CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 11

dPL: Algebraic Data Types

The language simPL has one important deficiency; it lacks the possibility to directly build 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 11.2 by algebraic data types, which permit the programmer to directly define complex data structures that remains well-typed. To access the data components, we introduce a new language construct, called *pattern-matching* that can be used to perform both matching and extraction on the new algebraic data types. Section 11.3 gives examples of how to use this extension in practice. Section 11.4 shows the necessary extension in a type system to support algebraic data types for dPL.

Section 11.5 covers an interpreter for the dPL language, along the lines of the simPL interpreter. We explore alternative evaluation strategies in the context of dPL in Section 11.6. Section 11.7 shows a few programming techniques that become possible with one of these strategies. Finally, we extend the virtual machine for simPL in Section 11.8 by instructions that allow us to manage dPL's new data structures.

11.1 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. But 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 dPL (data-oriented Programming Language).

11.2 dPL with Algebraic Data Types

In order to define and use data types in a type-safe manner, we will introduce a new way to construct new data types in an algebraic manner. Our new type formation is expressed using the following construct:

$$\begin{array}{rcl} \texttt{type}\ t\ \texttt{'v}_1 \ldots \texttt{'v}_s = & & C_1\ t1_1 \ldots t1_{n1} \\ & \mid & C_2\ t2_1 \ldots t2_{n2} \\ & & \vdots \\ & \mid & C_m tm_1 \ldots tm_{nm} \end{array}$$

This construction is possibly polymorphic in that we allow type variables `'v`$_1$ ... `'v`$_s$. Secondly, we can use it to support both sum types and product types, but must provide a distinct constructor tag from $\{C_1, \ldots C_m\}$ for each of its component types. For example, if we wish to have a sum type that accepts either integer or boolean, we can use the following type definition to capture it.

```
type mix = I int
         | B bool
```

Using this type definition, we can construct an integer component under it using `I 3`, or a boolean component using `B true`. Both these values can now co-exist together in a type-safe manner under a new sum type, called `mix` here. At any instance in time, each value of this type either has an integer component or it has a boolean component, as distinguished by the constructor tag.

We may also capture a polymorphic pair of type, as an example of product types, as follows:

```
type pair 'a 'b = Pair 'a 'b
```

This type definition now allows two values of types `'a` and `'b` to be simultaneously captured by each of its elements, hence a product type. An example is `(Pair 3 true)`, with is type as `pair int bool`. This value now captures both an integer 3 and a boolean true in its two components.

More interesting algebraic data types can be used to capture complex data types, such as lists or trees. We can define the type for polymorphic list, as follows:

```
type list 'a = Nil
             | Node 'a (list 'a)
```

This can be used for a list of integers, e.g. `(Cons 1 (Cons 2 (Cons 3 Nil)))`, or a list of booleans, e.g. `(Cons true (Cons false (Cons true Nil)))`, or a list of either integers or booleans, e.g. `(Cons (I 1) (Cons (B true) Nil))`, with the help of `mix` type that we have defined earlier.

Data construction is therefore quite easy. How about the process of deconstruction for these data types? This is critical for accessing the components of data types. One possibility is to use primitive functions for testing an accessing the components. However, this may be a little low level and tedious, since we have to provide a set of such primitives for each algebraic data type that we define. Let us use instead a pattern-matching `match` construct, similar to the one you have seen in OCaml, that is very much suited for algebraic data types.

