CSE341, Fall 2011, Lecture 4 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture covers *records*, which, as explained below, are very, very similar to tuples. More importantly, it covers *datatypes*, including *constructors* (for making them) and *case expressions* (for accessing them), which are less like anything you have seen before. Behind the ML constructs of records and datatypes are some fundamental concepts that we describe first.

Conceptual Ways to Build New Types

Programming languages have *base types*, like `int`, `bool`, and `unit` and *compound types*, which are types that contain other types in their definition. We have already seen ways to build compound types in ML, namely by using tuple types, list types, and option types. This lecture will describe new ways to build even more flexible compound types and to give names to our new types. In creating a compound type, there are really only three essential building blocks. Any decent programming language will provide these building blocks in some way:[1]

- "Each-of": A compound type `t` describes values that contain *each of* values of type `t1`, `t2`, ..., ***and*** `tn`.
- "One-of": A compound type `t` describes values that contain a value of *one of* the types `t1`, `t2`, ..., ***or*** `tn`.
- "Self-reference": A compound type `t` may refer to itself in its definition in order to describe recursive data structures like lists and trees.

Each-of types are the most familiar to most programmers. Tuples are an example: `int * bool` describes values that contain an `int` *and* a `bool`. A Java class with fields is also an each-of sort of thing.

One-of types are also very common but unfortunately are not emphasized as much in many introductory programming courses. `int option` is a simple example: A value of this type contains an `int` *or* it does not. For a type that contains an `int` *or* a `bool` in ML, we need datatype bindings, the main topic of this lecture. In object-oriented languages with classes like Java, one-of types are achieved with subclassing, but that is the topic of a later lecture.

Self-reference allows types to describe recursive data structures. This is useful in combination with each-of and one-of types. For example, `int list` describes values that either contain nothing *or* contain an `int` *and* another `int list`. A list of integers in any programming language would be described in terms of *or*, *and*, and *self-reference* because that is what it means to be a list of integers.

Naturally, since compound types can nest, we can have any nesting of each-of, one-of, and self-reference. For example, consider the type `(int * bool) list list * (int option) list * bool`.

Records: Another Approach to "Each-of" Types

Record types are "each-of" types where each component is a *named field*. For example, the type `{foo : int, bar : int*bool, baz : bool*int}` describes records with three fields named `foo`, `bar`, and `baz`. This is just a new sort of type, just like tuple types were new in Lecture 2.

[1] As a matter of jargon you do not need to know, the terms "each-of types," "one-of types," and "self-reference types" are not standard – they are just good ways to think about the concepts. Usually people just use constructs from a particular language like "tuples" when they are talking about the ideas. Programming-language researchers use the terms "product types," "sum types," and "recursive types." Why product and sum? It is related to the fact that in Boolean algebra where 0 is false and 1 is true, *and* works like multiply and *or* works like addition.

A *record expression* builds a *record value*. For example, the expression `{bar = (1+2,true andalso true), foo = 3+4, baz = (false,9) }` would evaluate to the record value `{bar = (3,true), foo = 7, baz = (false,9)}`, which can have type `{foo : int, bar : int*bool, baz : bool*int}` because the order of fields never matters (we use the field names instead). In general the syntax for a record expression is `{f1 = e1, ..., fn = en}` where as always each `e` can be any expression. Here each `f` can be any field name, which is basically any sequence of letters or numbers.

In ML, we do not have to declare that we want a record type with particular field names and field types — we just write down a record expression and the type-checker gives it the right type. The type-checking rules for record expressions are not surprising: Type-check each expression to get some type `ti` and then build the record type that has all the right fields with the right types. *Because the order of field names never matters, the REPL always alphabetizes them when printing just for consistency.*

The evaluation rules for record expressions are analogous: Evaluate each expression to a value and create the corresponding record value.

Now that we know how to build record values, we need a way to access their pieces. For now, we will use `#foo e` where `foo` is a field name. Type-checking requires `e` has a record type with a field named `foo`, and if this field has type `t`, then that is the type of `#foo e`. Evaluation evaluates `e` to a record value and then produces the contents of the `foo` field.

By Name vs. By Position, Syntactic Sugar, and The Truth About Tuples

Records and tuples are *very* similar. They are both "each-of" constructs that allow any number of components. The only real difference is that records are "by name" and tuples are "by position." This means with records we build them and access their pieces by using field names, so the order we write the fields in a record expression does not matter. But tuples do not have field names, so we use the position (first, second, third, ...) to distinguish the components.

By name versus by position is a classic decision when designing a language construct or choosing which one to use, with each being more convenient in certain situations. As a rough guide, by position is simpler for a small number of components, but for larger compound types it becomes too difficult to remember which position is which.

