Datatypes, Patterns, and Parametric Polymorphism

Topics

- Tuples and records
- Positional vs. nominal
- Datatypes
- Pattern matching
- Parametric polymorphic types (generics)
- Lists and options
- Equality types

Tuples

Syntax: \((e_1, \ldots, e_n)\)

Evaluation:
1. Evaluate \(e_1\) to \(v_1\), ..., and \(e_n\) to \(v_n\).
2. The result is \((v_1, \ldots, v_n)\)

Type checking:
If \(e_1\) has type \(t_1\), ..., and \(e_n\) has type \(t_n\), then the pair expression has type \(t_a \times \ldots \times t_n\)

Tuple bindings

Syntax: \(\text{val} \ (x_1, x_2) = e\)

Type checking:
If \(e\) has type \(t_1 \times t_2\), then \#1 \(e\) has type \(t_1\) and \#2 \(e\) has type \(t_2\)

Evaluation:
1. Evaluate \(e\) to a pair of values \((v_1, v_2)\) in the current dynamic environment
2. Extend the current dynamic environment by binding \(x_1\) to \(v_1\) and \(x_2\) to \(v_2\).
Tuple accessors

Syntax: \#1 e \#2 e

Type checking:
If e has type t1 * t2, then \#1 e has type t1 and \#2 e has type t2

Evaluation:
1. Evaluate e to a pair of values v1 and v2 in the current dynamic environment
2. The result v1 if using \#1; v2 if using \#2

Examples

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

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}

Order of fields in a record value or type never matters

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}

Evaluates to
{id = 41111, name = "Wendy"}

If an expression, e.g. variable x, has this type, then get fields with:

#id x   #name x

No record type declarations!
– The same program could also make a
{id=true,ego=false} of type {id:bool,ego:bool}
**By position vs. by name**

*(structural/positional) (nominal)*

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

Common syntax decision

Common hybrid: function/method arguments

**Tuples are sugar**

\[(e_1, \ldots, e_n)\]

desugars to \(\{1=e_1, \ldots, n=e_n\}\)

\(t_1 \times \ldots \times t_n\)

desugars to \(\{1:t_1, \ldots, n:t_n\}\)

Records with contiguous fields 1...n printed like tuples

Can write \(\{1=4, 2=7, 3=9\}\). bad style

**How can we build lists?**

Racket: \((\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ \text{null}))\))

ML has a "no value" value written \(\)\), pronounced "unit," with type \textbf{unit}

What is the type of: \((1, (2, (3, ()))))\)

What is the type of: \((1, (2, (3, (4, ()))))\)

Why is this a problem?

**How to build bigger data types**

Data type building blocks in \textbf{any} language

– Product types (“Each of”):
  Value contains values of \textit{each of} \(t_1 \ t_2 \ \ldots \ t_n\)
  Value contains \(a \ t_1 \ \text{and} \ a \ t_2 \ \text{and} \ \ldots \ \text{and} \ a \ t_n\)

– Sum types (“One of”):
  Value contains values of \textit{one of} \(t_1 \ t_2 \ \ldots \ t_n\)
  Value is \(t_1 \ \text{xor} \ a \ t_2 \ \text{xor} \ \ldots \ \text{xor} \ a \ t_n\)

– Recursive types (“Self reference”):
  A \(t\) value can refer to other \(t\) values
Datatype bindings

```
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`

Datatypes: constructing values

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

- Values of type `mytype` produced by *one* of the constructors
- Value contains:
  - Tag: which constructor (e.g., `TwoInts`)
  - Carried value (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

Datatypes: using values

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 much 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.
Pattern matching

Syntax:

- (For now), each pattern $p_i$ is:
  - a constructor name followed by the right number of variables:
    - $C \text{ or } D \ x \text{ or } E (x,y) \text{ or } ...$
- Patterns are not expressions.
  - We do not evaluate them.
  - We match $e_0$ against their structure.
- Precise type-checking/evaluation rules later...

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, 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
```

Building (our own) list datatype

A list is either:

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

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

```plaintext
val some_ints = Cons (1, Cons (2, Cons (3, Empty)))
```
Accessing (our own) lists

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

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

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

ML lists: creating

The empty list is a value: `[]`

A list of expressions/values is an expression/value:

```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,v1,...,vn]`

ML lists: accessing

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

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

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

ML lists: type-checking

For any type `t`, type `t list` describes lists where all elements have type `t`.

```ml
[] : t list  list for any type `t`  
ML syntax: 'a list  (“quote a” or “alpha”)  
```

```ml
el::e2 : t list  if and only if:  
  - el : t  and  
  - e2 : t list
```

More `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')

• These examples only work on lists of ints.
• Should be more general: work on any list – and any function for map...

Polymorphic types + type inference

The identity function:  

fun id (x : int) = x  
val id : int -> int

Omit the type:  

fun id x = x  
val id : 'a -> 'a

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."
  – ∀'a, id : 'a -> 'a

Polymorphic types + type inference

fun swap pair =  
  let val (x,y) = pair in (y,x) end
val swap : ('a * 'b) -> ('b * 'a)

Works on any type of pair!

val pair = swap (4,"hello")
('a * 'b) is more general than (int * string).

Here, int instantiates 'a and string instantiates 'b.
Polymorphic datatypes

Lists that can hold elements of any one type.

```haskell
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"

```haskell
datatype 'a list = []
    | :: of 'a * 'a list
```

The type `int list` is an instantiation of the type `'a list`, where the type variable `'a` is instantiated with `int`.

Polymorphic list functions

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

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

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

Exceptions

An exception binding introduces a new kind of exception

```haskell
exception MyFirstException
exception MySecondException of int * int
```

The `raise` primitive raises (a.k.a. throws) an exception

```haskell
raise MyFirstException
raise (MySecondException (7,9))
```

A handle expression can handle (a.k.a. catch) an exception
- If doesn’t match, exception continues to propagate

```haskell
e1 handle MyFirstException => e2
e3 handle MyFirstException => e4
    | MySecondException (x,y) => 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 `t option = NONE | SOME of 'a`

t option is a type for any type `t`

Building:
- `NONE` has type `t option` (much like `[]` has type `'a 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 and the power of what you cannot do.

Type `'a` means "some type, but don't know what type"

What can a function of type `'a list` -> `int` do?

```haskell
fun f (xs : 'a list) : int = ...
```

```haskell
'a' -> 'a' ?
```

```haskell
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?

```haskell
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:

```haskell
= : (''a * ''a) -> bool
fun contains (xs : ''a list, x : ''a) : bool = ...
```