CSE341: Programming Languages

Lecture 4

Records (“each of”), Datatypes (“one of”), Case Expressions

Dan Grossman

Fall 2011
Review

• Done: functions, tuples, lists, local bindings, options

• Done: syntax vs. semantics, environments, mutation-free

• Today: Focus on compound types
  – New feature: records
    • New concept: syntactic sugar (tuples are records)
  – New features: datatypes, constructors, case expressions
How to build bigger types

- Already know:
  - Have various base types like int bool unit char
  - Ways to build (nested) compound types: tuples, lists, options

- Today: more ways to build compound types

- First: 3 most important type building blocks in any language
  - “Each of”: A t value contains values of each of t1 t2 ... tn
  - “One of”: A t value contains values of one of t1 t2 ... tn
  - “Self reference”: A t value can refer to other t values

Remarkable: A lot of data can be described with just these building blocks

Note: These are not the common names for these concepts
Examples

- Tuples build each-of types
  - `int * bool` contains an `int and a bool`

- Options build one-of types
  - `int option` contains an `int or it contains no data`

- Lists use all three building blocks
  - `int list` contains an `int and another int list or it contains no data`

- And of course we can nest compound types
  - `((int * int) option) * (int list list))` option
Rest of today

• Another way to build each-of types in ML
  – Records: have named fields
  – Connection to tuples and idea of syntactic sugar

• A way to build and use our own one-of types in ML
  – For example, a type that contains and int or a string
  – Will lead to pattern-matching (more next lecture), one of ML’s coolest and strangest-to-Java-programmers features
  – How OOP does one-of types discussed later in course
Records

Record values have fields (any name) holding values

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

Record types have fields (and name) holding types

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

The order of fields in a record value or type never matters
– REPL alphabetizes fields just for consistency

Building records:

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

Accessing components:

\#myfieldname e

(Evaluation rules and type-checking as expected)
Example

{\text{name} = \text{"Amelia"}, \text{id} = 41123 - 12}

Evaluates to

{\text{id} = 41111, \text{name} = \text{"Amelia"}}

And has type

{\text{id} : \text{int}, \text{name} : \text{string}}

If some expression such as a variable \( x \) has this type, then get fields with:

\#id \ x \quad \#name \ x

Note we didn’t have to declare any record types
– The same program could also make a

{\text{id} = \text{true}, \text{ego} = \text{false}} \text{ of type } \{\text{id} : \text{bool}, \text{ego} : \text{bool}\}
By name vs. by position

• Little difference between \((4, 7, 9)\) and \(\{f=4, g=7, h=9\}\)
  – Tuples a little shorter
  – Records a little easier to remember “what is where”
  – Generally a matter of taste, but for many (6? 8? 12?) fields, a record is usually a better choice

• A common decision for a construct’s syntax is whether to refer to things *by position* (as in tuples) or *by some (field) name* (as with records)
  – A common hybrid is like with Java method arguments (and ML functions as used so far):
    • Caller uses *position*
    • Callee uses *variables*
    • Could totally do it differently; some languages have
The truth about tuples

Last week we gave tuples syntax, type-checking rules, and evaluation rules

But we could have done this instead:
- Tuple syntax is just a different way to write certain records
- \((e_1, \ldots, e_n)\) is another way of writing \(\{1=e_1, \ldots, n=e_n\}\)
- \(t_1*\ldots*t_n\) is another way of writing \(\{1:t_1, \ldots, n:t_n\}\)
- In other words, records with field names 1, 2, …

In fact, this is how ML actually defines tuples
- Other than special syntax in programs and printing, they don’t exist
- You really can write \(\{1=4, 2=7, 3=9\}\), but it’s bad style
"Tuples are just syntactic sugar for records with fields named 1, 2, … n"

- **Syntactic**: Can describe the semantics entirely by the corresponding record syntax

- **Sugar**: They make the language sweeter 😊

Will see many more examples of syntactic sugar

- They simplify *understanding* the language
- They simplify *implementing* the language

Why? Because there are fewer semantics to worry about even though we have the syntactic convenience of tuples
Datatype bindings

A “strange” (?) and totally awesome (!) way to make one-of types:

– A datatype binding

```haskell
datatype mytype = TwoInts of int * int |
                  Str of string |
                  Pizza
```