Java method arguments (and ML function arguments as we have described them so far) actually take a hybrid approach: The method body uses variable *names* to refer to the different arguments, but the caller passes arguments by *position*. There are other languages where callers pass arguments by name.

Despite "by name vs. by position," records and tuples are still so similar that we can define tuples entirely in terms of records. Here is how:

- When you write `(e1,...,en)` it is another way of writing `{1=e1,...,n=en}`, i.e., a tuple expression is a record expression with field names 1, 2, ..., n.
- The type `t1 * ... * tn` is just another way of writing `{1:t1, ..., n:tn}`.
- Notice that `#1 e`, `#2 e`, etc. now already mean the right thing: get the contents of the field named 1, 2, etc.

In fact, this is how ML actually defines tuples: A tuple *is* a record. That is, all the syntax for tuples is just a convenient way to write down and use records. The REPL just always uses the tuple syntax where possible, so if you evaluate `{2=1+2, 1=3+4}` it will print the result as `(7,3)`. Using the tuple *syntax* is better style, but we did not need to give tuples their own *semantics*: we can instead use the "another way of writing" rules above and then reuse the semantics for records.

This is the first of many examples we will see of ***syntactic sugar***. We say, "tuples are just syntactic sugar for records with fields named 1, 2, ..., n." It is *syntactic* because we can describe everything we need about tuples in terms of equivalent record syntax. It is *sugar* because it makes the language sweeter. The term *syntatic sugar* is widely used. Syntactic sugar is a great way to keep the "key ideas" in a programming-language small (making it easier to implement) while giving programmers convenient ways to write things. Indeed, in Homework 1 we used tuples in every problem without knowing records existed even though tuples are records.

Datatype Bindings: Our Own "One-of" Types

We now introduce *datatype bindings*, our third kind of binding after variable bindings and function bindings. We start with a silly but simple example because it will help us see the many different aspects of a datatype binding. We can write:

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

Roughly, this defines a new type where values will either have an `int * int` or a `string` or nothing. Any value will also be "tagged" with information that lets us know which *variant* it is: These "tags," which we will call *constructors*, are `TwoInts`, `Str`, and `Pizza`. Two constructors could be used to tag the same type of underlying data; in fact this is common even though our example uses different types for each variant.

More precisely, the example above adds four things to the environment:

- A new type `mytype` that we can now use just like any other type
- Three *constructors* `TwoInts`, `Str`, and `Pizza`

A *constructor* is two different things. First, it is eiether a function for creating values of the new type (if the variant has `of t` for some type `t`) or it is actually a value of the new type (otherwise). In our example, `TwoInts` is a function of type `int*int -> mytype`, `Str` is a function of type `string->mytype`, and `Pizza` is a value of type `mytype`. Second, we use constructors in case-expressions as described further below.

So we know how to build values of type `mytype`: call the constructors (they are functions) with expressions of the right types (or just use the `Pizza` value). The result of these function calls are values that "know which variant they are" (they store a "tag") and have the underlying data passed to the constructor. The REPL represents these values like `TwoInts(3,4)` or `Str "hi"`.

What remains is a way to retrieve the pieces...

How ML Does *Not* Provide Access to Datatype Values

Given a value of type `mytype`, how can we access the data stored in it? First, *we need to find out which variant it is* since a value of type `mytype` might have been made from `TwoInts`, `Str`, or `Pizza` and this affects what data is available. Once we know what variant we have, then we can access the pieces, if any, that variant carries.

Recall how we have done this so for lists and options, which are also one-of types: We had functions for testing which variant we had (`null` or `isSome`) and functions for getting the pieces (`hd`, `tl`, or `valOf`), which raised exceptions if given arguments of the wrong variant.

ML could have taken the same approach for datatype bindings. For example, it could have taken our datatype definition above and added to the environment functions `isTwoInts`, `isStr`, and `isPizza` all of type `mytype->bool`. And it could have added functions like `getTwoInts` of type `mytype -> int*int` and `getStr` of type `mytype -> string`, which might raise exceptions.

But ML does not take this approach. Instead it does something better. You could write these functions yourself using the better thing, though it is usually poor style to do so. (In fact, after learning the better thing, we will no longer use the functions for lists and options the way we have been.)

How ML *Does* Provide Access to Datatype Values: Case Expressions

The better thing is a *case expression*. Here is a basic example for our example datatype binding:

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

In one sense, a case-expression is like a more powerful if-then-else expression: Like a conditional expression, it evaluates two of its subexpressions: first the expression between the `case` and `of` keywords and second the expression in the *first branch that matches*. But instead of having two branches (one for `true` and one for `false`), we can have one branch for each variant of our datatype (and the next lecture will generalize this further). Like conditional expressions, each branch's expression must have the same type (`int` in the example above) because the type-checker cannot know what branch will be used.