To access the first element of the `Pair` constructor, we can use the following function that match `x` against the pattern `Pair a b`, and then returning the first matched variable `a`

```
fun x -> match x with Pair a b -> a end end
```

To access the second element, we can use the following function instead that returns the second matched variable `b`.

```
fun x -> match x with Pair a b -> b end end
```

For data types with more than one constructor tags, we can list a number of patterns and the operations that are applicable for each of those cases. For example, we can write a recursive function to compute the length of a list, as follows:

```
recfun len xs -> match xs with
   Nil -> 0 ;
   Cons a b -> 1+(len b) end end
```

11.2.1 Syntax of dPL

With this, we have completed the language machinery to define and manipulate algebraic data types. Our programs will now consist of two parts, firstly a declaration for algebraic data types, followed by an expression that is to be evaluated. This can be expressed by the following rules on the syntax of dPL:

$$\frac{T_1 \;\in\; TDecl \;\cdots\; T_n \;\in\; TDecl \qquad E \in Expr}{T_1; \cdots; T_n; E \in Prog}$$

The next two rules captures the syntax on type declaration and constructor type for dPL programs.

$$\frac{T_1 \in Ctr \;\cdots\; T_m \in Ctr}{\texttt{type}\; c\; \texttt{'v}_1\texttt{...'v}_n \;=\; T_1 \mid \cdots \mid T_n \in TDecl}$$

$$\frac{}{C\; t_1 \;\ldots\; t_n \in Ctr}$$

The following two rules capture the syntax for data construction and deconstruction on algebraic data types used in dPL programs.

$$\frac{E_1 \in Expr \;\cdots\; E_n \in Expr}{C\; E_1 \;\ldots\; E_n \in Expr}$$

$$\frac{P_1 \in Pat \;\cdots\; P_n \in Pat \qquad E_1 \in Expr \;\cdots\; E_n \in Expr}{\texttt{match}\; E\; \texttt{with}\; P_1\texttt{->}\; E_1;\; \cdots\; ;\; P_n\texttt{->}\; E_n\; \texttt{end} \in Expr}$$

For pattern-matching, we currently use simple patterns of the form below.

$$\frac{}{C\ v_1\ \ldots\ v_n \in Pat}$$

More complex patterns can be also supported, and they can be viewed as syntactic sugar for dPL. It is possible to transform complex pattern-matching into a set of nested `match` constructs that performs one level of matching at a time. Consider the following code fragment that uses a more complex two-level pattern:

```
match xs with
  Cons x (Con y ys) -> ..
end
```

This can be translated to the following code with a nested `match` command that uses only simple patterns.

```
match xs with
 Cons x nv ->
  match nv with  (Con y ys) -> .. end
end
```

Take note that a fresh variable `nv` is used as a variable to pass an intermediate value to the inner `match` construct. Thus, with a means to transform complex patterns into simple patterns, it is sufficient to define just simple patterns for the core language of dPL.

The other syntax rules of dPL are similar to that for simPL and you may refer to Section **??** for details.

11.3 Examples

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

```
let even = recfun {int->int->int->list int}
  even i counter done ->
    if counter=done then Nil
    else Cons 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

```
Cons 2 (Cons 4 (Cons 6 Nil))
```

Similarly, we may construct useful higher-order functions over list data type with the help of pattern-matching. An example is:

```
recfun {(’a->’b)->list ’a->list ’b}
 map f xs -> match xs with
   Nil -> Nil ;
   Cons y ys -> Cons (f y) (map f ys) end end
```

To square every element of the list `Cons 1 (Cons 2 (Cons 3 Nil))`, we apply `map` to the list and the square function

```
(map (Cons 1 (Cons 2 (Cons 3 Nil))) fun x -> x * x end)
```

which returns `Cons 1 (Cons 4 (Cons 9 Nil))`.

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

```
recfun {(’a->’b->’b)->list ’a->’b->’b}
fold f xs start ->
 match xs with
  Nil -> start;
  Cons y ys -> f y (fold f xs start)
 end
end
```

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

