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 \(v_1\), \(\ldots\), and \(e_n\) to \(v_n\).
2. The result is \((v_1, \ldots, v_n)\)

Type checking:
If \(e_1\) has type \(t_1\), \(\ldots\), 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:  \[ #1 \ e \quad #2 \ e \]

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

Evaluation:
1. Evaluate \( e \) to a pair of values \( v_1 \) and \( v_2 \) in the current dynamic environment
2. The result \( v_1 \) if using \#1; \( v_2 \) if using \#2

Examples

```haskell
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

\{ f_1 = v_1, ..., f_n = v_n \}

**Record types** have fields (any name) holding types

\{ f_1 : t_1, ..., f_n : t_n \}

Order of fields in a record value or type never matters

Building records:

\{ f_1 = e_1, ..., f_n = e_n \}

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)

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

Common syntax decision

Common hybrid: function/method arguments

Tuples are sugar

\( (e_1,\ldots,e_n) \) desugars to

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

\( t_1*\ldots*t_n \) 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

Lists

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

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

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

– Recursive types (“\textit{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:
    – $C$ or $D$ or $E(x,y)$ 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

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

Alternate ways of identifying real-world things/people

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

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

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

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'

Syntactic sugar for lists: build

The empty list is a value:   [ ]

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

Syntactic sugar for lists: access

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'

Type-checking list operations

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

```ml
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

```ml
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:

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

Omit the type:

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

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

Polymorphic types + type inference

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

Works on any type of pair!

```ml
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')
```

- Type inference system chooses most general type.
- Polymorphic types show up commonly with higher-order functions.
- Polymorphic function types often give you a good idea of what the function does.

**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 'a option = NONE | SOME of 'a

options are a type for any type t

Building:
- NONE has type 'a 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

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

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?

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

'a -> 'a ?

fun g (x : 'a) : 'a = ...