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 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”

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

\[
\begin{align*}
E_1 & \quad E_n \\
\hline
[q_1 : E_1, \ldots, q_n : E_n] \\
E & \quad E \cdot q
\end{align*}
\]

where \( q, q_1, \ldots, q_n \) denote properties
The operator

\[ E \text{ hasproperty } q \]

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 [] does not have any properties.

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

Examples:

empty []

empty [SomeProperty: 1]
Syntax

\[
E \quad \quad \text{where } p \in \{\emptyset, \text{empty}\}
\]

\[
p[E]
\]
We write $E_1 :: E_2$ as abbreviation for $[\text{First}: E_1, \text{Second}: E_2]$.

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 \([\text{[]}\)—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 :: [].

The following function computes the length of a given list.

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

```plaintext
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
Errors arise from

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

```plaintext
try (evaluate input)
catch e
with if e hasproperty DivisionByZero
    then (evaluate (readNewUserInput))
    else ..
end
end
```
Syntax of Exception Handling

\[
\begin{align*}
E_1 & \quad E_2 \\
\hline
\text{try } E_1 \text{ catch } x \text{ with } E_2 \text{ end}
\end{align*}
\]
Built-in Exceptions

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

Examples:

4711 / 0

[].SomeProperty
Let the Programmer Throw Exceptions

\[
E \quad \text{throw } E \text{ end}
\]
Example

```rePL
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.
Review of Data Structures in rePL

Exceptions in rePL

Denotational Semantics of rePL
- rePL0: simPL plus Records
- rePL1: rePL0 plus Exceptions

Pass-by-name and Pass-by-need

A Virtual Machine for rePL
## Semantic Domains for rePL0

<table>
<thead>
<tr>
<th>Sem. domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>{..., −2, −1, 0, 1, 2, ...}</td>
<td>ring of integers</td>
</tr>
<tr>
<td><strong>EV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + {⊥} + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>expressible values</td>
</tr>
<tr>
<td><strong>DV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>denotable values</td>
</tr>
<tr>
<td><strong>Id</strong></td>
<td>alphanumeric string</td>
<td>identifiers</td>
</tr>
<tr>
<td><strong>Env</strong></td>
<td><strong>Id</strong> (\rightsquigarrow) <strong>DV</strong></td>
<td>environments</td>
</tr>
<tr>
<td><strong>Fun</strong></td>
<td><strong>DV</strong> (<em>\cdots)</em> <strong>DV</strong> (\rightsquigarrow) <strong>EV</strong></td>
<td>function values</td>
</tr>
<tr>
<td><strong>Rec</strong></td>
<td><strong>Id</strong> (\rightsquigarrow) <strong>DV</strong></td>
<td>records</td>
</tr>
</tbody>
</table>
Rules for rePL0

\[
\Delta \vdash E_1 \equiv v_1 \quad \cdots \quad \Delta \vdash E_n \equiv v_n \quad \text{where } f = \emptyset[q_1 \leftarrow v_1] \\
\Delta \vdash [q_1 : E_1, \ldots, q_n : E_n] \rightarrow f \quad \cdots [q_n \leftarrow v_n]
\]
Rules for rePL0

\[ \Delta \models E \Rightarrow v \]

\[ \Delta \models E.q \Rightarrow v' \]

where \( v' = v(q) \)
Rules for rePL0

\[ \Delta \models E \Rightarrow v \]

\[ \Delta \models \text{empty } E \Rightarrow true \quad \text{if } \text{dom}(v) = \emptyset \]

\[ \Delta \models E \Rightarrow v \quad \text{if } \text{dom}(v) \neq \emptyset \]

\[ \Delta \models \text{empty } E \Rightarrow false \]
Rules for rePL0

\[ \Delta \vdash E \mapsto v \]

if \( q \in \text{dom}(v) \)

\[ \Delta \vdash E \text{ hasproperty } q \mapsto \text{true} \]

\[ \Delta \vdash E \mapsto v \]

if \( q \notin \text{dom}(v) \)

\[ \Delta \vdash E \text{ hasproperty } q \mapsto \text{false} \]
## Semantic Domains for rePL1

