Pairs (2-tuples)

Need a way to build pairs and a way to access the pieces

\textbf{Build:}

\begin{itemize}
  \item Syntax: \((e_1, e_2)\)
  \item Evaluation: Evaluate \(e_1\) to \(v_1\) and \(e_2\) to \(v_2\); result is \((v_1, v_2)\)
    \begin{itemize}
      \item A pair of values is a value
    \end{itemize}
  \item Type-check:
    \begin{itemize}
      \item If \(e_1\) has type \(\text{ta}\) and \(e_2\) has type \(\text{tb}\),
      \item then the pair expression has type \(\text{ta} \times \text{tb}\)
    \end{itemize}
    \begin{itemize}
      \item A new kind of type
    \end{itemize}
\end{itemize}

\textbf{Access: via a new form of binding (better style)}

\begin{itemize}
  \item Syntax: \(\text{val} \ (x_1, x_2) = e\)
  \item Type-checking:
    \begin{itemize}
      \item If \(e\) has type \(\text{ta} \times \text{tb}\),
      \item then \(x_1\) has type \(\text{ta}\) and \(x_2\) has type \(\text{tb}\)
    \end{itemize}
  \item Evaluation:
    \begin{itemize}
      \item Evaluate \(e\) to a pair of values \(v_1\) and \(v_2\) in the current dynamic environment
      \item Return \(v_1\) if using \(#1\); return \(v_2\) if using \(#2\).
    \end{itemize}
\end{itemize}
Examples

Functions can take and return pairs

```ml
fun swap (pr : int*bool) = 
  let val (x,y) = pr in (y,x) end

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) = 
  let val (x1,y1) = pr1 
     val (x2,y2) = pr2 
   in x1 + y1 + x2 + y2 end

fun div_mod (x : int, y : int) = 
  (x div y, x mod y)

fun sort_pair (pr : int*int) = 
  let val (x,y) = pr 
   in if x < y then pr else (y,x) end
```

Tuples

Actually, you can have tuples with more than two parts

- A new feature: a generalization of pairs
- \((e_1, e_2, \ldots, e_n)\)
- \(t_a \times t_b \times \ldots \times t_n\)
- \#1 e, \#2 e, \#3 e, …
- \(\text{val } (x_1, \ldots, x_n) = e\)

These really are flat n-tuples, not nested pairs.

Nesting

Pairs and tuples can be nested however you want

- Not a new feature: implied by the syntax and semantics

```ml
val x1 = (7,(true,9)) (* int * (bool*int) *)
val x2 = #1 (#2 x1) (* bool *)
val x3 = (#2 x1) (* bool*int *)
val x4 = ((3,5),((4,8),(0,0))) (* (int*int)*(int*int)* (int*int) *)
```

Lists

Let’s try to add lists to ML. Racket does this with pairs, e.g.:

```racket
(cons 1 (cons 2 (cons 3 null)))
```

ML has a "no value" value written (), pronounced "unit," with type `unit`

So let’s try: 

```ml
(1, (2, (3, ())))
```

What is the type of this expression?

What is the type of: 

```ml
(1, (2, (3, (4, ()))))
```

Why is this a problem?
Lists

Despite nested tuples, the type of an expression still "commits" to a particular fixed "amount" of data.

In contrast, a list:

• Can have any number of elements
• But all list elements have the same type

We need a new tool to build lists in ML.

How to build bigger types

• Already know:
  • Base types like int bool unit char
  • Ways to build (nested) compound types: tuples

• Today: more interesting compound types

• First: 3 most important type building blocks in any language
  • Product types ("Each of"): A t value contains values of each of t1 t2 … tn
  • Sum types ("One of"): A t value contains values of one of t1 t2 … tn
  • Recursive types ("Self reference"): A t value can refer to other t values

• Remarkable: much data can be described by just these building blocks

Note: versions in "quotes" are not widely used terms.

Records

Record values have fields (any name) holding values

{f1 = v1, …, fn = vn}

Record types have fields (any name) holding types

{f1 : t1, …, fn : tn}

The order of fields in a record value or type never matters
• REPL alphabetizes fields just for consistency

Building records:

{f1 = e1, …, fn = en}

Accessing components:

#myfieldname e

(Evaluation rules and type-checking as expected)

Example

{name = "Wendy", id = 41123 - 12}

Has type

{id : int, name : string}

And evaluates to

{id = 41111, name = "Wendy"}

If some expression such as a variable x has this type, then get fields with:

#id x  #name x

Note we did not have to declare any record types
• The same program could also make a

{id=true, ego=false} of type (id:bool, ego:bool)
By position vs. by name

(4, 7, 9) \{f=4, g=7, h=9\}

Common syntax decision:
• parts by position (as in tuples) or by name (as with records)
• Concise vs. clear.
• Taste, practicality, etc.

