New vocabulary

Data:
• Constructed data
• Value constructor

Code:
• Pattern
• Pattern matching
• Clausal definition
• Clause

Types:
• Type variable (\'a)
Scheme problems

Unsolved:
  • No abstractive/encapsulation for data

Solved:
  • Only one data structure
    (Define as many forms as you like)
  • Wrong number of arguments
    (Doesn’t typecheck)
  • car or cdr of non-list
    (Doesn’t typecheck)
  • car or cdr of empty list
    (Use pattern matching, not car/cdr)
Datatype definitions

datatype suit = HEARTS | DIAMONDS | CLUBS | SPADES

datatype 'a list = nil (* copy me NOT! *) | op :: of 'a * 'a list

datatype 'a heap = EHEAP | HEAP of 'a * 'a heap * 'a heap

type suit val HEARTS : suit, ...
type 'a list val nil : forall 'a . 'a list
val op :: : forall 'a .
    'a * 'a list -> 'a list

type 'a heap
val EHEAP: forall 'a .
val HEAP : forall 'a .
    'a * 'a heap * 'a heap -> 'a heap
Your turn: Define a type

An ordinary S-expression is one of

- A symbol (string)
- A number (int)
- A Boolean (bool)
- A list of ordinary S-expressions

Two steps:
1. For each form, choose a value constructor
2. Write the datatype definition
Other constructed data: Tuples

Always only one way to form

- Expressions \((e_1, e_2, \ldots, e_n)\)
- Patterns \((p_1, p_2, \ldots, p_n)\)

Example:

```haskell
let val (left, right) = splitList xs
in if abs (length left - length right) < 1
   then NONE
   else SOME "not nearly equal"
end
```
Eliminate values of algebraic types

New language construct case (an expression)

fun length xs =
    case xs
    of [] => 0
    | (x::xs) => 1 + length xs

Clausal definition is preferred
(sugar for val rec, fn, case)
case works for any datatype

```ml
fun toStr t =
    case t
    of EHEAP => "empty heap"
    | HEAP (v, left, right) =>
          "nonempty heap"

But often a clausal definition is better style:

```
Define algebraic data types for $SX_1$ and $SX_2$, where

\[ SX_1 = ATOM \cup \text{LIST}(SX_1) \]
\[ SX_2 = ATOM \cup \{ (\text{cons} \ v_1 \ v_2) \mid v_1 \in SX_2, v_2 \in SX_2 \} \]

(take $ATOM$, with ML type $atom$ as given)
Wait for it . . .
Exercise answers

```
datatype sx1 = ATOM1 of atom
            | LIST1 of sx1 list

datatype sx2 = ATOM2 of atom
            | PAIR2 of sx2 * sx2
```
Exception handling in action

```plaintext
loop (evaldef (reader (), rho, echo))
handle EOF => finish ()
| Div => continue "Division by zero"
| Overflow => continue "Arith overflow"
| RuntimeError msg => continue ("error: " ^ msg)
| IO.Io {name, ...} => continue ("I/O error: " ^ name)
| SyntaxError msg => continue ("error: " ^ msg)
| NotFound n => continue (n ^ "not found")
```
ML Traps and pitfalls
Order of clauses matters

fun take n (x::xs) = x :: take (n-1) xs
  | take 0 xs    = []
  | take n []    = []

(* what goes wrong? *)
Gotcha — overloading

- fun plus x y = x + y;
> val plus = fn : int -> int -> int
- fun plus x y = x + y : real;
> val plus = fn : real -> real -> real
Gotcha — equality types

- (fn (x, y) => x = y);

> val it = fn : ∀ ''a . ''a * ''a -> bool

Tyvar ''a is “equality type variable”:
- values must “admit equality”
- (functions don’t admit equality)
Gotcha — parentheses

Put parentheses around anything with case, handle, fn

Function application has higher precedence than any infix operator
Syntactic sugar for lists

- 1 :: 2 :: 3 :: 4 :: nil; (* :: associates to the right *)
> val it = [1, 2, 3, 4] : int list

- "the" :: "ML" :: "follies" :: [];
> val it = ["the", "ML", "follies"] : string list

> concat it;
val it = "theMLfollies" : string
ML from 10,000 feet
The value environment

Names bound to immutable values
Immutable ref and array values point to mutable locations

