S1.4: Inductive Values and Types

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

These slides were jointly developed by Gopalan Nadathur and Eric Van Wyk.

New types, new values

So far, we have been programming with built-in types and values in OCaml.

These include values of these types: `int`, `string`, `'a list`.

In this section we are interested in mechanism for creating new values, and also the new types that describe them.

Of specific interest will be data that has an "inductive" nature to it.

For example,

- binary trees - where nodes store data and potentially have sub-trees.
- expressions - where we represent simple arithmetic expressions as data to be manipulated by our programs

Type abbreviations

We can introduce new names for existing types in OCaml.

For example
```
type myInt = int
```

The `type` keyword indicates a type declaration

Here, `myInt` is just a new name for `int`.

This use is a bit like a `typedef` in C.

Type abbreviations

In practice, this is useful for giving new names for non-trivial existing types.

For example

```
type intpair = int * int
```

or

```
type dictionary = (int * string) list
```

No new values are created here. Just a new name for an existing type.

New values: enumerated types

We can create new types that have a finite number of values.

For example, a type for colors.

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

This new type has 3 values.

This is similar to an enumerated type in C.

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
```

Notice that the pattern matching we have on Boolean values is just a special case of this feature.

Disjoint Unions in OCaml

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

E.g., we may want one type that combines the `int` and `string` types in a way that allows us to tell which type the object comes from.

We can do that in OCaml using the following type declaration

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



```
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 _)`

Recap: Type Declarations in OCaml

We were exploring some of the possibilities for type declarations in OCaml.

- A means for introducing abbreviations
  `type intlist = int pair`
- A means for introducing new enumerated types
  `type color = Red | Blue | Green`
- A means for "combining" different types in a discriminated way (disjoint union)
  `type intorstr = Int of int | Str of string`

The last two actually introduce not just a type but also new ways to construct *values*.

In fact, constants like `Red` and `Int` are also called *value constructors*.

Such constructors take 0 or 1 argument to produce a value.

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`

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

Notice that having to go through the "tag" makes sure we can never mess up the types in a disjoint union in OCaml.

Disjoint Unions in OOP

How would we realize a type like `shape` in a language like Java?

- Define an abstract class corresponding to `shape`
- Define a subclass for to each `shape` constructor

```
class Circle extends Shape {
float x; float y; float radius;
// circle specific methods here
}
```

Questions: Is the OCaml way more or less flexible? Does either have preferred features?

Type Constructors and Parameterized Types

Sometimes we want a means for constructing a "structured" type that is parameterized by the type of its components

Such a mechanism is referred to as a "type constructor"

We have already seen built-in type constructors in OCaml:

- the `list` type constructor

```
int list          type of lists of integers
bool list         type of lists of booleans
(int list) list   type of lists of lists of integers
```


another type constructor

- the "tupling type constructor `*`

```
int * int         type of pairs of integers
int * int list    type pairs of integers and integer lists
```

The second is a bit unusual both in fixity and arity, but is still a type constructor

Notice that for such type constructors to be useful, they need also to be complemented with suitable *value constructors*

Type Constructors in Some Other Languages

Here are some examples of constructed types and corresponding values:

- In C:
  - the array type constructor, `int []`, an integer array
  - the pointer type constructor, `int *`, an integer pointer
  - the structure constructor, `struct { float x; float y;}`, corresponding value
- In Java:
  - the *class* type constructor, a `Person` class and a corresponding value
  - The `ArrayList` constructor, the `ArrayList<Integer>` type/class, some `ArrayList` of integers

User Defined Type Constructors in OCaml

The really interesting aspect of OCaml is that the *user* can define *new* type constructors

The mechanism is the same one we used for type declarations, except that we can also indicate a *type* parameter.

A simple example of such a declaration:
```
type 'a pair = 'a * 'a
```

With this declaration, we get the type constructor `pair` that can be used to construct types such as `int pair`

Note that We can also have multiple type parameters
```
type ('a,'b) pair = 'a * 'b
```


User Defined Type and Value Constructors

Type declarations can also be used to introduce value constructors together with type constructors

A simple, useful example of this is a "maybe" type constructor.

For example, when you are searching a database using an index, you 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 type declaration that provides suitable type and value constructors:
```
'a maybe = Nothing | Just of 'a
```

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

Notice also the use of the parameter for the type constructor in the types of the value constructors.

Recursive Datatypes

Perhaps the most useful kinds of type is one where an object of that type is built from other objects of that 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

In OCaml, we can use the same type definition to build type and value constructors for such data.

The magic: some data constructors take the same type as arguments!

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`

Notice 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 some element type have this 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)
```

Notice again the polymorphic and recursive nature of the value constructors.

Computing Over Recursive Data

Operations over recursive structures usually break down as follows:

- you do something direct in the simple "base" cases
- you use the operation on the recursive substructures and then combine the result in some relevant way

But then we have all we need in pattern matching and recursive functions to define such operations!

For example, consider adding the numbers in a list represented using

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

It is easy to define a function `sumList` using the mechanisms mentioned above