<table>
<thead>
<tr>
<th>Sem. domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>{\ldots, -2, -1, 0, 1, 2, \ldots}</td>
<td>ring of integers</td>
</tr>
<tr>
<td><strong>EV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Exc</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>expressible values</td>
</tr>
<tr>
<td><strong>DV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>denotable values</td>
</tr>
<tr>
<td><strong>Id</strong></td>
<td>alphanumeric string</td>
<td>identifiers</td>
</tr>
<tr>
<td><strong>Env</strong></td>
<td><strong>Id</strong> ⇝ <strong>DV</strong></td>
<td>environments</td>
</tr>
<tr>
<td><strong>Fun</strong></td>
<td><strong>DV</strong> * * * <strong>DV</strong> ⇝ <strong>EV</strong></td>
<td>function values</td>
</tr>
<tr>
<td><strong>Rec</strong></td>
<td><strong>Id</strong> ⇝ <strong>DV</strong></td>
<td>records</td>
</tr>
<tr>
<td><strong>Exc</strong></td>
<td><strong>Rec</strong></td>
<td>exceptions</td>
</tr>
</tbody>
</table>
Rules for rePL1

\[ \Delta \models E_1 \rightarrow e \]
\hline
\[ \Delta \models E_1 + E_2 \rightarrow e \quad \text{if } e \in \text{Exc} \]
\hline
\[ \Delta \models E_1 \rightarrow v \quad \Delta \models E_2 \rightarrow e \]
\hline
\[ \Delta \models E_1 + E_2 \rightarrow e \quad \text{if } v \notin \text{Exc} \text{ and } e \in \text{Exc} \]
\hline
\[ \Delta \models E_1 \rightarrow v_1 \quad \Delta \models E_2 \rightarrow v_2 \]
\hline
\[ \Delta \models E_1 + E_2 \rightarrow v_1 + v_2 \quad \text{if } v_1, v_2 \notin \text{Exc} \]
Rules for rePL1

\[
\begin{align*}
\Delta \vdash E_1 \rightsquigarrow e & \quad \text{if } e \in \text{Exc} \\
\Delta \vdash E_1/E_2 \rightsquigarrow e & \\
\Delta \vdash E_1 \rightsquigarrow v & \quad \Delta \vdash E_2 \rightsquigarrow e & \quad \text{if } v \not\in \text{Exc} \text{ and } e \in \text{Exc} \\
\Delta \vdash E_1/E_2 \rightsquigarrow e & \\
\Delta \vdash E_1 \rightsquigarrow v_1 & \quad \Delta \vdash E_2 \rightsquigarrow v_2 & \quad \text{if } v_1, v_2 \not\in \text{Exc} \text{ and } v_2 \neq 0 \\
\Delta \vdash E_1/E_2 \rightsquigarrow v_1/v_2 & 
\end{align*}
\]
Rules for rePL1

\[
\begin{align*}
\Delta \vdash E_1 \mapsto v_1 & \quad \Delta \vdash E_2 \mapsto 0 \\
\hline
\Delta \vdash E_1/E_2 \mapsto e
\end{align*}
\]

if \( v_1 \notin \text{Exc} \) and
where \( e = \text{[DivisionByZero: true]} \), and \( e \in \text{Exc} \)
Rules for rePL1

\[ \Delta \vdash E \mapsto e \]
\[ \Delta \vdash E \cdot q \mapsto e \]  
if \( e \in \text{Exc} \)

\[ \Delta \vdash E \mapsto v \]
\[ \Delta \vdash E \cdot q \mapsto v' \]   
if \( q \in \text{dom}(v) \) and where \( v' = v(q) \)

\[ \Delta \vdash E \mapsto v \]
\[ \Delta \vdash E \cdot q \mapsto e \]    
if \( q \not\in \text{dom}(v) \) and where
\[ e = [\text{InvalidRecordAccess: true}] \],
and \( e \in \text{Exc} \)
Rules for rePL1

\[ \Delta \vdash E \rightarrow v \]

\[ \Delta \vdash \text{throw } E \text{ end } \rightarrow e \]

if \( v \in \text{Rec} \cup \text{Exc} \) and

where \( e = v, \ e \in \text{Exc} \)
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 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 are expressions together with an environment.
## Semantic Domains

<table>
<thead>
<tr>
<th>Sem. domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>{\ldots, -2, -1, 0, 1, 2, \ldots}</td>
<td>ring of integers</td>
</tr>
<tr>
<td><strong>EV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Exc</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>expressible values</td>
</tr>
<tr>
<td><strong>DV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Fun</strong> + <strong>Rec</strong> + <strong>Thunk</strong></td>
<td>denotable values</td>
</tr>
<tr>
<td><strong>Id</strong></td>
<td>alphanumeric string</td>
<td>identifiers</td>
</tr>
<tr>
<td><strong>Env</strong></td>
<td><strong>Id</strong> ⇝ <strong>DV</strong></td>
<td>environments</td>
</tr>
<tr>
<td><strong>Fun</strong></td>
<td><strong>DV</strong> * * * <strong>DV</strong> ⇝ <strong>EV</strong></td>
<td>function values</td>
</tr>
<tr>
<td><strong>Rec</strong></td>
<td><strong>Id</strong> ⇝ <strong>DV</strong></td>
<td>records</td>
</tr>
<tr>
<td><strong>Exc</strong></td>
<td><strong>Rec</strong></td>
<td>exceptions</td>
</tr>
<tr>
<td><strong>Thunk</strong></td>
<td>rePL * <strong>Env</strong></td>
<td>thunks</td>
</tr>
</tbody>
</table>
Evaluation of Application

