Virtual Machine for rePL

Pass-by-name

- Pass-by-name is an evaluation strategy where arguments of functions are passed immediately, instead of evaluating them.
- Pass-by-need is an optimization of pass-by-name.

Pass-by-name

- Pass argument expressions without evaluating them
- Evaluate them when needed
- What environment to use?

Thunks

Thunks are expressions together with an environment.

Semantic Domains

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** + **Thunk**	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
Thunk	rePL $*$ **Env**	thunks

Evaluation of Application

$$\frac{\Delta \Vdash E \rightarrowtail f}{\Delta \Vdash (E\ E_1\ \ldots\ E_n) \ \rightarrowtail f((E_1, \Delta), \ldots, (E_n, \Delta))}$$

Evaluation of Identifiers

$$\frac{\Delta' \Vdash E \rightarrowtail v}{\Delta \Vdash x \rightarrowtail v} \text{ if } \Delta(x) \text{ is thunk of the form } (E, \Delta').$$

$$\frac{}{\Delta \Vdash x \rightarrowtail \Delta(x)} \text{ if } \Delta(x) \text{ is not a thunk.}$$

Pass-by-need

- Observation: In pass-by-name, the same thunk could be evaluated multiple times
- Pass-by-need is an optimization of pass-by-name that avoids multiple evaluation of the same thunk.
- Pass-by-need enjoys popularity in functional programming (Haskell, Miranda).

Infinite Lists

```
let cons = fun x y -> x :: y end
in let makeints = recfun makeints i ->
                     (cons i (makeints i + 1))
                 end
   in  let allints = (makeints 0)
       in  allints.Second.Second.First
       end
end
```

Squaring Infinite Lists

```
let allints = (makeints 0)
in
   let allsquares = (map allints fun x -> x * x end)
   in  allsquares.Second.Second.First
   end
end
```

1. Review of Data Structures in rePL
2. Exceptions in rePL
3. Denotational Semantics of rePL
4. Pass-by-name and Pass-by-need
5. A Virtual Machine for rePL
   - Records
   - Exceptions
   - Implementing Records Efficiently

New Instructions for Record Construction

New instructions:

$$\frac{s}{\texttt{LDPS}\ q.s} \qquad \frac{s}{\texttt{RCDS}\ i.s}$$

where q is a property and i is an integer.

Compilation of Record Construction

$$\frac{E_1 \hookrightarrow s_1 \qquad \cdots \qquad E_n \hookrightarrow s_n}{[q_1 : E_1, \ldots, q_n : E_n] \hookrightarrow \texttt{LDPS}\ q_1.s_1.\ldots.\texttt{LDPS}\ q_n.s_n.\texttt{RCDS}\ n}$$

Execution of LDPS

$$\frac{s(pc) = \texttt{LDPS}\ q}{(os, pc, e, rs) \rightrightarrows_s (q.os, pc+1, e, rs)}$$

Execution of RCDS

Consider LDPS $q_1.s_1.\ldots.$LDPS $q_n.s_n.$RCDS n resulting from
$[q_1 : E_1, \ldots, q_n : E_n]$.
After LDPS $q_1.s_1.\ldots.$LDPS $q_n.s_n$, instruction RCDS n finds association list on operand stack.

$$\frac{s(pc) = \texttt{RCDS}\ n}{(v_n.q_n.\ldots.v_1.q_1.os, pc, e, rs) \rightrightarrows_s (\{(q_1, v_1), \ldots, (q_n, v_n)\}.os, pc + 1, e, rs)}$$

Operations on Records

Operations: `empty`, ".", `hasproperty`

New instructions:

$$\frac{s}{\texttt{EMPTY}.s} \qquad \frac{s}{\texttt{DOT}.s} \qquad \frac{s}{\texttt{HASP}.s}$$

Compilation of Record Operations

$$\frac{E \hookrightarrow s}{\texttt{empty}\ E \hookrightarrow s.\texttt{EMPTY}}$$

$$\frac{E \hookrightarrow s}{E\ \texttt{.}\ q \hookrightarrow s.\texttt{LDPS}\ q.\texttt{DOT}}$$

$$\frac{E \hookrightarrow s}{E\ \texttt{hasproperty}\ q \hookrightarrow s.\texttt{LDPS}\ q.\texttt{HASP}}$$