Each branch has the form `p => e` where `p` is a *pattern* and `e` is an expression, and we separate the branches with the `|` character. Patterns look like expressions, but do not think of them as expressions. Instead they are used to *match* against the result of evaluating the case's first expression (the part after `case`). This is why evaluating a case-expression is called *pattern-matching*.

For this lecture, we keep pattern-matching simple: Each pattern uses a different constructor and pattern-matching picks the branch with the "right one" given the expression after the word `case`. The result of evaluating that branch is the overall answer; no other branches are evaluated. For example, if `TwoInts(7,9)` is passed to `f`, then the second branch will be chosen.

That takes care of the "check the variant" part of using the one-of type, but pattern matching *also* takes care of the "get out the underlying data" part. Since `TwoInts` has two values it "carries", a pattern for it can (and, for now, must) use two variables (the `(i1,i2)`). As part of matching, the corresponding parts of the value (continuing our example, the 7 and the 9) are bound to `i1` and `i2` in the environment used to evaluate the corresponding right-hand side (the `i1+i2`). In this sense, pattern-matching is like a let-expression: It binds variables in a local scope. The type-checker knows what types these variables have because they were specified in the datatype binding that created the constructor used in the pattern.

Why are case-expressions better than functions for testing variants and extracting pieces?

- We can never "mess up" and try to extract something from the wrong variant. That is, we will not get exceptions like we get with `hd []`.
- If a case expression forgets a variant, then the type-checker will give a warning message. This indicates that evaluating the case-expression could find no matching branch, in which case it will raise an exception. If you have no such warnings, then you know this does not occur.
- If a case expression uses a variant twice, then the type-checker will give an error message since one of the branches could never possibly be used.
- If you still want functions like `null` and `hd`, you can easily write them yourself (but do not do so for your homework).
- Pattern-matching is so much more general and powerful that it will be the subject of the next lecture. (In fact, given how general it is, we will hold off giving the full syntax, type-checking rules, and evaluation rules.)

Useful Examples and Don't Use "Each-of" When You Want "One-of"

Let us now consider several examples where "one-of" types are useful, since so far we considered only a silly example.

First, they are good for enumerating a fixed set of options – and much better style than using, say, small integers. For example:

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

Many languages have support for this sort of *enumeration* including Java and C, but ML takes the next step of letting variants carry data, so we can do things like this:

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

We can then combine the two pieces with an each-of type[2]

```
type card = suit * card_value
```

One-of types are also useful when you have different data in different situations. For example, suppose you want to identify students by their id-numbers, but in case there are students that do not have one (perhaps they are new to the university), then you will use their full name instead. This datatype binding captures the idea directly:

```
datatype id = StudentNum of int
            | Name of string * (string option) * string
```

Unfortunately, this sort of example is one where programmers often show a profound lack of understanding of one-of typse and insist on using each-of types, which is like using a saw as a hammer (it works, but you are doing the wrong thing). Consider BAD code like this:

```
(* If student_num is -1, then use the other fields, otherwise ignore other fields *)
type bad_id = {student_num : int, first : string, middle : string option, last : string}
```

This approach requires all the code to follow the rules in the comment, with no help from the type-checker. It also wastes space, having fields in every record that should not be used.

On the other hand, each-of types are exactly the right approach if we want to store for each student their id-number (if they have one) *and* their full name:

```
type name_record = { student_num : int option,
                     first       : string,
                     middle      : string option,
                     last        : string }
```

Our last example is a data definition for arithmetic expressions containing constants, negations, additions, and multiplications.

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

[2]`type name = t` is a *type synonym*, which is not explained in this lecture.

Thanks to the self-reference, what this data definition really describes is *trees* where the leaves are integers and the internal nodes are either negations with one child, additions with two children or multiplications with two children. We can write a function that takes an exp and evaluates it:

```
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)
```

So this function call evaluates to 15:

```
eval (Add (Constant 19, Negate (Constant 4)))
```

Notice how constructors are just functions that we call with other expressions (often other values built from constructors).

There are many functions we might write over values of type exp and most of them will use pattern-matching and recursion in a similar way. For examples, other functions you could write that process exp values could compute:

- The largest constant in an expression
- A list of all the constants in an expression (use list append)
- A bool indicating whether there is at least one multiplication in the expression
- The number of addition expressions in an expression

Here is the last one:

```
fun number_of_adds e =
    case e of
        Constant i      => 0
      | Negate e2       => number_of_adds e2
      | Add(e1,e2)      => 1 + number_of_adds e1 + number_of_adds e2
      | Multiply(e1,e2) => number_of_adds e1 + number_of_adds e2
```