\[ \Delta \models E \mapsto f \]

\[ \Delta \models (E \ E_1 \ldots \ E_n) \mapsto f((E_1, \Delta), \ldots, (E_n, \Delta)) \]
Evaluation of Identifiers

\[ \Delta' \models E \rightsquigarrow v \]

if \( \Delta(x) \) is thunk of the form \((E, \Delta')\).

\[ \Delta \models x \rightsquigarrow v \]

\[ \Delta' \models x \mapsto \Delta(x) \]

if \( \Delta(x) \) 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).
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
let allints = (makeints 0)
in
  let allsquares = (map allints fun x -> x * x end)
in  allsquares.Second.Second.First
end
end
Review of Data Structures in rePL

Exceptions in rePL

Denotational Semantics of rePL

Pass-by-name and Pass-by-need

A Virtual Machine for rePL

- Records
- Exceptions
- Implementing Records Efficiently
New Instructions for Record Construction

New instructions:

\[
\begin{align*}
&\text{LDPS } q.s \\
&\text{RCDS } i.s
\end{align*}
\]

where \( q \) is a property and \( i \) is an integer.
Compilation of Record Construction

\[
E_1 \leftrightarrow s_1 \quad \ldots \quad E_n \leftrightarrow s_n
\]

\[
[q_1:E_1, \ldots, q_n:E_n] \leftrightarrow \text{LDPS } q_1.s_1.\ldots.\text{LDPS } q_n.s_n.\text{RCDS } n
\]
Execution of LDPS

\[ s(pc) = \text{LDPS} \ q \]

\[
(\text{os}, pc, e, rs) \xrightarrow{s} (q.os, pc + 1, e, rs)
\]
Consider LDPS \( q_1.s_1 \ldots \ldots \)LDPS \( q_n.s_n \).RCDS \( n \) resulting from \([q_1:E_1, \ldots, q_n:E_n]\). After LDPS \( q_1.s_1 \ldots \ldots \)LDPS \( q_n.s_n \), instruction RCDS \( n \) finds association list on operand stack.

\[
s(pc) = \text{RCDS} \ n
\]

\[
(v_n.q_n. \ldots .v_1.q_1.os, pc, e, rs) \Rightarrow_s (\{(q_1, v_1), \ldots , (q_n, v_n)\}.os, pc + 1, e, rs)
\]
Operations on Records

Operations: empty, ".", hasproperty
New instructions:

\[
\begin{align*}
\text{EMPTY}.s \\
\text{DOT}.s \\
\text{HASP}.s
\end{align*}
\]
Compilation of Record Operations

\[
\begin{align*}
E & \leftrightarrow s \\
\text{empty } E & \leftrightarrow s.\text{EMPTY} \\
E & \leftrightarrow s \\
E \text{ hasproperty } q & \leftrightarrow s.\text{LDPS } q.\text{HASP}
\end{align*}
\]
Executions of Record Operations

\[ s(pc) = \text{EMPTY} \]

\[ \text{if } v = \emptyset \]

\[ (v.os, pc, e, rs) \Rightarrow_s (true.os, pc + 1, e, rs) \]

\[ s(pc) = \text{EMPTY} \]

\[ \text{if } v \neq \emptyset \]

\[ (v.os, pc, e, rs) \Rightarrow_s (false.os, pc + 1, e, rs) \]
Executions of Record Operations

\[
\begin{align*}
  s(pc) &= \text{DOT} \\
  (q.v.os, pc, e, rs) &\Rightarrow_s (v'.os, pc + 1, e, rs) \\
  \quad \text{if } v(q) = v' \\
  s(pc) &= \text{HASP} \\
  (q.v.os, pc, e, rs) &\Rightarrow_s (true.os, pc + 1, e, rs) \\
  \quad \text{if } \exists v_i. (q, v_i) \in v \\
  s(pc) &= \text{HASP} \\
  (q.v.os, pc, e, rs) &\Rightarrow_s (false.os, pc + 1, e, rs) \\
  \quad \text{if } \neg \exists v_i. (q, v_i) \in v
\end{align*}
\]
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.

