CSci 2041: Advanced Programming Principles
User Defined Types and Inductive Data

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

The Structure of Types and Expressions

All types and expressions in OCaml are determined by the collection of base types

These come in two varieties:

- Those that are system-defined

  We have seen several examples of these types already `int`, `float`, `bool`, `string`, tuples, lists

  Each of these types come with a set of values and mechanisms for constructing (complex) expressions

- those that are user-defined

  OCaml provides rather powerful mechanisms for defining such types

  Associated with these mechanisms are also the means for defining rich collections of values

Our next goal: understanding how these devices work

Take Two on What We Will Study

- We have seen some “builtin” types in OCaml and how to compute with them

  For example, `int`, `float`, `'a list`

- However, interesting programs usually want to deal with new kinds of data that are also recursive
  - Person records, that have items of information such as name, age, ssno, employment status, etc
  - Binary trees, that have nodes with data and possibly subtrees as components
  - Expressions in programs that are constructed from constants and variables using operators such as $+$ and $*$

- The questions we want to address about OCaml:
  - How do we build types to correspond to such data?
  - How do we construct values of such types?
  - How do we compute over such data?

Introducing New Type Names in OCaml

As a first, simple extension, we see that OCaml allows us to introduce new names for already known types

The structure of such a definition in OCaml

```
type <name> = <type-exp>
```

In such a declaration

- `type` is a "keyword" introducing a type declaration
- `<name>` is the name of the type being introduced
- `<type-exp>` is the type being given a name

Some example uses of such definitions

```
type intpair = int * int
type intlist = int list
```

Note that these definitions only provide abbreviations for known types; *they do not introduce any new types or values*

Exercise

Write down two values of each of the types defined below

```
type intandstr = int * string

type i_and_s_list = intandstr list
```

Parameterizing Type Abbreviations with Types

We often want to describe a type of a particular structure but with a choice of component types

We can do this by *parameterizing* the type abbreviation, e.g.

```
type 'a pair = 'a * 'a
```

Such a declaration gives us a "type function," called a *type constructor*, for producing real types, e.g.

```
(int pair)    (bool pair)    (int list) pair
```

Question: Can you identify builtin type constructors like this?

We can also have multiple type parameters, e.g.

```
type ('a, 'b) pair = 'a * 'b
```

Example use: `(int, bool) pair` to yield `int * bool`

Introducing New Values with Types

The type declarations we have seen till now have the form

```
type ('<n1>,...,'<nk>) <name> = <type-exp>
```

In this form

- We only introduce new ways of writing already known types
- We also do not change the set of values known by the system

We will now look at how we enrich the collection of known types as well as the set of known values

The brief story: in place of `<type-exp>` we describe the different ways to construct objects of the new type

Enumerated Types in OCaml

Sometimes we may want to introduce a type together with a finite number of *new* values

For example we may want a new type called *color* with the values *Red*, *Blue* and *Green*

We can do this through the following declaration in OCaml

```
type color = Red
           | Blue
           | Green
```

After this declaration, we will have the `color` type and three new *values* or, more precisely, *value constructors*

Note that this declaration identifies a type that is like an enumerated type in a language like C

Exercise

- Define a type `weekday` that has as values the constant `Mon`, ..., `Sun`
- Identify a type amongst the base types that is actually an enumerated type like `color`

Pattern Matching on User Defined Types

Once we have defined a new type with its values, OCaml automatically extends pattern matching to such a type

For example, we can define the following function

```
let isRed c =
      match c with
        | Red -> true
        | Blue -> false
        | Green -> false
```

Note that the pattern matching we have on boolean values is just a special case of this feature

Exercise

Define the function

```
isWorkDay : weekday -> bool
```

that returns `true` just in the case that the argument represents a day between Monday and Friday

Make sure to use pattern matching over the `weekday` type in your definition

Disjoint Unions in OCaml

Sometimes we may want to form a "marked" union of two *different* types

E.g., we may want to combine the `int` and `string` collections in a way that allows us to tell where the object comes from

We can do that in OCaml using the following type declaration

```
type intorstr = Int of int | Str of string
```

This definition actually introduces two new value constructors that are of *function* type

```
Int : int -> intorstr
Str : string -> intorstr
```

Thus, values of type `intorstr` are of the form `(Int _)` or `(Str _)`

Pattern Matching on “Disjoint Union” Types

Pattern matching lifts in the way one would expect also to disjoint union types

For example, suppose you want to sum up the integers in a list of type `(intorstr list)`

