Algebraic Data Types with dPL

YSC-3208: Programming Language Design & Implementation

Răzvan Voicu

Week 08, March 6-10

Data Types in SimPL

- One deficiency in simPL its inability to model complex data types.
- If we drop well-typedness requirement, we can actually use functions to encode data types!

Pairs as Functions

Constucting a pair using a conditional

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

Accessing Pairs

Accessing pairs by application

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

Generic Construction of Pairs

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

Motivation

Disadvantages of data structures as functions:

- difficult to distinguish functions from data structures
- definition of data structures with many components gives rise to large nested conditionals
- inefficient, due to the function closures created, and linear execution of nested conditionals.
- loss of strong typing property

Defining Algebraic Data Types

Provide a new way to declare algebraic data types:

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

Polymorphism through type variables $\texttt{'v}_1\ \ldots\ \texttt{'v}_s$

Distinct constructor tags : $\{C_1,\ \ldots\ C_m\}$

Defining Sum Type

- Algebraic data types allow us to define sum/union types.
- An example is:

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

- Sum/union types allow us to combine different types into the same domain with the help of constructor tags.
  For example, `I 3` supports an integer value, while `B true` supports a boolean value in the `mix` type.

Defining Enumerated Data Type

- Algebraic data types can also support enumerated types.
- An example is:

```
type color = Red | Green | Blue
```

Defining Product Type

- Algebraic data types allow us to define product types.
- An example is:

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

- This type is polymorphic and allows two values of types `'a` and `'b` to be simultaneously captured.
  Thus `(Pair 3 true)` is a value with type `pair int bool`.
  Nested pairs are also possible. Example:
  `(Pair 3 (Pair true 4))`

Defining Recursive Type

- We can also define recursive types:

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

- This allows us to build lists of arbitrary lengths. For example :
  - A two-elements list is constructed using `(Cons e1 (Cons e2 Nil))`.
  - A three-elements is constructed using `(Cons e1 (Cons e2 (Cons e3 Nil)))`.

Data Accesses using Pattern-Matching

- While data construction is easy, we need a special pattern-matching language construct for accessing the values by *data deconstruction*.
- Some examples:

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

match x with Pair a b -> b end

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

Syntax of dPL

Program consists of type declarations and an expression:

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

Syntax of dPL

Syntax for type declaration:

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

Syntax for constructor type.

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

Syntax of dPL

Data construction:

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

Deconstruction:

$$\frac{P_1 \in \mathit{Pat} \quad \cdots \quad P_m \in \mathit{Pat} \qquad E_1 \in \mathit{Expr} \quad \cdots \quad E_m \in \mathit{Expr}}{\texttt{match}\ E\ \texttt{with}\ P_1\texttt{->}\ E_1\texttt{;}\ \ \cdots\ \texttt{;}\ P_m\texttt{->}\ E_m\ \texttt{end} \in \mathit{Expr}}$$

Simple Patterns

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

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

Each complex pattern can be compiled into simpler patterns.

Syntax of dPL (from simPL)

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

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

$$\frac{E}{p[E]} \text{ where } p \in \{\texttt{\textbackslash}, \texttt{\~{}}\}$$

Syntax of dPL (from simPL)

$$\frac{E \qquad E_1 \qquad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}}$$

$$\frac{E \quad E_1 \quad \cdots \quad E_n}{\texttt{(}E \; E_1 \cdots E_n\texttt{)}}$$

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

dPL Inherits From simPL

$$\frac{E}{\texttt{fun}\ \ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}}$$

$$\frac{E}{\texttt{recfun}\ f\ \ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}}$$

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
```

Examples

The following function applies a function to every element of its given list.

```
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
```

Examples

The following function applies a binary function `f` to reduce the list to some given value.

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

Syntax for Types

Marginally extended from polymorphic simPL.

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

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

Syntax for Types

$$\frac{t_1 \in \mathit{Type} \quad \cdots \quad t_m \in \mathit{Type}}{c \; t_1 \cdots t_m \in \mathit{Type}}$$

Type Rules

We can check if a constructor belongs to declared type as follows:

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

Type Rules

Type rules for variables and data constructors:

$$\frac{\Gamma(x) = t \quad inst(t) \searrow t_2}{\Gamma \vdash x : t_2}\textbf{[VarI]}$$

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

Type Rule for Pattern Matching

We can type check/infer on pattern-matching, as follows:

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

Type Environment Extension

Need to extend type environment with local variables used by pattern-matching.

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

Type Rule for Try-Catch

- What about type rules for exception handling?
- Define a new type $\perp$.
- Support $\forall t \cdot \perp <: t$.

Type Rules

Type rules for exception handling:

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

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

Semantic Domains for dPL

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** + **Dat**	expressible values
DV	**Bool** + **Int** + **Fun** + **Dat**	denotable values
Id	alphanumeric string	identifiers
Env	**Id** $\leadsto$ **DV**	environments
Fun	**DV** $\leadsto$ **EV**	function values
Dat	**c** **DV** $\cdots$ **DV**	data constructors
Exc	$\perp$(**Int**)	exceptions

Rules for dPL

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

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

Rules for dPL

Exception occurring:

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

Missing pattern:

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

Rules for dPL

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

New Instruction for Data Construction

New instructions:

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

where x is its symbolic name, i is its integer constructor tag and n is its arity.

Compilation of Data Construction

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

Execution of LDCS

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

New Instruction for Pattern-Matching

New instructions:

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

where n denotes the number of branches, m denotes the number of local variables to be removed at the end of scope.

Compilation of Pattern-Matching

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

Compilation of Pattern-Matching (optimised)

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

Compilation of Pattern Clauses

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

Execution of SWITCH

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

Execution of ENDSC

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

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

Compilation for try Statement

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

Compilation for `throw` Statement

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

Execution of TRY Instruction

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

-1 denotes the TRY catch frame

Execution of ENDTRY Instruction

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

Throwing and Catching an Exception

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

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