Executions of Record Operations

$$\frac{s(pc) = \texttt{EMPTY}}{(v.os, pc, e, rs) \rightrightarrows_s (true.os, pc+1, e, rs)} \text{ if } v = \emptyset$$

$$\frac{s(pc) = \texttt{EMPTY}}{(v.os, pc, e, rs) \rightrightarrows_s (false.os, pc+1, e, rs)} \text{ if } v \neq \emptyset$$

Executions of Record Operations

$$\frac{s(pc) = \texttt{DOT}}{(q.v.os, pc, e, rs) \rightrightarrows_s (v'.os, pc+1, e, rs)} \quad \text{if } v(q) = v'$$

$$\frac{s(pc) = \texttt{HASP}}{(q.v.os, pc, e, rs) \rightrightarrows_s (\mathit{true}.os, pc+1, e, rs)} \quad \text{if } \exists v_i.(q, v_i) \in v$$

$$\frac{s(pc) = \texttt{HASP}}{(q.v.os, pc, e, rs) \rightrightarrows_s (\mathit{false}.os, pc+1, e, rs)} \quad \text{if } \nexists v_i.(q, v_i) \in v$$

Built-in Exceptions

Division by zero and record access throw exceptions.
Idea: place instructions for raising these two exceptions at the end of the instruction sequence.

$$\frac{E \hookrightarrow s_1 \quad \texttt{[divisionByZero:true]} \hookrightarrow s_2 \quad \texttt{[invalidRecordAccess:true]} \hookrightarrow s_3}{E s_1.\texttt{DONE}.s_2.\texttt{THROW}.s_3.\texttt{THROW}}$$

beginning address of s_2: $addr_{divisionByZero}$
beginning address of s_3: $addr_{invalidRecordAccess}$

Primitive Operations Throwing Exceptions

$$\frac{s(pc) = \texttt{DIV}}{(0.i_1.os, pc, e, rs) \rightrightarrows_s (os, addr_{divisionByZero}, e, rs)}$$

$$\frac{s(pc) = \texttt{DOT}}{(q.v.os, pc, e, rs) \rightrightarrows_s (os, addr_{invalidRecordAccess}, e, rs)}$$

if $\not\exists v.(q, v') \in v$

Programmer-defined Exception Throws

$$\frac{E \hookrightarrow s}{\texttt{throw } E \texttt{ end} \;\hookrightarrow\; s.\texttt{THROW}}$$

Use Runtime Stack for Catching Exceptions

- Use runtime stack to keep track of the `catch...with...` part of `try` expressions
- Exception from `try` part will pop stackframes, until it finds the appropriate `catch...with...` part

Translation of `try` Statement

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{\begin{array}{c}\texttt{try}\ E_1\ \texttt{catch}\ x\ \texttt{with}\ E_2\ \texttt{end} \\ \hookrightarrow \\ (\texttt{TRY}\ x\ |s_1| + 3).s_1.\texttt{ENDTRY}.(\texttt{GOTOR}\ |s_2| + 1).s_2\end{array}}$$

Execution of TRY Instruction

$$\frac{s(pc) = \texttt{TRY}\ x\ i}{(os, pc, e, rs) \rightrightarrows_s (os, pc + 1, e, (\texttt{catch}, x, pc + i, os, e).rs)}$$

Execution of ENDTRY Instruction

$$\frac{s(pc) = \texttt{ENDTRY}}{(os, pc, e, (\texttt{catch}, x, pc', os, e).rs) \rightrightarrows_s (os, pc + 1, e, rs)}$$

Throwing of an Exception

$$\frac{s(pc) = \texttt{THROW}}{(os, pc, e, (pc', os', e').rs) \rightrightarrows_s (os, pc, e, rs)}$$

$$\frac{s(pc) = \texttt{THROW}}{(v.os, pc, e, (\texttt{catch}, x, pc', os', e').rs) \rightrightarrows_s (os', pc', e'[x \leftarrow v], rs)}$$

Problems with Records in RVM

- representation of records as a set of pairs is inefficient
- properties are strings

Observations

- Properties always appear literally
- Records are always constructed with `[...]`, which explicitly lists all properties

