CS4215—Programming Language Implementation

Martin Henz

Friday 2 March, 2012

Chapter 10

rePL: Adding Data Abstractions

The language simPL has one important deficiency; it lacks the possibility to directly form complex data structures. We shall see in the first section that functions can express data structures. However, this is syntactically complicated and inefficient in practice. Therefore, we extend simPL in Section 10.2 by records, which permit the programmer to directly define complex data structures. Section 10.3 gives examples of how to use this extension in practice.

To access record components, we introduce a new partial semantic function, in addition to division. In Section 10.4, we take a fresh look at how to handle exceptional situations arising from partial semantic functions, and how the programmer can control such situations. Section 10.5 covers an interpreter for the rePL language, along the lines of the simPL interpreter.

We explore alternative evaluation strategies in the context of rePL in Section 10.6. Section 10.7 shows a few programming techniques that become possible with one of these strategies.

Finally, we extend the virtual machine for simPL by instructions that allow us to cover rePL's data structures.

10.1 Data Structures in simPL

Data structures in simPL have to be expressed using functions. For example, we can represent a pair containing the numbers 10 and 20 by the function

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

In the body of the `let`, we can access the first component of the pair `p` by applying `p` to the integer 1, and the second component by applying it to the integer 2.

```
let ...
in ... (p 1) ... (p 2) ...
end
```

To construct such pairs, we can define a function

```
let pair =
   fun x y ->
      fun i ->
         if i=1 then x else y
         end
      end
   end
in ...
end
```

In the body of this `let`, we can now construct pairs as in

```
let p = (pair 10 20) in ... end
```

Thus in principle it is possible to use functions for expressing data structures. This approach has several disadvantages:

- It is difficult to distinguish functions from data structures.
- The 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, which makes it hard for the programmer to manipulate complex data structures.
- The approach is inefficient, due to the function closures created and due to the linear execution of nested conditionals.

Therefore, we shall introduce data structures directly in an extension of simPL called rePL (record Programming Language).

10.2 Data Structures in rePL

The following rules for rePL programs result from the rules for simPL by removing type declarations.

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

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

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

For primitive operators, we use the infix and prefix notation for operators with the associativity and operator precedences described in Section 5.2.

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

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

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

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

We introduce data structures in form of records. Records are bracket-enclosed sequences of property-value associations. In rePL, properties are distinguished from identifiers by starting with a capital letter, whereas identifiers in rePL must begin with a lower-case letter. For example, in rePL we can represent a pair containing 10 and 20 as a record of the form

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

Such records can be accessed by a new operator ".", as in

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

Of course, 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]]
```

Thus, we need the following syntactic rules accommodate record construction and access in rePL.

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

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

In order to check if a record has a given property, we add an operator E `hasproperty` q, which returns *true*, if the record that results from evaluating E has the property q, and *false* otherwise.

One record is distinguished from all other records in that it has no properties. This record—denoted by `[]`—is so important that we introduce a unary primitive operator `empty` that tests whether its argument is `[]`. The expression `empty []` returns *true*, whereas `empty [SomeProperty:1]` returns *false*.

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

As syntactic convenience, we introduce an abbreviation for records that represent pairs. We write $E_1\texttt{::}E_2$ 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)`.

A common pattern of usage of records is to construct 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. A list containing the numbers 10, 20, 30, and 40 looks like this in rePL:

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

10.3 Examples

The following function constructs a list with the first `n` even natural numbers.

```
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.

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

Another example is the function `map` that applies a given function to all elements of a list.

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

To square every element of the list `1 :: 2 :: 3 :: []`, we apply `map` to the list and the square function

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

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

An important function for list programming is the `fold` function that folds a given list together, using a given function at every step.

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

This function can be used for iteration over lists. For example, in order to sum up all elements of the list `1 :: 4 :: 9 :: []`, `fold` can be applied as follows.

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

which returns 14.

A final syntactic convienience provides a notation for strings in rePL. Strings are lists of characters, where characters are represented by their Latin-1 encoding, defined by ISO 8859-1, see [cJS87]. Using this convention, we can write the string `"abc"` as an abbreviation for the list `97 :: 98 :: 99 :: []`.