You can define the following function to do it

```
let rec sumList =
      function
        | [] -> 0
        | (hd :: tl) ->
              match hd with
                | (Int i) -> i + sumList tl
                | (Str s) -> sumList tl
```

Note that having to go through the “tag” makes sure we can never mess up the types in a disjoint union in OCaml

Exercise

Suppose we have the following types given to us

```
type coord = float * float

type circ_desc = coord * float
type tri_desc = coord * coord * coord
type sqr_desc = coord * coord * coord * coord
```

The last three are meant to give us the components that characterize a circle, a triangle and a rectangle, respectively

- Define a type `shape` in OCaml that is capable of representing any one of a circle, a triangle and a rectangle
- Define a function of the following type

  ```
  isRect : shape -> bool
  ```

  that returns `true` if its argument is a rectangle and `false` otherwise

Introducing Polymorphic Value Constructors

We can parameterize the type being defined and then use the type parameters in the type of the value constructor

This results in a new type constructor and *polymorphic* value constructors

Example

When searching a database using an index, we want to be able to return a value that

- provides what was found if the search was successful
- indicates that the search was unsuccessful otherwise

A new type that suits this purpose

```
'a maybe = Nothing | Just of 'a
```

This declaration actually gives us two *polymorphic* value constructors `Nothing` and `Just`

Exercise

Define a function `listHd` for finding the head of a list that also works on *empty* lists

Hint: think of returning something of a `maybe` type

The Option Type in OCaml

Actually, OCaml has a *builtin* type constructor like the `maybe` constructor

It is call `option` and you can think of it as being defined as follows

```
type 'a option = None | Some of 'a
```

Inductive Datatypes

A really useful form for a type is when all its data is built in one of a *fixed* number of ways, possibly using data of the *same* type

For example

- a list is built from a head element and another (smaller) list
- a binary tree is built from a root element and two (smaller) trees
- an arithmetic expression is built using an operator and some number of smaller arithmetic expressions

Moreover, these are the *only* ways to build such data

In OCaml, we can use the type definition mechanism we have already seen to build type and value constructors for such data

The magic: some data constructors take as arguments object of the type we are currently defining!

Analyzing the List Type

The list type constructor is actually paired with two value constructors:

- the (0 argument) constructor `[]` of type `'a list`
- the 1 argument constructor `::` of type
  `('a * 'a list) -> 'a list`

Note that the `::` constructor takes as argument the same type as the object it produces

It is this that gives lists their recursive structure

If OCaml did not already have lists, we could use our type declarations again to define them:

```
type 'a myList = Nil | Cons of 'a * 'a myList
```

Defining a Binary Tree Type

Binary trees over a given type of elements have the following structure

- They are *empty*, or
- They (are *nodes* that) consist of a data item of the designated type and two subtrees with the same type of elements

To represent them in OCaml, we need to define a type constructor and two corresponding value constructors:

```
type 'a btree = Empty
              | Node of ('a * 'a btree * 'a btree)
```

Note again the polymorphic and recursive nature of the value constructors

Exercise

Recall the type declaration for binary trees from the previous slide:

```
type 'a btree = Empty
              | Node of ('a * 'a btree * 'a btree)
```

- Draw pictures of two *different* integer binary trees
- For each of the trees you have drawn, write the OCaml expressions that would represent them

Computing Over Inductive Data

We have already seen how to define functions that work over *all* lists:

- We use pattern matching to distinguish between lists that have different structures
- We describe what needs to be done in each case, possibly using recursion over component lists

A concrete example of this:

```
let rec sumlist lst =
   match lst with
    | [] -> 0
    | (h::t) -> h + (sumlist t)
```

We can use the same approach over user defined inductive types because they to possess pattern-matching and recursion!