```
(fold (fun x y -> x + y end) (Cons 1 (Cons 4 (Cons 9 Nil))) 0)
```

which returns 14.

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

11.4 Type System for dPL

As we have endeavoured to support a strongly typed language, we shall now look at how algebraic data types can be supported in an extended type system for dPL. The syntax of types for simPL is marginally extended to the following for dPL.

$$\frac{}{\mathit{int} \in \mathit{Type}} \qquad \frac{}{\mathit{bool} \in \mathit{Type}} \qquad \frac{}{\texttt{'a} \in \mathit{Type}}$$

$$\frac{t_1 \in \mathit{Type} \qquad t_2 \in \mathit{Type}}{t_1 \texttt{ -> } t_2 \in \mathit{Type}} \qquad \frac{t \in \mathit{Type}}{\forall \texttt{ 'a}.t \in \mathit{Type}}$$

$$\frac{t_1 \in \mathit{Type} \ \cdots \ t_n \in \mathit{Type}}{c\ t_1 \ \cdots \ t_n \in \mathit{Type}}$$

$$\frac{t_1 \in \mathit{Type} \ \cdots \ t_n \in \mathit{Type}}{C\ e_1 \ \cdots \ t_n \in \mathit{Type}}$$

Let us now provide a set of type rules to cater to data constructors and their operations. We start with a way to determine if a constructor belongs to a declared type, as follows:

$$\frac{\begin{array}{c} \mathit{type}\ c\ 'a_1 \ldots 'a_m = \cdots \mid C\ t''_1 \cdots t''_n \mid \cdots \in \mathit{TDecl} \\ \rho = ['a_1 \mapsto t'_1, \ldots, 'a_n \mapsto t'_n] \qquad t_1 = \rho t''_1 \cdots t_n = \rho t''_n \end{array}}{C\ t_1 \ \cdots \ t_n \in c\ t'_1 \ldots t'_m}$$

Note that *TDecl* captures a set of algebraic data types that has been declared in the given program. We also determine a substitution ρ to capture the type instantiation that has to be given for some data type. With this auxiliary meta-function, we can now provide a type rule for data constructor, as shown below. As we intend to infer the instantiated type for some polymorphic type, we use a set of fresh type variables, namely $'a_1 \ldots 'a_m$, to assist with type inference.

$$\frac{\begin{array}{c} C\ t_1 \ \cdots \ t_n \in c\ 'a_1 \ldots 'a_m \qquad \mathit{fresh}\ 'a_1 \ldots 'a_m \\ \Gamma \vdash e_1 \ : \ t_1 \quad \cdots \quad \Gamma \vdash e_n \ : \ t_n \end{array}}{\Gamma \vdash C\ e_1 \ \cdots \ e_n : c\ 'a_1 \ldots 'a_m} \ \textbf{[Constr]}$$

We can also provide a type rule for the pattern-matching, as shown below. Note that all the branches must have the same type, namely t.

$$\frac{\begin{array}{c}\Gamma \vdash E : t_0 \\ \Gamma{+}(P_1 : t_0) \vdash E_1 : t \\ \cdots \\ \Gamma{+}(P_n : t_0) \vdash E_n : t\end{array}}{\Gamma \vdash \texttt{match}\ E\ \texttt{with}\ P_1\texttt{->}\ E_1\texttt{;}\ \cdots\ \texttt{;}\ P_n\texttt{->}\ E_n\ \texttt{end} : t}$$

In the above rule, there is a need to extend the type environment with local variables used by pattern-matching. We support such a type extension by the notation $\Gamma{+}(P : t)$, where $P : t$ is defined as below.

$$\frac{C\ t_1 \cdots t_n\ \in\ t}{(C\ v_1 \ldots v_n) : t \Longrightarrow [v_1{:}t_1, \ldots, v_n{:}t_n]}$$

Let us also provide the type rules for exception handling, with the help of $\perp$ type to denote a raised exception. Such a $\perp$ type is special in that it is at the bottom of the type hierarchy, and can in fact unify with any type.

$$\frac{\Gamma \vdash e : Int}{\Gamma \vdash \texttt{throw}\ e\ :\ \perp}\ \textbf{[throw]}$$

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma{+}[x{\mapsto}Int] \vdash e_2 : t}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{catch x with}\ e_2\ \texttt{end} :\ t}\ \textbf{[try-catch]}$$

11.5 An Interpreter for dPL

For building an interpreter for dPL, we will provide a denotational semantics for the language.

11.5.1 Semantics for dPL

To support algebraic data types, we extend our semantic domains as follows.

Sem. domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Exc} + \mathbf{Fun} + \mathbf{Dat}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun} + \mathbf{Dat}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \leadsto \mathbf{DV}$	environments