10.4 Exceptions

In Section 7.6, we saw that an error value can adequately represent the only exceptional behavior in the interpreter of simPL, namely the partial function division, which is undefined for the argument 0. In rePL, we added another partial function, namely record access. Any attempt to access a property of a record

that does not have that property is undefined. We would like to distinguish between division by zero and such an invalid record access. The programmer should be able to take appropriate corrective action, when either one of these two exceptional situations arise.

What we need is a way to wrap an expression E_1 such that when an exception occurs in E_1, its execution is terminated, and another expression E_2 is executed instead, to handle the exception. For example, the following expression tries to evaluate a record `input`, which represents the input of the user of a calculator. If an exception occurs, the user is asked for a new input.

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

Note that in the expression following `with`, the variable `e` declared by `catch` refers to a record that describes the exception. The exception record resulting from a division by zero has the property `DivisionByZero`. Other properties of the record may indicate the first argument of the division, and the filename and line number division in the source code so that the programmer can locate the error quickly.

The syntax of `try` expressions is given by the following rule.

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

Invalid record access as in `[].SomeProperty` leads to an exception with the property `InvalidRecordAccess`.

With the ability of "catching" exceptions in place, it seems natural that the programmer should be able to create her own exceptions, and catch them. The concept is called "throwing" an exception, and supported by rePL as follows.

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

By throwing an exception, the programmer can define by herself what situations are considered exceptions in her programs. For example, the following program fragment defines as exceptional any situation where a given `percentage` value exceeds 100.

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

```
   end
else ... end
```

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

10.5 An Interpreter for rePL

As usual, we proceed in steps. The first step, rePL0, consists of simPL (without types) extended by records and the corresponding operators ".", `empty`, and `hasproperty`. The second step rePL1 adds the `try`...`catch`... and `throw` expressions.

10.5.1 An Interpreter for rePL0

To support records, we extend our semantic domains as follows.

Sem. dom.	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	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values
Rec	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	records

Note that the functions that represent records look exactly like environments, but are going to be used differently.

We need to add rules for record construction and the primitive operations on records to the semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$. The rule for record construction uses the usual extension formalism to construct a **Rec** value.

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

Record access applies the record value to the given property.

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E\texttt{.}q \rightarrowtail v'} \quad \text{where } v' = v(q)$$

The `empty` operator checks if the domain of the given record value is empty.

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

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

The operator `hasproperty` similarly checks if a given property is in the domain of the given record.

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

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

10.5.2 An Interpreter for rePL1

Similar to the interpreter of simPL2, we introduce exception values **Exc**. Exceptions are represented by records.

Sem. dom.	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Exc} + \mathbf{Fun} + \mathbf{Rec}$	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
Exc	**Rec**	exceptions

Note that we can distinguish exceptions from other records because of the disjoint union in the definition of **EV**.

Any expression whose evaluation encounters an exception evaluates to that exception. As in Section 8.6, it is tedious to write down all possible cases. To show the principle, we look at the case of addition, division and "`.`".

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

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

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

Note that we need to be a bit more specific here than in Section 8.6, because different exceptions can come from different arguments of the addition.

The first three rules for division are similar.

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

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

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

The last rule for division covers the case that the meaning of the second argument of division is 0. We need to "raise the appropriate exception, corresponding to the exceptional situation.

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail 0}{\Delta \Vdash E_1\texttt{/}E_2 \rightarrowtail e} \text{ if } v_1 \notin \mathbf{Exc} \text{ and where } e = \texttt{[DivisionByZero:true]},$$

and $e \in \mathbf{Exc}$

The rules are carefully designed to be non-overlapping, to avoid ambiguities in the interpreter.

The rules for record access are as follows.

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

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

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash E\texttt{.}q \rightarrowtail e} \quad \text{if } q \notin \mathit{dom}(v) \text{ and where } e = \texttt{[InvalidRecordAccess: true]}, \text{ and } e \in \mathbf{Exc}$$

Finally, the meaning of `throw` expressions is defined as follows.