Common hybrid: function/method arguments:
• Caller: positional
• Callee: nominal
• Could totally do it differently; some languages have

Tuples are sugar

{el,...,en} desugars to \{l=el,...,n=en\}
{tl*...*tn} desugars to \{1:tl,...,n:tn\}

Records with contiguous fields 1...n printed like tuples
Can write \{1=4, 2=7, 3=9\}, bad style

Datatype bindings

Sum/one-of types:
```plaintext
datatype mytype = TwoInts of int * int | Str of string | Pizza
```

Algebraic Data Type
• Adds new type mytype to environment
• Adds constructors to environment: TwoInts, Str, Pizza
• Constructor: function that makes values of new type (or is a value of new type):
  - TwoInts : int * int -> mytype
  - Str : string -> mytype
  - Pizza : mytype

Constructing values

• Each value of type mytype came from one of the constructors
• Value contains:
  - Tag: which constructor (e.g., TwoInts)
  - Carried data (e.g., (7, 9))
• Examples:
  - TwoInts (3+4, 5+4) evaluates to TwoInts (7, 9)
  - Str if true then “hi” else “bye” evaluates to Str “hi”
  - Pizza is a value
Using values

Two aspects to accessing a datatype value
1. Check what variant it is (what constructor made it)
2. Extract carried data (if that variant has any)

ML could create functions to get parts of datatype values
• Like to pair? or cdr in Racket
• Instead it does something better... totally awesomely better.

Pattern matching

Case expression and pattern-matching

```
fun f x = (* f has type mytype -> int *)
case x of
  Pizza => 3
  TwoInts(i1,i2) => i1+i2
  Str s => String.size s
```

All-in-one:
• Multi-branch conditional, picks branch based on variant.
• Extracts data and binds to branch-local variables.
• Type-check: all branches must have same type.
  • Gets even better later.

Why pattern-matching rocks

1. Cannot forget a case (inexhaustive pattern-match warning)
2. Cannot duplicate a case (redundant pattern type-checking error)
3. Cannot forget to test the variant correctly and get an error ((car null) in Racket)
4. It's much more general. Supports elegant, concise code.
Useful examples

- Enumerations, including carrying other data

```plaintext
datatype suit = Club | Diamond | Heart | Spade
datatype card_value = Jack | Queen | King | Ace | Num of int
```

- Alternate ways of identifying real-world things/people

```plaintext
datatype id = StudentNum of int | Name of string | * (string option) | * string
```

Don’t do this!

Languages lacking convenient sum/one-of types foster bad style where product/each-of types are misused in place of sum/one-of types:

```plaintext
(* use the student_num and ignore other fields unless the student_num is -1 *)
{ student_num : int, first : string, middle : string option, last : string }
```

- Unclear. No help from the language managing/remembering variants.

That said...

But if instead the point is that every “person” in your program has a name and maybe a student number, then each-of is the way to go:

```plaintext
{ student_num : int option, first : string, middle : string option, last : string }
```

Lists!

A list is either:
- The empty list; or
- A pair of a list element and a list that holds the rest of the list.

Algebraic data types are just what we need for lists!

```plaintext
datatype mylist = Empty | Cons of int * mylist
```

```plaintext
val some_ints = Cons (1, Cons (2, Cons (3, Empty)))
```
Accessing Lists

```ml
val some_ints = Cons (1, Cons (2, Cons (3, Empty)))
```

```ml
fun length (xs : mylist) =  
  case xs of  
    | Empty => 0  
    | Cons (x, xs') => 1 + length xs'
```

```ml
fun sum (xs : mylist) =  
  case xs of  
    | Empty => 0  
    | Cons (x, xs') => x + sum xs'
```

Syntactic sugar for lists: build

Lists are important enough for their own syntax.

- The empty list is a value: `[]`
- A list of expressions/values is an expression/value; elements separated by commas:
  ```ml
  [e1,e2,...,en]  
  [v1,v2,...,vn]
  ```
- If `e1` evaluates to `v` and `e2` evaluates to a list `[v1,...,vn]`, then `e1::e2` evaluates to `[v,...,vn]`
  ```ml
e1::e2 (* pronounced "cons" *)
  ```

Syntactic sugar for lists: access

- With pattern-matching, of course.

```ml
val some_ints = [1,2,3]
```

```ml
fun length (xs : int list) =  
  case xs of  
    | [] => 0  
    | x::xs' => 1 + length xs'
```

```ml
fun sum (xs : int list) =  
  case xs of  
    | [] => 0  
    | x::xs' => x + sum xs'
```

Type-checking list operations

For any type `t`, type `t list` describes lists with all elements of type `t`
- `int list` `bool list` `int list list (int * int) list` `int list * int list` ...
- `[]` can have type `t list` list for any type
- SML uses type `"a list` to indicate this ("quote a" or "alpha")