• Adds a new type `mytype` to the environment
• Adds *constructors* to the environment: `TwoInts`, `Str`, and `Pizza`
• A constructor is (among other things), a function that makes values of the new type (or is a value of the new type):
  
  – `TwoInts : int * int -> mytype`
  – `Str : string -> mytype`
  – `Pizza : mytype`
The values we make

```plaintext
datatype mytype = TwoInts of int * int
           | Str of string
           | Pizza
```

• Any value of type `mytype` is made from *one of* the constructors
• The value contains:
  – A “tag” for “which constructor” (e.g., `TwoInts`)
  – The corresponding data (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
Using them

So we know how to build datatype values; need to access them.

There are two aspects to accessing a datatype value:

1. Check what variant it is (what constructor made it)
2. Extract the data (if that variant has any)

Notice how our other one-of types used functions for this:

- `null` and `iSome` check variants
- `hd`, `tl`, and `valOf` extract data (raise exception on wrong variant)

ML could have done the same for datatype bindings

- For example, functions like “isStr” and “getStrData”
- Instead it did something better
Case

ML combines the two aspects of accessing a one-of value with a case expression and pattern-matching

– Pattern-matching much more general/powerful (lecture 5)

Example:

```haskell
fun f x = (* f has type mytype -> int *)
case x of
  Pizza => 3
| TwoInts(i1,i2) => i1+i2
| Str s => String.size s
```

• A multi-branch conditional to pick branch based on variant
• Extracts data and binds to variables local to that branch
• Type-checking: all branches must have same type
• Evaluation: evaluate between case … of and right branch
Patterns

In general the syntax is:

```
case e0 of
  p1  => e1
|  p2  => e2
|    ...
|  pn  => en
```

For today, each pattern is a constructor name followed by the right number of variables (i.e., C or C x or C (x, y) or …)

- Syntactically most patterns (all today) look like expressions
- But patterns are not expressions
  - We do not evaluate them
  - We see if the result of e0 matches them
Why this way is better

0. You can use pattern-matching to write your own testing and data-extractions functions if you must
   – But don’t do that on your homework

1. You can’t forget a case (inexhaustive pattern-match a warning)

2. You can’t duplicate a case (a type-checking error)

3. You won’t forget to test the variant correctly and get an exception (like `hd []`)

4. Pattern-matching can be generalized and made more powerful, leading to elegant and concise code
Useful examples

Let’s fix the fact that our only example datatype so far was silly…

• Enumerations, including carrying other data

```plaintext
datatype suit = Club | Diamond | Heart | Spade

datatype card_value = Jack | Queen | King
                  | Ace | Num of int
```

• Alternate ways of representing data about things (or people 😊)

```plaintext
datatype id = StudentNum of int
            | Name of string
            * (string option)
            * string
```
Don’t do this

Unfortunately, bad training and languages that make one-of types inconvenient lead to common *bad style* where each-of types are used where one-of types are the right tool

```csharp
(* use the studen_num and ignore other fields unless the student_num is ~1 *)
{
    studen_num : int,
    first      : string,
    middle     : string option,
    last       : string
}
```

- Approach gives up all the benefits of the language enforcing every value is one variant, you don’t forget branches, etc.
- And it makes it less clear what you are doing
That said…

But if instead, the point is that every “person” in your program has a name and maybe a student number, then each-of is the way to go:

```haskell
{ student_num : int option,
  first       : string,
  middle      : string option,
  last        : string }
```
Expression Trees

A more exciting (?) example of a datatype, using self-reference

```plaintext
datatype exp = Constant of int
  | Negate of exp
  | Add of exp * exp
  | Multiply of exp * exp
```

An expression in ML of type `exp`:

```
Add (Constant (10+9), Negate (Constant 4))
```

How to picture the resulting value in your head:

```
Add
  /
Constant
  /
19
  /
Negate
  /
  /
Constant
  /
  /
4
```
Recursion

Not surprising:
Functions over recursive datatypes are usually recursive

```haskell
fun eval e =
case e of
  Constant i => i
  | Negate e2 => ~(eval e2)
  | Add(e1,e2) => (eval e1) + (eval e2)
  | Multiply(e1,e2) => (eval e1) * (eval e2)
```