User-defined Data Types

Prof. Clarkson
Summer 2016

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

Previously in 3110:

• **Functions:**
  – writing them, binding variables in them,
  – recursive, anonymous, higher-order
  – map and fold

Today:

• Turn attention to **data**
• Ways to define your own data types: records, tuples, variants
Record definition

• A **record** contains several named **fields**
• Before you can use a record, must **define** a record type:

```go
type time = {hour: int; min: int; ampm: string}
```

• To **build** a record:
  – Write a record expression:
    ```go
    {hour=10; min=10; ampm="am"}
    ```
  – Order of fields doesn’t matter:
    ```go
    {min=10; hour=10; ampm="am"} is equivalent
    ```

• To **access** record's field:  `r.hour`
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 $e.f \implies v$
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**
  — e.g., “Would the student named NN. step forward?”
  vs. “Would the student in seat $n$ step forward?”

• You’re accustomed to both:
  — Java object fields accessed by name
  — Java method arguments passed by position
    (but accessed in method body by name)

• OCaml has something you might not have seen:
  — A kind of heterogeneous data accessed by position
PAIRS AND TUPLES
Pairs

A **pair** of data: two pieces of data glued together

e.g.,

- \((1, 2)\)
- \((\text{true}, "Hello")\)
- \(([1; 2; 3], 0.5)\)

We need language constructs to *build* pairs and to *access* the pieces...
Pairs: building

• Syntax: \((e_1, e_2)\)

• Evaluation:
  – If \(e_1 \implies v_1\) and \(e_2 \implies v_2\)
  – Then \((e_1, e_2) \implies (v_1, v_2)\)
  – A pair of values is itself a value

• Type-checking:
  – If \(e_1 : t_1\) and \(e_2 : t_2\),
  – then \((e_1, e_2) : t_1*t_2\)
  – A new kind of type, the **product type**
Pairs: accessing

- **Syntax**: \texttt{fst e} and \texttt{snd e}
  
  *Projection functions*

- **Evaluation**:
  - If \( e \Rightarrow (v_1,v_2) \)
  - then \( \text{fst} \ e \Rightarrow v_1 \)
  - and \( \text{snd} \ e \Rightarrow v_2 \)

- **Type-checking**:
  - If \( e : \text{ta}*\text{tb} \),
  - then \( \text{fst} \ e \) has type \( \text{ta} \)
  - and \( \text{snd} \ e \) has type \( \text{tb} \)
Tuples

Actually, you can have *tuples* with more than two parts
- A new feature: a generalization of pairs
- Syntax, semantics are straightforward, except for projection...

• \((e_1, e_2, \ldots, e_n)\)
• \(t_1 * t_2 * \ldots * t_n\)
• \(\text{fst } e, \text{ snd } e, \text{ ???}\)

Instead of generalizing projection functions, use *pattern matching*...

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

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 \( x = e_1 \) in \( e_2 \)
  – let \([\text{rec}]\) \( f \) \( x_1 \ldots x_n = e_1 \) in \( e_2 \)

• Everywhere we had a variable identifier \( x \), we can really use a pattern!
  – let \( p = e_1 \) in \( e_2 \)
  – let \([\text{rec}]\) \( f \) \( p_1 \ldots p_n = e_1 \) in \( e_2 \)

• 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!
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)
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)`
Unit

• Can actually have a tuple () with no components whatsoever
  – Think of it as a degenerate tuple
  – Or, like a Boolean that can only have one value

• “Unit” is
  – a value written ()
  – and a type written unit
Pattern matching records

(* OK *)
let get_hour t =
  match t with
  | {hour=h; min=m; ampm=s} -> h

(* better *)
let get_hour t =
  match t with
  | {hour=h; min=_; ampm=_} -> h

(* better *)
let get_hour t =
  match t with
  | {hour; min; ampm} -> hour

(* better *)
let get_hour t =
  match t with
  | {hour} -> hour

(* better *)
let get_hour {hour} = hour

(* best *)
let get_hour t = t.hour

New kind of pattern, the record pattern:
{f1[=p1]; ...; fn[=pn]}
By name vs. by position, again

How to choose between coding \((4, 7, 9)\) and \(\{ f=4; g=7; h=9 \}\)?

- Tuples are syntactically **shorter**
- Records are **self-documenting**
- For many (3? 4? 5?) fields, a record is usually a better choice
VARIANTS
Variant

define type day = Sun | Mon | Tue | Wed
| Thu | Fri | Sat

define let day_to_int d =
  match d with
  | Sun  -> 1
  | Mon  -> 2
  | Tue  -> 3
  | Wed  -> 4
  | Thu  -> 5
  | Fri  -> 6
  | Sat  -> 7
Building and accessing variants

Syntax: \texttt{type } t = C_1 \mid \ldots \mid C_n

the \texttt{Ci} are called \textit{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</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORMAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRE</td>
<td>½</td>
<td></td>
<td>½</td>
<td></td>
</tr>
<tr>
<td>WATER</td>
<td>2</td>
<td>½</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- **NORMAL**
- **FIRE**
- **WATER**
- **DEFENSE**
- **ATTACK**
Pokémon variant

```
let eff_to_float = function
    | ENormal   -> 1.0
    | ENotVery  -> 0.5
    | ESuper    -> 2.0

let eff_att_vs_def : ptype*ptype -> peff = function
    | (TFire,TFire)   -> ENotVery
    | (TWater,TWater) -> ENotVery
    | (TFire,TWater)  -> ENotVery
    | (TWater,TFire)  -> ESuper
    | _               -> ENormal
```
Argument order: records

If you are worried about clients of function forgetting which order to pass arguments in tuple, use a record:

```plaintext
type att_def = {att:ptype; def:ptype}

let eff_att_vs_def : att_def -> peff = function
  | {att=TFire;def=TFire}   -> ENotVery
  | {att=TWater;def=TWater} -> ENotVery
  | {att=TFire;def=TWater}  -> ENotVery
  | {att=TWater;def=TFire}  -> ESuper
  | _                      -> ENormal
```