Fun	$\mathbf{DV} \leadsto \mathbf{EV}$	function values
Dat	$\mathbf{c}\ \mathbf{DV} \cdots \mathbf{DV}$	data constructors
Exc	$\perp(\mathbf{Int})$	exceptions

We introduce a new domain **Dat** to capture data constructors that may be produced by algebraic data type construction. Our evaluation of them will proceed in a left-to-right manner, as shown below, where we may raise an exception if the evaluation of one of its constructor arguments raises it.

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \quad \cdots \quad \Delta \Vdash E_k \rightarrowtail v_k \quad k < n \quad \Delta \Vdash E_{k+1} \rightarrowtail \perp(n)}{\Delta \Vdash c\ E_1 \ldots E_n \rightarrowtail \perp(n)}$$

If all the arguments evaluate to their corresponding values without exception, we will then return a constructed data node, as shown below.

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \quad \cdots \quad \Delta \Vdash E_n \rightarrowtail v_n}{\Delta \Vdash c\ E_1 \ldots E_n \rightarrowtail c\ v_1 \ldots v_n}$$

For evaluation of pattern-matching construct, we will first evaluate the main expression E, until either an exception is raised or a data constructor is evaluated.

$$\frac{\Delta \Vdash E \rightarrowtail \perp(n)}{\Delta \Vdash \texttt{match}\ E\ \texttt{with}\ P_1\texttt{->}\ E_1;\ \cdots\ ;\ P_n\texttt{->}\ E_n\ \texttt{end} \rightarrowtail \perp(n)}$$

In case a data constructor, say $C_i\ v_1\ \ldots\ v_n$, is evaluated, we will attempt to find its branch base on its constructor C_i. In case the pattern is missing in its branches, we will raise an exception, as shown below.

$$\frac{\Delta \Vdash E \rightarrowtail C_i\ v_1\ \ldots\ v_n \qquad \forall k \in \{1..n\}.tag(P_k) \neq C_i}{\Delta \Vdash \texttt{match}\ E\ \texttt{with}\ P_1\texttt{->}\ E_1;\ \cdots\ ;\ P_n\texttt{->}\ E_n\ \texttt{end} \rightarrowtail \perp(0)}$$

Otherwise, the pattern-matching construct, will provide a mapping , say $[w_1 \mapsto v_1, \ldots, w_m \mapsto v_m]$, that is used to extend the environment for interpreting the selected branch.

$$\frac{\Delta \Vdash E \rightarrowtail C_i\ v_1\ \ldots\ v_m \qquad \Delta + [w_1 \mapsto v_1, \ldots, w_m \mapsto v_m] \Vdash E_i \rightarrowtail v}{\Delta \Vdash \texttt{match}\ E\ \texttt{with}\ \cdots ;\ C_i\ w_1\ \ldots\ w_m\ \texttt{->} E_i;\ \cdots\ \texttt{end} \rightarrowtail v}$$

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

11.6.1 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.

11.6.2 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. domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Exc} + \mathbf{Fun} + \mathbf{Dat}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun} + \mathbf{Dat} + \mathbf{Thunk}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values
Dat	$\mathbf{c}\ \mathbf{DV} \cdots \mathbf{DV}$	data constructors
Exc	$\perp(\mathbf{Int})$	exceptions
Thunk	$\mathbf{Fun} * \mathbf{Env}$	thunks

Thus, denotable values can be thunks, which are pairs consisting of a dPL 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.

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

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.

11.6.3 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".

11.7 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 ->
                     (Cons i (makeints i + 1)) end
    head = fun x -> match x with Cons a b -> a end end
    tail = fun x -> match x with Cons a b -> b end end
in let allints = (makeints 0)
   in  head (tail (tail allints))
   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 pattern-matching construct 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 allints = (makeints 0)
in
   let allsquares = (map allints fun x -> x * x end)
   in  head (tail (tail allsquares))
   end
end
```