\[ E \rightarrow s_1 \]
\[ \text{[divisionByZero:} \text{true]} \rightarrow s_2 \]
\[ \text{[invalidRecordAccess:} \text{true]} \rightarrow s_3 \]

\[ Es_1.\text{DONE}.s_2.\text{THROW}.s_3.\text{THROW} \]

beginning address of \(s_2\): \(addr_{\text{divisionByZero}}\)
beginning address of \(s_3\): \(addr_{\text{invalidRecordAccess}}\)
Primitive Operations Throwing Exceptions

\[ s(pc) = \text{DIV} \]

\[ (0.i_1.os, \text{pc}, e, rs) \Rightarrow_s (os, addr_{\text{divisionByZero}}, e, rs) \]

\[ s(pc) = \text{DOT} \]

\[ (q.v.os, \text{pc}, e, rs) \Rightarrow_s (os, addr_{\text{invalidRecordAccess}}, e, rs) \]

if \( \forall v. (q, v') \in v \)
Programmer-defined Exception Throws

\[ E \leftarrow s \]

\[ \text{throw } E \text{ end } \leftarrow s.\text{THROW} \]
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

\[ E_1 \xrightarrow{\cdot} s_1 \quad E_2 \xrightarrow{\cdot} s_2 \]

\[
\text{try } E_1 \text{ catch } x \text{ with } E_2 \text{ end } \xrightarrow{\cdot} (\text{TRY } x |s_1| + 3).s_1.\text{ENDTRY}.(\text{GOTOR } |s_2| + 1).s_2
\]
Execution of TRY Instruction

\[ s(pc) = \text{TRY} \times i \]

\[
(os, pc, e, rs) \Rightarrow_s (os, pc + 1, e, (\text{catch}, x, pc + i, os, e).rs)
\]
Execution of ENDTRY Instruction

\[ s(pc) = \text{ENDTRY} \]

\[ (os, pc, e, (\text{catch}, x, pc', os, e).rs) \xrightarrow{s} (os, pc + 1, e, rs) \]
Throwing of an Exception

\[ s(pc) = \textsc{throw} \]

\[ (os, pc, e, (pc', os', e').rs) \Rightarrow_s (os, pc, e, rs) \]

\[ s(pc) = \textsc{throw} \]

\[ (v.os, pc, e, (\texttt{catch}, x, pc', os', e').rs) \Rightarrow_s (os', pc', e'[x \leftarrow v], rs) \]
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)

\[
E \rightarrow s
\]

\[
\frac{E . q \rightarrow s.\text{LDCI } idp(q).\text{DOT}}{E \rightarrow s}
\]

\[
E \text{ hasproperty } q \rightarrow s.\text{LDCI } idp(q).\text{HASP}
\]
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 \{a, b\} (thus \(idr(\{a, b\} = 13\))
- the number 55 to the property a
  \(idp(a) = 55\)
- the number 77 to the property b
  \(idp(b) = 77\)
- 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.
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

\[ E_1 \mapsto s_1 \quad \cdots \quad E_n \mapsto s_n \]

\[ [q_1 : E_1, \ldots, q_n : E_n] \mapsto \text{LDCI } idp(q_1).s_1 \ldots \text{LDCI } idp(q_n).s_n.\text{RCD } n \text{ idr} \{q_1, \ldots, q_n\} \]
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

\[ s(pc) = \text{RCD} \ n \ m \]

\[
(v_n.i_n \ldots .v_1.i_1.os, pc, e, rs) \Rightarrow_s
((m, \{(p(m, idp(q_1)), v_1), \ldots, (p(m, idp(q_n)), v_n)\}).os, pc + 1, e, rs)
\]
New Execution of Record Operations

\[
s(pc) = \text{DOT}
\]

if

\[
(i.(m,a).os, pc, e, rs) \Rightarrow_s (a(j).os, pc + 1, e, rs)
\]

\[p(m, i) = j, j \geq 0\]

\[
s(pc) = \text{DOT}
\]

if

\[
(i.(m,a).os, pc, e, rs) \Rightarrow_s (os, addr_{\text{invalidRecordAccess}}, e, rs)
\]

\[p(m, i) = -1\]
New Execution of Record Operations

\[
s(pc) = \text{HASP} \\
\text{if } p(m, i) \geq 0 \\
(i.(m, a).os, pc, e, rs) \Rightarrow_s (true.os, pc + 1, e, rs)
\]

\[
s(pc) = \text{HASP} \\
\text{if } (i.(m, a).os, pc, e, rs) \Rightarrow_s (false.os, pc + 1, e, rs)
\]

\[
p(m, i) = -1
\]
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