Exercise

Recall the definition of the `btree` type

```
type 'a btree = Empty
              | Node of ('a * 'a btree * 'a btree)
```

Define a function `sumTree` that adds up the numbers in an integer binary tree represented using these constructors

The Structure of Type Declarations (Summary)

These come in two forms

- Where they provide abbreviations for already known types

  ```
  type ('<n1>,...,'<nk>) <name> = <type-exp>
  ```

  Here the type variables `'<n1>`, ..., `'<nk>` can be used in the type `<type-exp>`

- Where they actually identify new types together with ways to construct objects of that type

  ```
  type ('<n1>,...,'<nk>) <name> =
                         <val1> | ... | <valj>
  ```

  Here each `<vali>` is of form

  `<name>` or `<name> of <type-exp>`

  where `'<n1>`, ..., `'<nk>` can be used in `<type-exp>`

Designing Representations of Data

Complex data of a particular category that we want to compute over typically have the following structure

- They can take one of a few different forms
- Each of these forms is made up from subcomponents

The main issue in designing good representations is understanding and articulating this structure

Once we have done that, OCaml gives us a natural way to capture the analysis in a type definition

If we are using some other language, the analysis and the organization of thinking around it is still crucial

Example: Treating Programs as Data

We will look at how we can represent and manipulate programs as data

This is something this is quite useful to do in practice

- We want to determine if a program is type correct (or even infer types for programs)
- We want to write interpreters for programs
- We want to write program transformers or compilers for programs

In all these cases, we need to have an internal representation of programs

More generally, this is an example of representing *symbolic data* that is central to AI and many other application areas

Treating Programs as Data (Example)

Will treat a fragment of OCaml expressions comprising

- Identifiers
- Constants such as integers, `true`, `false`, etc
- Arithmetic expressions constructed using the operators $+$, $-$, $*$, $/$
- Boolean valued expressions constructed using
  - comparison operators $<$, $>$, $=$ (on integers only)
  - boolean operators `not`, `and`, `or`
- *if-then-else* expressions that could yield boolean or integer results
- *let* expressions

We disallow polymorphism and all our expressions must be of either boolean or integer type

Concrete Versus Abstract Syntax

When we first think of programs, we view them as character streams, e.g.

```
(2 + 3) * x + 7
if (x < 4) then 2 + 3 else 17 * y
```

This is known as *concrete syntax*

However, by the time we manipulate them in our programs, we will assume their functional structure is known

```
plus(times(plus(2,3),x),7)
cond(less(x,4),plus(2,3),times(17,y))
```

This form is known as *abstract syntax*

Notice that we do not have any need to represent parentheses in abstract syntax

The process of going from concrete to abstract syntax is tackled by *parsing*, something we won't presently consider

How Many Types to Use?

Our expressions can be of integer or boolean types

One possibility: use two different types for these categories

However this poses problems

- We will need two different forms of *let* and *if-then-else*

  ```
  if (2 < 3) then true else false
  if (2 < 3) then 5 else 7
  let x = 3 in x + 5
  let x = true in x
  ```

- it requires some analysis to determine which form to use and, indeed to rule out cases like

  ```
  if (2 < 3) then true else 7
  ```

The more common approach: be neutral about types in representation and let programs do the analysis later

Designing an OCaml Type for Programs

Lets then pick the type `expr` for program expressions

We now only need to consider the different possibilities for expressions and design an `expr` case for each

- Identifiers, distiguished by their names

  We can capture all the information by the following

  ```
  Id of string
  ```

- Constants, that are integers or `true` or `false`

  For integers we can use `Int of int`

  For the booleans, we have two possibilities

  - We follow the treatment for integers (`Bool of bool`)
  - We use to designated constants `True` and `False`

  Lets do the latter for now

An OCaml Type for Programs (Contd)

- Arithmetic expressions

  For each of these we use a special value constructor

  The constructor will take a tuple of `expr`s as arguments

  ```
  Plus of expr * expr | Minus of expr * expr |
  Times of expr * expr | Div of expr * expr
  ```

- Boolean expressions: a similar idea works

  ```
  Lss of expr * expr | Gtr of expr * expr |
  Eq of expr * expr | And of expr * expr |
  Or of expr * expr | Not of expr
  ```

- Similarly for the *if-then-else*

  ```
  Cond of expr * expr * expr
  ```

- Tediously similar for *let* expressions too

  ```
  Let of string * expr * expr
  ```

An OCaml Type for Programs (Contd)

Putting the pieces together we get the following type declaration

```
type  expr =
   Id of string | Int of int | True | False
 | Plus of expr * expr | Minus of expr * expr
 | Times of expr * expr | Div of expr * expr
 | Lss of expr * expr | Gtr of expr * expr
 | Eq of expr * expr | And of expr * expr
 | Or of expr * expr | Not of expr
 | Cond of expr * expr * expr
 | Let of string * expr * expr
```

As exercises, lets represent the following as `expr` expressions

```
if (2 < 3) then true else false
if (2 < 3) then 17 * 25 else 7
let x = 5 in x + 3
```

Computing Over Program Representations

We can think of different kinds of computations over programs

These computations will typically require the consideration of cases and recursion that we know how to do already!