- `e1::e2` type-checks with type `t list` if and only if:
  - `e1` has type `t` and
  - `e2` has type `t list`

More on `'a` soon! (Nothing to do with `'a` in Racket.)
Example list functions

fun countdown (x : int) = 
  if x=0
  then []
  else x :: countdown (x-1)

fun append (xs : int list, ys : int list) = 
  case xs of
  [] => ys
  | x::xs' => x :: append (xs', ys)

fun rev (xs : int list) = 
  let fun revtail (acc : int list, xs : int list) =
    case xs of
      [] => acc
      | x::xs' => revtail (x :: acc, xs')
  in
    revtail ([], xs)
  end

Example higher-order list functions

fun map (f : int -> int, xs : int list) = 
  case xs of
  [] => []
  | x::xs' => f x :: map (f, xs')

- But these examples only work on lists of ints.
- They should be more general: work on any list
  * and any function for map.

Polymorphic types and type inference

The identity function: 

fun id (x : int) = x

val id : int -> int

It should work on anything! Omit the type: 

fun id x = x

General!
- 'a is a polymorphic type variable that stands in for any type.
- "id takes an argument of any type and returns a result of that same type."
Polymorphic datatypes

- Let’s make lists that can hold elements of any one type.
  
  \[
  \text{datatype 'a mylist = Empty | Cons of 'a * 'a mylist}
  \]

- A list of "alphas" is either:
  - the empty list; or
  - a pair of an "alpha" and a list of "alphas"

\[
\text{datatype 'a list = [ ] | :: of 'a * 'a list}
\]

- The type \text{int list} is an instantiation of the type \text{'a list}, where the type variable \text{'a} is instantiated with \text{int}.

Polymorphic list functions

\[
\text{fun append (xs, ys) =}
\text{  case xs of}
\text{    [ ] \Rightarrow ys}
\text{   | x :: xs' \Rightarrow x :: append (xs', ys)}
\]

\[
\text{fun rev (xs) =}
\text{  let fun revtail (acc : int list, xs : int list) =}
\text{    case xs of}
\text{      [ ] \Rightarrow acc}
\text{      | x :: xs' \Rightarrow revtail (x :: acc, xs')}
\text{    in revtail [ ] xs end}
\]

\[
\text{fun map (f, xs) =}
\text{  case xs of}
\text{    [ ] \Rightarrow [ ]}
\text{    | x :: xs' \Rightarrow f x :: map (f, xs')}
\]

Exceptions

An exception binding introduces a new kind of exception

\[
\text{exception MyFirstException}
\text{exception MySecondException of int * int}
\]

The \text{raise} primitive raises (a.k.a. throws) an exception

\[
\text{raise MyFirstException}
\text{raise (MySecondException (7,9))}
\]

A handle expression can handle (a.k.a. catch) an exception

\[
\text{e1 handle MyFirstException \Rightarrow e2}
\text{e3 handle MyFirstException \Rightarrow e4}
\text{  | MySecondException (x,y) \Rightarrow e5}
\]
Actually...

Exceptions are a lot like datatype constructors...
• Declaring an exception adds a constructor for type `exn`
• Can pass values of `exn` anywhere (e.g., function arguments)
  • Not too common to do this but can be useful
• `handle` can have multiple branches with patterns for type `exn`, just like a `case` expression.

• See examples in `exnopt.sml`

Options

```
datatype 'a option = NONE | SOME of 'a
```
• `t option` is a type for any type `t`
  • (much like `t list`, but a different type, not a list)

Building:
• `NONE` has type `t option` (much like `[]` has type `t list`)
• `SOME e` has type `t option` if `e` has type `t` (much like `e::[]`)

Accessing:
• Pattern matching with case expression

Good style for functions that don’t always have a meaningful result.
See examples in `exnopt.sml`

Parametric Polymorphism (again) and the power of what you cannot do.

• Type `'a` means “some type, but don’t know what type”
  • There is no way to “figure out” what type it actually is.
  • No operation can distinguish between values of unknown type `'a`.

• Example: What can a function of type `'a list` -> `int` do?
  ```
  fun f (xs : `'a list`) : `int` = ...
  ```

• `'a` -> `'a` ?
  ```
  fun g (x : `'a`) : `'a` = ...
  ```

Equality Types

So if we cannot inspect values of type `'a` in any way, how do we write a general `contains` function?

```
fun contains (xs : `'a list`, x : `'a`) : bool = ...
```

`eqtypes` (equality types):
Special category of types that support comparison.
Accompanying `eqtype` variables with double quotes

Mostly accurate:
```
e : ("'a * 'b") -> bool
```

```
fun contains (xs : "'a list", x : "'a") : bool = ...
```