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \texttt{throw}\ E\ \texttt{end} \rightarrowtail e} \text{ if } v \in \mathbf{Rec} \cup \mathbf{Exc} \text{ and where } e = v,\ e \in \mathbf{Exc}$$

Note the subtle point that in the evaluation of E, and exception e may occur. In this case, the exception e will be thrown rather than the "intended" value of E.

10.6 Pass-by-value, Pass-by-name, and Pass-by-need

10.6.1 Pass-by-value

We have seen in previous chapters that evaluation of simPL programs is restricted such that only values can be passed as parameters to functions. This strategy of evaluating expressions is therefore called *pass-by-value* (In textbooks, you find the term "call-by-value".)

Pass-by-value enjoys a simplicity and efficiency that makes it the standard parameter passing technique in programming. Imperative languages such as Java, C, Pascal and functional programming languages such as SML, Ocaml, LISP and Scheme all use pass-by-value parameter passing.

10.6.2 Pass-by-name

In this section, we modify the semantics of simPL such that the evaluation of function parameters is delayed until their value is actually used in the function body. Remember that function definitions evaluate to mathematical functions, which take a denotable value as argument. Application applies these functions to the evaluated argument. In order to describe pass-by-name, we need to allow to pass the expression to be evaluated as argument. Whenever this expression is needed during evaluation of the body of the function, it gets evaluated.

But what environment should be used when the expression gets finally evaluated? The standard answer is similar to the standard answer for function definitions: The environment at the time of creation.

Thus we need to pass to the functions the expression that represents the argument, together with the environment in which we need to evaluate it. The data structures needed for call-by-name evaluation, containing an expression and an environment with respect to which the expression is evaluated, is called *thunk*.

Sem. dom.	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Exc} + \mathbf{Fun} + \mathbf{Rec}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun} + \mathbf{Rec} + \mathbf{Thunk}$	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
Exc	**Rec**	exceptions
Thunk	rePL $*$ **Env**	thunks

Thus, denotable values can be thunks, which are pairs consisting of a rePL expression (syntax) and an environment. The semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$ needs to be modified correspondingly. Evaluation of function definition remains unchanged.

The evaluation of application simply passes thunks to the function to which the first component evaluates.

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

Evaluation of identifiers needs to distinguish the case that the environment has a thunk stored under the given identifier.

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

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

To get a similar behavior of by-name evaluation for let-expressions, we can use the translation of let to application in the previous chapter.

10.6.3 Pass-by-need

Pass-by-need is an optimization of pass-by-name such that a given passed expression is evaluated at most once. That means once it is evaluated, the result is remembered and the next access to the corresponding formal parameter uses this value. This evaluation scheme is used by functional programming languages like Haskell and Miranda. Pass-by-need is also called "delayed evaluation" or "lazy evaluation". Correspondingly, pass-by-value is called "eager evaluation".

10.7 Lazy Programming

In programming languages with pass-by-name or (more common) pass-by-need, programming techniques become possible that are not easily expressible with pass-by-value.

For example, we can define and access the (infinite) list of all integers as follows.

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

The expression `(makeints 0)` evaluates to a thunk. Since the evaluation of arguments is delayed as much as possible, call-by-name and call-by-need are also called "lazy" evaluation.

Only the record access operation triggers the computation of the list. Since the body of `makeints` is also evaluated lazily, the list will be evaluated only up to the second element. The result of the expression is the value 2.

We can continue this game and map the integers to their squares as in the following.

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

This program evaluates to 4 in call-by-name.

10.8 A Virtual Machine for rePL

In this section, we extend the simPL virtual machine sVM to accommodate the features of rePL, leading to the rePL virtual machine rVM. Sections 10.8.1, 10.8.2, and 10.8.3 show the compilation and execution of record construction, record operations, and handling and raising exceptions, respectively. Section 10.8.4 addresses efficiency issues regarding records in an implmentation of rVM.