11.8 A Virtual Machine for dPL

In this section, we extend the simPL virtual machine sVM to accommodate the features of dPL, leading to the dPL virtual machine dVM. Sections 11.8.1, 11.8.2, and 11.8.3 show the compilation and execution of record construction, the execution of pattern-matching and handling and raising exceptions, respectively.

11.8.1 Data Construction

To accommodate the compilation of data construction, we introduce the instructions `LDCS` $((C_i, i), n)$ (LoaD Constructor Symbolic) where C_i denotes the symbolic name of the data constructor, i is its corresponding integer constructor tag and n denotes the constructor's arity, as follows (see Section 8.3).

$$\frac{s}{\texttt{LDCS}\ ((C_i, i), n).s}$$

The compiler uses these instructions to compile data constructors, as follows.

$$\frac{E_1 \hookrightarrow s_1 \qquad \cdots \qquad E_n \hookrightarrow s_n}{C_i\ E_1\ \ldots\ E_n \hookrightarrow s_1.\ldots.s_n.\texttt{LDCS}\ ((C_i, i), n)}$$

We execute the LDCS instruction, as follows:

$$\frac{s(pc) = \texttt{LDCS}\ ((\_, i), n)\ q}{(v_n.\ldots.v_1.os, pc, e, rs) \rightrightarrows_s (C(i, v_1, \ldots, v_n).os, pc+1, e, rs)}$$

Thus, we simply push a data object corresponding to the algebraic data type being constructed onto the operand stack.

11.8.2 Pattern-Matching for Deconstructing Data

Data deconstruction is achieved using pattern-matching. For our virtual machine, we will use two machine instructions to help with pattern-matching, as follows:

$$\frac{s}{\texttt{SWITCH}\ n.s} \qquad \frac{s}{\texttt{ENDSC}\ (m, l).s}$$

The first instruction SWITCH n is meant to switch between the different branches of pattern-matching constructs. The parameter n is used to denote the number of branches.

The second instruction ENDSC (m, l) is use to remove m local variables from the environment at the end of a branch. This occurs because these local variables are to be removed at the end of scope of the pattern-matching branches. This instruction also contains an address label l where the virtual machine is expected to jump to.

The compiler translates the operations as follows.

$$\frac{\textit{fresh}\ l, l_1 \cdots l_n \quad E \hookrightarrow s \quad P_1; l; E_1 \hookrightarrow l_1 : s_1 \quad \cdots \quad P_n; l; E_n \hookrightarrow l_n : s_n}{\begin{array}{c}\texttt{match}\ E\ \texttt{with}\ P_1\texttt{->}\ E_1\texttt{;}\ \cdots\ \texttt{;}\ P_n\texttt{->}\ E_n\ \texttt{end} \\ \hookrightarrow s.\texttt{SWITCH}\ n.\texttt{GOTO}\ l_1.\ldots.\texttt{GOTO}\ l_n.s_1.\ldots.s_n.\texttt{LABEL}\ l\end{array}}$$

A slightly more optimized version of this compilation would inline the last branch in order to reduce a jump instruction, as shown below.

$$\frac{\textit{fresh}\ l, l_1 \cdots l_n \quad E \hookrightarrow s \quad P_1; l; E_1 \hookrightarrow l_1 : s_1 \quad \cdots \quad P_n; l; E_n \hookrightarrow l_n : s_n}{\begin{array}{c}\texttt{match}\ E\ \texttt{with}\ P_1\texttt{->}\ E_1;\ \cdots\ ;\ P_n\texttt{->}\ E_n\ \texttt{end} \\ \hookrightarrow s.\texttt{SWITCH}\ n.\texttt{GOTO}\ l_1.\ldots.\texttt{GOTO}\ l_{n-1}.s_n.s_1.\ldots.s_{n-1}.\texttt{LABEL}\ l\end{array}}$$