Lets consider one example, that of determining legality of identifier use in expressions such as

```
let x = 5 in let y = 7 in x + y
let x = 5 in x + y
```

The lab and homework consider other examples

Checking Legality of Name Usage (Example)

Here is a scheme that we can use to organize the checking

- We check legality relative to a given list of identifier names
  - at the very beginning, this list is empty
  - every time we go into the body of a *let*, we add the identifier of the *let* to the list
- The check itself has a simple structure
  - If the expression is a identifier, it is legal only if the the name appears in the list
  - Constants of any kind are always legal
  - if it is an expression other than a *let* it is legal if the subcomponents are legal
  - for a let, the first expression is checked with the old list and the body with the augmented list

This idea can be realized in a function of the following type

```
legalExpr : expr -> string list -> bool
```

The Full Definition of the Function

```
let rec legalExpr e nl =
  match e with
   | Id s -> member s nl
   | (Int _ | True | False) -> true
   | (Plus (e1,e2) | Minus (e1,e2) |
      Times (e1,e2) | Div (e1,e2) |
      Lss (e1,e2) | Gtr (e1,e2) |
      Eq (e1,e2) | And (e1,e2) | Or (e1,e2)) ->
      legalExpr e1 nl && legalExpr e2 nl
   | Not e1 -> legalExpr e1 nl
   | Cond (e1,e2,e3) ->
     legalExpr e1 nl && legalExpr e2 nl &&
          legalExpr e3 nl
   | Let (s,e1,e2) ->
     legalExpr e1 nl && legalExpr e2 (s::nl)
```

Sum and Product Types

The example we have considered shows the important of two kinds of operations on types in building representations

- Forming the *product* of a finite collection of types

  For example, given the type `expr`, we want a type corresponding all the pairs that can be formed from this set

  Such product types can be realized as a *tuple* type in OCaml

- Forming the *sum* or *disjoint union* of two types

  Again, given ways to form an `expr` type expression, we want to have a type that *combines* all these ways

  The combination must satisfy a proviso: we should be able to tell which set the item came from

  Type declarations in OCaml allow sum types to be defined

The bottomline: both operations are supported in OCaml

Record Types in OCaml

OCaml has a built-in type constructor for record types

The syntax for using this constructor

```
{ lab1 : type1; lab2 : type2; ... labn : typen; }
```

An example of its use

```
# type db_entry =
     { name : string ;
       salary : float ;
       phone : string
     };;
type db_entry = { name : string; salary : float;
                  phone : string; }
#
```

Record types are like tuple types, except that components are identified by labels and not position

Identifying Values of Record Type

The value and type constructors has very similar syntax

```
{ lab1 = exp1; lab2 = exp2; ... labn = expn; }
```

An example use

```
#  let jasrec =
     { name = "jason";
       salary = 50.0 +. 25.0;
       phone ="x6831";
     };;
val jasrec : db_entry =
    {name = "jason"; salary = 75.; phone = "x6831"}
#
```

To be type correct, a binding must be indicated for *all* the labels and the type for each must match the required one

Note also that the expressions are evaluated *eagerly*

Accessing Fields in a Record

This can be done in two ways

- Using the familiar means for projecting based on the label

```
# jasrec.name;;
- : string = "jason"
#
```

- Using pattern matching

```
# let { name = h; salary = s; } = jasrec;;
val h : string = "jason"
val s : float = 75.
#
```

The pattern does not need to mention all the labels

The Type Expressions of OCaml

We assume a vocabulary of

- sorts $\begin{cases} \text{builtins like } \texttt{int}, \texttt{bool}, \text{etc} \\ \text{user defined by type declarations like for } \texttt{color} \end{cases}$

- type constructors $\begin{cases} \text{builtins like } \star,\ \texttt{list},\ \text{records},\ \text{etc} \\ \text{user defined via type decls, e.g., } \texttt{btree} \end{cases}$

Type expressions are then generated as follows:

- any sort or type variable is a type
- $((\tau_1, \ldots, \tau_n)$ *tycon*) is a type if *tycon* is an *n*-ary type constructor and $\tau_1, \ldots, \tau_n$ are types
- $\tau_1 \rightarrow \tau_2$ is a type if τ_1 and τ_2 are types