10.8.1 Record Construction

To accommodate the compilation of record construction, we introduce the instructions `LDPS` q (LoaD Property Symbolic) and `RCDS` i (ReCorD Symbolic) as follows (see Section 8.3).

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

where q is a property and i is an integer. The compiler uses these instructions to compile records as follows.

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

We execute the `LDPS` instruction such that the property is pushed on the operand stack.

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

Consider the instruction sequence `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 execution of `LDPS` $q_1.s_1.\ldots.$`LDPS` $q_n.s_n$, the instruction `RCDS` n will find the association list for the record to be constructed in reverse order on the operand stack. Therefore, the correct rule for executing `RCDS` n is

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

Thus we simply push the function corresponding to the record (represented by a set of pairs) onto the operand stack.

10.8.2 Operations on Records

The operations `empty`, "`.`", and `hasproperty` are represented by respective rVM instructions `EMPTY`, `DOT`, and `HASP`.

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

The compiler translates the operations as follows.

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

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

The execution of these instructions is equally straightforward.

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

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

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

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

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

10.8.3 Handling and Raising Exceptions

Division by zero and record access throw exceptions, which are records of a form specified in Section 10.5.2. We introduce an rVM instruction `THROW`, which throws the record it finds on top of the operand stack as an exception. For division and record access to be able to throw exceptions, we find it convenient to modify the compilation of rePL expressions.

The idea is to place instructions for raising these two exceptions at the end of the instruction sequence. The instructions for division and record access can then jump to those instructions if necessary.

$$\frac{E \hookrightarrow s_1 \texttt{ [DivisionByZero:true] } \hookrightarrow s_2 \texttt{ [InvalidRecordAccess:true] } \hookrightarrow s_3}{E \twoheadrightarrow s_1.\texttt{DONE}.s_2.\texttt{THROW}.s_3.\texttt{THROW}}$$

We shall denote the beginning address of s_2, which is $|s_1|+1$, by $addr_{DivisionByZero}$ and the beginning address of s_3, which is $|s_1|+1+|s_2|+1$, by $addr_{InvalidRecordAccess}$. Division can now throw an exception as follows.

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

Similarly, a failing record access can throw an exception as follows.

$$\frac{s(pc) = \texttt{DOT}}{(q.v.os, pc, e, rs) \rightrightarrows_s (os, addr_{InvalidRecordAccess}, e, rs)} \text{ if } \nexists v.(q, v') \in v$$

Not surprisingly, we translate `throw` expressions as follows.

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

But what should the instruction `THROW` do? How can we exit the context in which exception is thrown and find the appropriate exception? And, how should `try...catch...with...end` expressions be translated and executed to be able to catch any exceptions arising in its `try` expression?

The answer to these questions lies in the runtime stack. We can reuse the runtime stack to keep track of the `catch...with...end` part of `try` expressions. An exception that is thrown in the `try` part will then pop stackframes from the runtime stack, until it finds the appropriate `catch...with...end` part.

We translate `try` statements as follows.

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

Note that the `TRY` instruction carries with it the `catch` identifier x and the relative address of the `with` part. The `try` statement is executed by pushing a special stack frame on the runtime stack, which remembers the `catch` identifier, the current environment and the address of the `with` expression.

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

The `ENDTRY` instruction pops the `catch` frame from the runtime stack.

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

Finally, we are in the position to explain throwing of an exception. An exception inspects the runtime stack in search for a `catch` stackframe. If the top frame on the runtime stack is not marked as a `catch` frame, it is simply popped.

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

In this case, the program counter is not advanced. The effect is that the `THROW` instruction pops stack frames from the runtime stack until it finds a `catch` frame. If the top frame is a `catch` frame, the `THROW` instruction installs the operand stack of that frame. It extends the frame's environment by a binding of the `catch` variable to the exception, which is still on top of the operand stack. Finally, it sets the program counter to the address stored in the stackframe.