Representing Properties

- compiler constructs set Q of all properties in a given E
- compiler calculates a bijection *idp* between Q and $[0 \ldots |Q| - 1]$
- compiler replaces every occurrence of a property q in an instruction by $idp(q)$

Compilation of Record Operations (revisited)

$$\frac{E \hookrightarrow s}{E\ \texttt{.}\ q \hookrightarrow s.\texttt{LDCI}\ idp(q).\texttt{DOT}}$$

$$\frac{E \hookrightarrow s}{E\ \texttt{hasproperty}\ q \hookrightarrow s.\texttt{LDCI}\ idp(q).\texttt{HASP}}$$

Record Construction

- All records are constructed by `[...]`
- compiler calculates a bijection *idr* between the set R of all property sets of records and $[0 \ldots |R| - 1]$.
- associate with each property q of each record its alphabetical position in the corresponding property set: $p(idr(\{q_1, \ldots, q_n\}), idp(q))$, starting with 0. If a record with index m does not have a property with index n, we set $p(m, n) = -1$.

Example

- Let us say compiler assigns the number 13 to the set of properties $\{\mathtt{a}, \mathtt{b}\}$ (thus $idr(\{\mathtt{a}, \mathtt{b}\} = 13)$
- the number 55 to the property `a`
  $idp(\mathtt{a}) = 55$
- the number 77 to the property `b`
  $idp(\mathtt{b}) = 77$
- the position of property `a` in `[a:5 b:7]` is $p(13, 55) = 0$
- For any property identifier $n \neq 55, 77$: $p(13, n) = -1$.

Representing Records

Represent a record with properties $q_1, \ldots, q_n$ as a pair consisting of identifier $idr(\{q_1, \ldots, q_n\})$ and array that maps the alphabetical position of each q in the corresponding property list.

Example

In the example above, since $p(13, 55) = 0$ and $p(13, 77) = 1$, we can represent the record `[a:5 b:7]` by the pair $(13, [0 : 5, 1 : 7])$.

New Translation of Record Construction

$$\frac{E_1 \hookrightarrow s_1 \qquad \cdots \qquad E_n \hookrightarrow s_n}{\begin{array}{c}\texttt{[}q_1 : E_1\,,\,\ldots\,, q_n : E_n\texttt{]}\\ \hookrightarrow \\ \texttt{LDCI}\ \mathit{idp}(q_1).s_1.\ldots.\texttt{LDCI}\ \mathit{idp}(q_n).s_n.\texttt{RCD}\ n\ \mathit{idr}(\{q_1, \ldots, q_n\})\end{array}}$$

Efficient Records in RVM

- compiler passes the table p to RVM
- `RCD` constructs an array, whose indices corresponding to the record properties are given by p.

New Execution of Record Construction

$$\frac{s(pc) = \texttt{RCD}\ n\ m}{\begin{array}{c}(v_n.i_n.\ldots.v_1.i_1.os, pc, e, rs) \\ \rightrightarrows_s \\ ((m, \{(p(m, idp(q_1)), v_1), \ldots, (p(m, idp(q_n)), v_n)\}).os, pc + \\ 1, e, rs)\end{array}}$$

New Execution of Record Operations

$$\frac{s(pc) = \mathtt{DOT}}{(i.(m,a).os, pc, e, rs) \rightrightarrows_s (a(j).os, pc+1, e, rs)} \text{ if}$$

$$p(m,i) = j, j \geq 0$$

$$\frac{s(pc) = \mathtt{DOT}}{(i.(m,a).os, pc, e, rs) \rightrightarrows_s (os, addr_{invalidRecordAccess}, e, rs)} \text{ if}$$

$$p(m,i) = -1$$

New Execution of Record Operations

$$\frac{s(pc) = \texttt{HASP}}{(i.(m,a).os, pc, e, rs) \rightrightarrows_s (true.os, pc+1, e, rs)} \text{ if } p(m,i) \geq 0$$

$$\frac{s(pc) = \texttt{HASP}}{(i.(m,a).os, pc, e, rs) \rightrightarrows_s (false.os, pc+1, e, rs)} \text{ if}$$

$p(m, i) = -1$

Summary of Record Implementation

Constant time record access achieved by:

- representing properties by integers using *idp*
- mapping record property sets to integers using *idr*
- record access through arrays using lookup table *p*

Overview of Next Lecture

- Imperative Programming: The language imPL