CSE341: Programming Languages

Lecture 4

Records, Datatypes, Case Expressions

Brett Wortzman

Spring 2020
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

• Coming soon: 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 this Lecture

• 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 an *int* or a *string*
  – Will lead to *pattern-matching*, one of ML’s coolest and strangest-to-Java-programmers features

• Later in course: How OOP does one-of types
  – Key contrast with procedural and functional programming
**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:

\texttt{#myfieldname e}

(Evaluation rules and type-checking as expected)
Example

{name = "Jakob", id = 4 - 3}

Evaluates to

{id = 1, name = "Jakob"}

And has type

{id : int, name : string}

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

\[ #id \ x \quad #\text{name} \ x \]

Note we did not have to declare any record types

– The same program could also make a

\[ \{\text{id}=\text{true}, \text{ego}=\text{false}\} \] of type \{id:bool,ego: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

Previous lecture 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 \(\{1=e_1,\ldots,n=e_n\}\)
– \(t_1\ast\ldots\ast t_n\) 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
**Syntactic sugar**

“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

Another example we saw: **andalso** and **orelse** vs. **if** **then** **else**
Five different things

1. Syntax: How do you write language constructs?
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a well-known project) provide “standard”? (E.g., file access, data structures)
   - Questionably part of the language
5. Tools: What do language implementations provide to make your job easier? (E.g., REPL, debugger, code formatter, …)
   - Not actually part of the language

These are 5 separate issues
   - In practice, all are essential for good programmers
   - Many people confuse them, but shouldn’t
Our Focus

This course focuses on semantics and idioms

• Syntax is usually uninteresting
  – A fact to learn, like “The American Civil War ended in 1865”
  – People obsess over subjective preferences

• Libraries and tools crucial, but often learn new ones “on the job”
  – We are learning semantics and how to use that knowledge to understand all software and employ appropriate idioms
  – By avoiding most libraries/tools, our languages may look “silly” but so would any language used this way
Datatype bindings

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

```
datatype food = Pizza of string
  | Burger of int * bool
  | Salad
```

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

```haskell
datatype food = Pizza of string
             | Burger of int * bool
             | Salad
```

- Any value of type `food` is made from *one of* the constructors
- The value contains:
  - A “tag” for “which constructor” (e.g., `Pizza`)
  - The corresponding data (e.g., "cheese")

- Examples:
  - `Pizza(if true then "cheese" else "pepperoni")` evaluates to `Pizza("cheese")`
  - `Burger(1+2,3<5)` evaluates to `Burger(3,true)`
  - `Salad` 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 `isSome` 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 “`isPizza`” and “`getPizzaData`”
  – 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
(* describe has type food -> string *)
fun describe (f : food) =
  case f of
    Salad => "salad"
  | Pizza topp => topp ^ " pizza"
  | Burger (n, cheese) => Int.toString(n) ^
    "x burger" ^ (if cheese then " with cheese" else "")
```

- 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 the 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 do not do that on your homework

1. You cannot forget a case (inexhaustive pattern-match warning)

2. You cannot duplicate a case (a type-checking error)

3. You will not 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