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

Since we are abusing the runtime stack to take note of `try` expressions, we need to add a rule for the `RTN` instruction such that `catch` frames are ignored.

$$\frac{s(pc) = \texttt{RTN}\ n}{(v.os, pc, e, (\texttt{catch}, x, pc', os', e').rs) \rightrightarrows_s (v.os, pc, e, rs)}$$

Like for normal frames in the instruction `THROW`, we refuse to advance pc in this case. The result is that `RTN` pops stack frames until it finds a regular (non-`catch`) frame.

10.8.4 Implementing Records in rVM

As given in Section 10.8.2, the instructions `DOT` and `HASP` have a major problem. The representation of records as a set of pairs forces these instructions to inspect association after association, until the right one is found. The time complexity of the instructions is therefore at least linear with respect to the size of the records. Furthermore, since properties are strings, each handling of an association requires a string operation. This section discusses how to overcome these weaknesses.

The first crucial observation is that all properties that are used at runtime appear syntactically in the expression being executed. Thus the compiler can construct the set Q of all properties in a given expression E. The compiler can

calculate a bijection idp between Q and $[0 \ldots |Q| - 1]$. In the resulting instructions, the compiler can replace every occurrence of a property q in an instruction by $idp(q)$. Similar to the implementation of sVM, which replaces identifiers by integers, the implementation of rVM replaces properties by integers.

The compilation rules for record access and the operation `hasproperty` now are as follows.

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

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

The second observation is that all records are constructed by the `[...]` syntax, which explicitly lists all properties of the record. Thus the compiler can calculate a bijection idr between the set R of all property sets of records that can appear at runtime and the numbers $[0 \ldots |R| - 1]$.

Now, we give each property q of each record with properties $q_1, \ldots, q_n$ its alphabetical position $p(idr(\{q_1, \ldots, q_n\}), idp(q))$, starting with 0. If a record with index m does not have a property with index n, then we set $p(m, n) = -1$.

Example 10.1 *Let us say the compiler assigns the number 13 to the set of properties* $\{\texttt{a}, \texttt{b}\}$ *(thus* $idr(\{\texttt{a}, \texttt{b}\}) = 13$*), and the numbers 55 and 77 to the properties* `a` *and* `b` *(*$idp(\texttt{a}) = 55$ *and* $idp(\texttt{b}) = 77$*), respectively. Then the position of property* `a` *in the record* `[A:5 B:7]` *is given by* $p(13, 55) = 0$*, since* `a` *is the alphabetically first property of the record. For any property identifier* $n \neq 55, 77$*, we have* $p(13, n) = -1$*.*

With these two observations in place, it is easy to see that we can represent each record with properties $q_1, \ldots, q_n$ as a pair consisting of the record identifier $idr(\{q_1, \ldots, q_n\})$ and an array that maps the alphabetical position of q in $q_1, \ldots, q_n$ given by $p(idr(\{q_1, \ldots, q_n\}), idp(q))$ to the corresponding property value.

Example 10.2 *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* $(13, [0 : 5, 1 : 7])$*.*

The compiler now translates record construction as follows.

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

In order for the rVM to take advantage of this arrangement, the compiler needs to pass the table p to the rVM. The new instruction `RCD` constructs an array, whose indices corresponding to the record properties are given by p.

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

Note that the record is now represented by a pair consisting of the record index m and the array containing the property values.

The instructions `DOT` and `HASP` now amount to array access using an index that we look up using p. If `DOT` does not find an index, it raises an exception, whereas `HASP` returns *false* in that case.

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

$$\frac{s(pc) = \texttt{DOT}}{(i.(m, a).os, pc, e, rs) \rightrightarrows_s (os, addr_{InvalidRecordAccess}, e, rs)} \text{ if } p(m, i) = -1$$

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

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

In summary, we have implemented record access operations such that they require constant time. To achieve this, the compiler replaces properties by integers, annotates record constructions with integers, and collects record-related information in a lookup-table p, which it passes to the rVM. The rVM then represents records by arrays, which are accessed by a record access instruction, using the indices computed by the compiler.

Bibliography

[cJS87] Technical committee: JTC 1/SC 2. Information processing—8-bit single-byte coded graphic character sets—part 1: Latin, alphabet no. 1. Technical report, International Organization for Standardization, 1987.