ML has no binding-changing assignment

Definitions add new bindings (hide old ones):

\[
\begin{align*}
\text{val} \; & \text{pattern} = \text{exp} \\
\text{val} \; & \text{rec} \; \text{pattern} = \text{exp} \\
\text{fun} \; & \text{id} \text{ent} \; \text{patterns} = \text{exp} \\
\text{datatype} \; \ldots & = \ldots
\end{align*}
\]
Nesting environments

At top level, definitions

Definitions contain expressions:
\[
def \ ::= \text{val } \text{pattern } = \text{exp}
\]

Expressions contain definitions:
\[
exp \ ::= \text{let } \text{defs} \text{ in } \text{exp} \text{ end}
\]

Sequence of \text{defs} has let-star semantics
What is a pattern?

pattern ::= variable
  | wildcard
  | value-constructor [pattern]
  | tuple-pattern
  | record-pattern
  | integer-literal
  | list-pattern

Design bug: no lexical distinction between
  • VALUE CONSTRUCTORS
  • variables

Workaround: programming convention
Function peculiarities: 1 argument

Each function takes 1 argument, returns 1 result

For “multiple arguments,” use tuples!

fun factorial n = 
  let fun f (i, prod) = 
    if i > n then prod else f (i+1, i*prod) 
  in  f (1, 1) 
end

fun factorial n = (* you can also Curry *) 
  let fun f i prod = 
    if i > n then prod else f (i+1) (i*prod) 
  in  f 1 1 
end
Mutual recursion

Let-star semantics will not do.

Use and (different from and also)!

fun a x = ... b (x-1) ...
and b y = ... a (y-1) ...

Syntax of ML types

Abstract syntax for types:

\[ ty \Rightarrow \text{TYVAR of string} \quad \text{type variable} \]

| \text{TYCON of string * ty list} | \text{apply type constructor} |

Each tycon takes fixed number of arguments.

- **nullary**: \( \text{int, bool, string, …} \)
- **unary**: \( \text{list, option, …} \)
- **binary**: \( \rightarrow \)
- **n-ary**: \( n\text{-ary tuples (infix \(*\)}} \)
Syntax of ML types

Concrete syntax is baroque:

\[ ty \Rightarrow tyvar \quad \text{type variable} \]
\[ \quad | \quad tycon \quad \text{(nullary) type constructor} \]
\[ \quad | \quad ty \ tycon \quad \text{(unary) type constructor} \]
\[ \quad | \quad (ty, \ \ldots, \ ty) \ tycon \quad \text{(n-ary) type constructor} \]
\[ \quad | \quad ty \ * \ \ldots \ * \ ty \quad \text{tuple type} \]
\[ \quad | \quad ty \rightarrow ty \quad \text{arrow (function) type} \]
\[ \quad | \quad (ty) \]

\[ tyvar \Rightarrow ' \text{identifier} \quad 'a', 'b', 'c', \ldots \]
\[ tycon \Rightarrow \text{identifier} \quad \text{list, int, bool,} \ldots \]
Polymorphic types

Abstract syntax of type scheme \( \sigma \):
\[
\sigma \Rightarrow \text{FORALL of tyvar list } \ast \text{ ty}
\]

Bad decision: \( \forall \) left out of concrete syntax

\[
\left( \text{fn } (f, g) \Rightarrow \text{fn } x \Rightarrow f \ (g \ x) \right)
\]

: \( \forall \ 'a, \ 'b, \ 'c \).

\( ('a \rightarrow 'b) \ast ('c \rightarrow 'a) \rightarrow ('c \rightarrow 'b) \)

Key idea: substitute for quantified type variables
Old and new friends

\[
\begin{align*}
\text{op o} & : \forall \ 'a, \ 'b, \ 'c . \\
& \quad (\ 'a \to \ 'b) \times (\ 'c \to \ 'a) \to \ 'c \to \ 'b \\
\text{length} & : \forall \ 'a . \ 'a \text{ list} \to \ \text{int} \\
\text{map} & : \forall \ 'a, \ 'b . \\
& \quad (\ 'a \to \ 'b) \to (\ 'a \text{ list} \to \ 'b \text{ list}) \\
\text{curry} & : \forall \ 'a, \ 'b, \ 'c . \\
& \quad (\ 'a \times \ 'b \to \ 'c) \to \ 'a \to \ 'b \to \ 'c \\
\text{id} & : \forall \ 'a . \ 'a \to \ 'a
\end{align*}
\]