The compilation for each branch is illustrated below. Take note that we provide a fresh `LABEL` l for each branch to facilitate the compilation of jumping into the code of each pattern-matching branches using the `SWITCH` instruction.

$$\frac{\textit{fresh}\ l \qquad E_i \hookrightarrow s_i}{C_i\ v_1\ \ldots\ v_m; l_2; E_i \hookrightarrow \texttt{LABEL}\ l.s_i.\texttt{ENDSC}\ (m, l_2)}$$

The execution of the $SWITCH$ instruction is now detailed below. It also adds the constructor arguments $v_1, \ldots, v_m$ into the environment register prior to jumping into the respective branches.

$$\frac{i \leq n \qquad s(pc) = \texttt{SWITCH}\ n}{(C(i, v_1, \ldots, v_m).os, pc, e, rs) \rightrightarrows_s (os, pc{+}i, e[v_1, \ldots, v_m], rs)}$$

The `ENDSC` would remove the corresponding local variables from environment register e, prior to setting the program counter to its label l which denotes the exit of the pattern-matching construct.

$$\frac{s(pc) = \texttt{ENDSC}\ (m, l)}{(os, pc, e[v_1, \ldots, v_m], rs) \ \rightrightarrows_s\ (os, l, e, rs)}$$

11.8.3 Exception Handling

Exception handling is processed in a dynamic manner. When entering the scope of a try-catch, we need to place the catch handler into the run-time stack. This is to allow any exception to search for its corresponding handler by popping part will pop stackframes, until it finds the appropriate `catch...with...` part

We introduce three new instructions

- `TRY` : this sets up the catch handler prior to entering the scope of a try-block. It also remembers position of the data stack, to allow its correct restoration following an exception.
- `ENDTRY` : this signifies the safe exit of a try-block without any exception. Its main task is to pop the catch-handler from the runtime stack.
- `THROW` : this raises an exception whose integral exception value is on the run-time stack.

The compilation for try-catch and throw constructs are now implemented, as follows. We compile the catch handler at label l which ends with an `ENDSC` instruction to remove a local variable and for exiting the catch handler.

$$\frac{\textit{fresh}\ l_1, l_2 \qquad E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{\begin{array}{c}\texttt{try}\ E_1\ \texttt{catch}\ n\ \texttt{with}\ E_2\ \texttt{end} \hookrightarrow \\ (\texttt{TRY}\ l).s_1.(\texttt{ENDTRY}\ l_2).\texttt{LABEL}\ l.s_2.\texttt{ENDSC}\ (1, l_2)\texttt{LABEL}\ l_2\end{array}}$$

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

The semantics for the new machine instructions are now detailed next. The `TRY` would set up a handler $(-1(os), l, e)$ on the runtime stack. This handler remembers the location l of the handler, and the data stack os and the environment e at the stack of the execution of the try-catch construct. We also use the -1 integer to distinguish catch handler from other normal stack frames.

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

The `ENDTRY` l is executed when a try-block is exited without encountering any exception that has not been properly handled. We expect its corresponding catch handler to be on top of the run-time stack which is directly popped prior to jumping to the exit l of the try-catch command.

$$\frac{s(pc) = \texttt{ENDTRY}\ l}{(os, pc, e, (-1(\_), \_, \_).rs) \rightrightarrows_s (os, l, e, rs)}$$

The THROW instruction is perhaps the most involved, as it is expected to pop each stack frame out until the corresponding catch handler (that has been marked with the -1 integer tag) is encountered.

$$\frac{n \neq -1 \qquad s(pc) = \texttt{THROW}}{(os, pc, e, (n, pc', e').rs) \rightrightarrows_s (os, pc, e, rs)}$$

When it enters the catch handler, it also adds the integral exception value into the environment. It also dynamically resets the operand stack and environment register to the one saved at the start of the try-catch construct.

$$\frac{s(pc) = \texttt{THROW}}{(\perp(i).os, pc, e, (-1(os'), pc', e').rs) \rightrightarrows_s (os', pc', e'[i], rs)}$$