Data Types

Prof. Clarkson
Fall 2016

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
Question

What do you think of A1?

A. It's a mystery.
B. It's a puzzle.
C. It's a riddle.
D. It's a conundrum.
E. It's an enigma.
• Please **have fun** and enjoy building the Enigma

• Please review the **late policy** in the course syllabus
  – sliding scale of penalty based on days late
  – *deadline* is the time by which you must successfully upload your solution files to CMS and confirm that CMS has recorded the correct versions of those files

• Please review the **academic integrity** policy in the course syllabus
  – we use MOSS to detect copying of code; *it works*
  – cite your sources (people, URLs)
  – don't claim other people's ideas/code as your own – that is a violation of AI and will lead to prosecution

• Please don't try to submit by email, regardless of reason
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*

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

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

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

```plaintext
let x = 1+4 in x*3
---> Evaluate e1 to a value v1

let x = 5 in x*3
---> Substitute v1 for x in e2, yielding a new expression e2'

  5*3

---> Evaluate e2' to v2

  15

  Result of evaluation is v2
```
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. 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
Anonymous functions

These two expressions are **syntactically different** but **semantically equivalent**:

\[
\text{let } x = 2 \text{ in } x+1
\]
\[
(\text{fun } x \rightarrow x+1) \ 2
\]

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

Syntax:

```
let x = e
```

Implicitly, “in rest of what you type”

E.g., you type:

```
let a="big";;  
let b="red";;  
let c=a^b;;   
```

OCaml understands as

```
let a="big" in
let b="red" in
let c=a^b in...
```
VARIANTS
Variant

```ocaml
type day = Sun | Mon | Tue | Wed 
            | Thu | Fri | Sat

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

Syntax: \texttt{type t = C1 | \ldots | Cn}

the \texttt{Ci} are called \textit{constructors}

\textbf{Evaluation:} a constructor is already a value

\textbf{Type checking:} \texttt{Ci : t}

\textbf{Destructing:} use pattern matching; constructor name is a pattern
Pokémon variant
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 *construct* 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 *destruct* and access record's field:  \( r\cdot hp \)

• Or can use pattern matching with *record patterns*:
  \( \{f1=p1; \ldots; fn=pn\} \)
  
  I guess you could call that record breaking
Pattern matching records

(* OK *)

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

(* better *)

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

(* better *)

```ml
let get_hp m =
    match t 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'))*
Tuples

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

We need language constructs to \textit{construct} tuples and to \textit{destruct} into pieces
• Construction is easy: just write the tuple, as before
• Destruction uses pattern matching...
Destructing 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!
Destructing pairs

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

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

What is the type of this expression?

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

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

- Normal: 1/2, 1/2
- Fire: 2, 1/2
- Water: 1/2, 2
Pokémon effectiveness

```plaintext
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

• [now] Questions about lecture and course logistics have priority over questions about content of A1
• [now-Wed] About **100 hours** of consulting scheduled to help you
• [Wed] A1 due

*This is record breaking.*

**THIS IS 3110**
SYNTAX AND SEMANTICS
Record expressions

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

• **Evaluation:**
  - If \(e_1\) evaluates to \(v_1\), and \(\ldots\) \(e_n\) 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 \(\ldots\) \(e_n : t_n\),
  - 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 \( e : t1 \)
  – and if \( t1 \) is a defined type of the form \{ \( f : t2, \ldots \) \}
  – then \( e.f : t2 \)
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 "\( e \implies v \)"

So we can now write:
If \( e \implies \{ f = v, \ldots \} \) then \( e.f \implies v \)
Constructing 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 \((e_1,\ldots,e_n) : t_1 \times \ldots \times t_n\)
  – A new kind of type, the product type