CS 3110

Data Types

Guest Lecture: Andrew Myers

Spring 2018

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

```
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:

```
let x = e1 in e2
```

`x` is an *identifier*

`e1` is the *binding expression*

`e2` is the *body expression*

`let x = e1 in e2` is itself an expression

`let` expressions

```
let x = e1 in e2
```

Evaluation:

- Evaluate **`e1`** to a value **`v1`**
- Substitute **`v1`** for **`x`** in **`e2`**, yielding a new expression **`e2'`**
- Evaluate **`e2'`** to **`v2`**
- Result of evaluation is **`v2`**

Let expressions

```
  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:

```
let x = 2 in x+1
(fun x -> 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;;
```

Toplevel understands as

```
let a="big" in
let b="red" in
let c=a^b in…
```

VARIANTS

Variant

```
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
```

Building and accessing variants

Syntax: `type t = C1 | ... | Cn`

the **`Ci`** are called *constructors*

Evaluation: a constructor is already a value

Type checking: `Ci : t`

Accessing: use pattern matching; constructor name is a pattern

Pokémon variant

  

DEFENSE → ATTACK ⤵	NOR	FIR	WAT	E
NORMAL				
FIRE		½	½	
WATER		2	½	

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.*

```
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: **`r.hp`**
- Or can use pattern matching with **record patterns**:
  `{f1=p1; ...; fn=pn}` *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 *)
let get_hp m =
  match m with
  | {name; hp; ptype} -> hp

(* better *)
let get_hp m =
  match m with
  | {hp} -> hp

(* best *)
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`
- `(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**: **`(p1, ..., pn)`**

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

(* ==> 6 *)

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:

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

Question

What is the type of this expression?

```
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?

```
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

DEFENSE → ATTACK ↓	NOR	FIR	WAT	E
NORMAL				
FIRE		½	½	
WATER		2	½	

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

- [Wed] A0 due

FOR RECITATION

Record expressions

- Syntax: `{f1 = e1; …; fn = en}`
- Evaluation:
  - If `e1` evaluates to `v1`, and … `en` evaluates to `vn`
  - Then `{f1 = e1; …; fn = en}` evaluates to `{f1 = v1, …, fn = vn}`
  - Result is a *record value*
- Type-checking:
  - If `e1` : `t1` and `e2` : `t2` and … `en` : `tn`,
  - and if `t` is a defined type of the form `{f1:t1, …, fn:tn}`
  - then `{f1 = e1; …; fn = en}: t`

Record field access

- Syntax: `e.f`
- Evaluation:
  - If `e` evaluates to `{f = v, …}`
  - Then `e.f` evaluates to `v`
- Type-checking:
  - If `e` : `t1`
  - and if `t1` is a defined type of the form `{f:t2, …}`
  - then `e.f` : `t2`

Evaluation notation

We keep writing statements like:
If `e` evaluates to `{f = v, …}` then `e.f` evaluates to `v`

Let's introduce a shorthand notation:

- Instead of "`e` evaluates to `v`"
- write "`e ==> v`"

So we can now write:
If `e ==> {f = v, …}` then `e.f ==> v`

Building tuples

- Syntax: `(e1,e2,...,en)`
  - parens are optional
- Evaluation:
  - If `ei ==> vi`
  - Then `(e1,...,en) ==> (v1,...,vn)`
  - A tuple of values is itself a value
- Type-checking:
  - If `ei:ti`
  - then `(e1,...,en):t1*...*tn`