Data Types

Prof. Clarkson
Fall 2017

Today’s music: *Pokémon Theme* by Jason Paige
Review

Previously in 3110:
• Functions
• Lists

Today:
• Let expressions
• Ways to define your own data types: variants, records, tuples
LET EXPRESSIONS
Let expressions

- Slightly different than the let definitions we've been using at the toplevel
- Enable binding of variables to values inside another expression
- Since they are expressions, they evaluate to values

```plaintext
let x = 2 in x+x  (* ==> 4 *)

let inc x = x+1 in inc 10  (* ==> 11 *)

let y = "big" in
let z = "red" in
y^z  (* ==> "bigred" *)
```
let expressions

Syntax:

\[
\text{let } x = e_1 \text{ in } e_2
\]

\(x\) is an identifier
\(e_1\) is the binding expression
\(e_2\) is the body expression

\text{let } x = e_1 \text{ in } e_2 \text{ is itself an expression}
let expressions

let x = e1 in e2

Evaluation:

- Evaluate $e_1$ to a value $v_1$
- Substitute $v_1$ for $x$ in $e_2$, yielding a new expression $e_2'$
- Evaluate $e_2'$ to $v_2$
- Result of evaluation is $v_2$
Let expressions

\[
\text{let } x = 1 + 4 \text{ in } x \times 3
\]

\[\rightarrow \text{ Evaluate } e_1 \text{ to a value } v_1\]

\[
\text{let } x = 5 \text{ in } x \times 3
\]

\[\rightarrow \text{ Substitute } v_1 \text{ for } x \text{ in } e_2, \text{ yielding a new expression } e_2'\]

\[
5 \times 3
\]

\[\rightarrow \text{ Evaluate } e_2' \text{ to } v_2\]

\[
15
\]

Result of evaluation is \(v_2\)
let expressions

let x = e1 in e2

Type-checking:

If e1:t1,
and if e2:t2 (assuming that x:t1),
then (let x = e1 in e2) : t2
Question

Which of these does not evaluate to 3?

A. let x = 3
B. let x = 2 in x+1
C. (fun x -> x+1) 2
D. let f x = x+1 in f 2
E. let f = fun x -> x+1 in f 2
Question

Which of these does not evaluate to 3?

A. \texttt{let x = 3}
B. \texttt{let x = 2 in x+1}
C. \texttt{(fun x -> x+1) 2}
D. \texttt{let f x = x+1 in f 2}
E. \texttt{let f = fun x -> x+1 in f 2}
Anonymous functions

These two expressions are syntactically different but semantically equivalent:

\[
\text{let } x = 2 \text{ in } x + 1
\]

\[
(\text{fun } x \to x + 1) \ 2
\]

Let expressions are syntactic sugar for anonymous function application
Let definitions in toplevel

Syntax:

\[ \text{let } x = e \]

Implicitly, “in rest of what you type”

E.g., you type:

\[
\begin{align*}
\text{let } a = "big" ; ; \\
\text{let } b = "red" ; ; \\
\text{let } c = a^b ; ; \\
\end{align*}
\]

Toplevel understands as

\[
\begin{align*}
\text{let } a = "big" \text{ in } \\
\text{let } b = "red" \text{ in } \\
\text{let } c = a^b \text{ in } \\
\end{align*}
\]
VARIANTS
Variant

```ocaml
let int_of_day d =
match d with
    | Sun -> 1
    | Mon -> 2
    | Tue -> 3
    | Wed -> 4
    | Thu -> 5
    | Fri -> 6
    | Sat -> 7
```

```ocaml
type day = Sun | Mon | Tue | Wed
            | Thu | Fri | Sat
```
Building and accessing variants

Syntax: \texttt{type t = C1 | \ldots | Cn}
the \texttt{Ci} are called 	extit{constructors}

Evaluation: a constructor is already a value

Type checking: \texttt{Ci : t}

Accessing: use pattern matching; constructor name is a pattern
Pokémon variant

<table>
<thead>
<tr>
<th></th>
<th>NOR</th>
<th>FIR</th>
<th>WAT E</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORMAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRE</td>
<td>½</td>
<td>½</td>
<td></td>
</tr>
<tr>
<td>WATER</td>
<td>2</td>
<td>½</td>
<td></td>
</tr>
</tbody>
</table>
Pokémon variant

type ptype =
    TNormal | TFire | TWater

type peff =
    ENormal  | ENotVery | ESuper

let eff_to_float = function
    | ENormal   -> 1.0
    | ENotVery  -> 0.5
    | ESuper    -> 2.0
RECORDS AND TUPLES
Records

- Several pieces of data glued together
- A record contains several named fields
- Before you can use a record, must define a record type:  *Why? Clean type inference.*

```typescript
    type mon = {name: string; hp : int; ptype: ptype}
```
Records

• To build a record:
  – Write a record expression:
    \{name="Charmander"; hp=39; ptype=TFire\}
  – Order of fields doesn’t matter:
    \{name="Charmander"; ptype=Tfire; hp=39\}
    is equivalent

• To access a record's field:  \texttt{r.hp}

• Or can use pattern matching with record patterns:
  \{f1=p1; \ldots; fn=pn\}  

  \textit{I guess you could call that record breaking}
Pattern matching records

(* OK *)

let get_hp m =
    match m with
    | {name=n; hp=h; ptype=t} -> h

(* better *)

let get_hp m =
    match m with
    | {name=_; hp=h; ptype=_} -> h
Advanced pattern matching records

(* better *)

```ml
let get_hp m =
  match m with
  | {name; hp; ptype} -> hp
```

(* better *)

```ml
let get_hp m =
  match m with
  | {hp} -> hp
```

(* best *)

```ml
let get_hp m = m.hp
```
By name vs. by position

• Fields of record are identified by name
  – order we write fields in expression is irrelevant

• Opposite choice: identify by position
Tuples

• Several pieces of data glued together
• A tuple contains several components
• (Don't have to define tuple type before use)

e.g.,
• (1,2,10)
• 1,2,10
• (true, "Hello")
• ([1;2;3], (0.5,'X'))
Tuple types

• \((1, 2, 10)\) : int*int*int
• \(1, 2, 10\) : int*int*int
• \((\text{true}, \ "Hello")\) : bool*string
• \([[1; 2; 3], (0.5, 'X'))\)
  : int list * (float*char)
Tuples

- 2-tuple: pair
- 3-tuple: triple
- beyond that: maybe better to use records

We need language constructs to build tuples and to access the components
- Building is easy: just write the tuple, as before
- Accessing uses pattern matching...
Accessing tuples

New kind of pattern, the **tuple pattern**: \((p_1, \ldots, p_n)\)

```ml
match (1,2,3) with
| (x,y,z) -> x+y+z

(* ==> 6 *)
```

```ml
let thrd t =
    match t with
    | (x,y,z) -> z

(* thrd : 'a*'b*'c -> 'c *)
```

Note: we never needed more than one branch in the match expression...
Pattern matching without match

(* OK *)
let thrd t =
  match t with
  | (x,y,z) -> z

(* good *)
let thrd t =
  let (x,y,z) = t in z

(* better *)
let thrd t =
  let (_,_,z) = t in z

(* best *)
let thrd (_,_,z) = z
Extended syntax for let

• Previously we had this syntax:
  - `let x = e1 in e2`
  - `let [rec] f x1 ... xn = e1 in e2`

• Everywhere we had a variable identifier x, we can really use a pattern!
  - `let p = e1 in e2`
  - `let [rec] f p1 ... pn = e1 in e2`

• Old syntax is just a special case of new syntax, since a variable identifier is a pattern
Pattern matching arguments

(* OK *)

```
let sum_triple t =
    let (x,y,z) = t
    in x+y+z
```

(* better *)

```
let sum_triple (x,y,z) = x+y+z
```

Note how that last version looks syntactically like a function in C/Java!
Accessing pairs

Built-in *projection functions* for first and second components:

```plaintext
let fst (x,_)  =  x
let snd (_,y)  =  y
```
Question

What is the type of this expression?

```haskell
let (x,y) = snd("big",("red",42))
  in (42,y)
```

A. \{x:string; y:int\}
B. int*int
C. string*int
D. int*string
E. string*(string*int)
What is the type of this expression?

```plaintext
let (x, y) = snd("big", ("red", 42))
in (42, y)
```

A. \{x:string; y:int\}
B. int*int
C. string*int
D. int*string
E. string*(string*int)
Pokémon effectiveness
Pokémon effectiveness

let eff = function

| (TFire,TFire)    | ->  ENotVery |
| (TWater,TWater)  | ->  ENotVery |
| (TFire,TWater)   | ->  ENotVery |
| (TWater,TFire)   | ->  ESuper   |
| _                | ->  ENormal  |
Semantics of tuples and records

Straightforward: see the notes, and slides at the end of this lecture
Upcoming events

• [Mon] Labor Day: no classes
• [Tue] recitations cancelled; lecture does meet
• [next Wed] A0 due

This is record breaking.

THIS IS 3110
Record expressions

• Syntax: \{f_1 = e_1; \ldots; f_n = e_n\}

• Evaluation:
  – If \(e_1\) evaluates to \(v_1\), and ... \(e_n\) evaluates to \(v_n\)
  – Then \{f_1 = e_1; \ldots; f_n = e_n\} evaluates to \{f_1 = v_1, \ldots, f_n = v_n\}
  – Result is a record value

• Type-checking:
  – If \(e_1 : t_1\) and \(e_2 : t_2\) and ... \(e_n : t_n\),
  – and if \(t\) is a defined type of the form \{f_1 : t_1, \ldots, f_n : t_n\}
  – then \{f_1 = e_1; \ldots; f_n = e_n\} : t
Record field access

- **Syntax:** \( e.f \)

- **Evaluation:**
  - If \( e \) evaluates to \( \{ f = v, \ldots \} \)
  - Then \( e.f \) evaluates to \( v \)

- **Type-checking:**
  - If \( e : t_1 \)
  - and if \( t_1 \) is a defined type of the form \( \{ f : t_2, \ldots \} \)
  - then \( e.f : t_2 \)
Evaluation notation

We keep writing statements like:
If $e$ evaluates to $\{f = v, \ldots\}$ then $e.f$ evaluates to $v$

Let's introduce a shorthand notation:
• Instead of "$e$ evaluates to $v$"
• Write "$e \Rightarrow v$"

So we can now write:
If $e \Rightarrow \{f = v, \ldots\}$ then $e.f \Rightarrow v$
Building tuples

• Syntax: \((e_1, e_2, \ldots, e_n)\)
  – parens are optional

• Evaluation:
  – If \(e_i \Rightarrow v_i\)
  – Then \((e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n)\)
  – A tuple of values is itself a value

• Type-checking:
  – If \(e_i : t_i\)
  – then \((e_1, \ldots, e_n) : t_1 \ast \ldots \ast t_n\)