S1.5: Inductive Values and Types

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

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

New values, new types

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

Reading in the textbook: Chapter 3.

Binary trees

Recall the trees we created and used in `whirlwind_tree.ml` and `whirlwind_parametric_tree.ml`.

We'll copy some of this over to `inductive.ml`. This file will contain most of the examples from this section.

Our first step is to review some of that material.

Declaring a new type and its values

We use the `type` construct to declare new types. The format is:

`type` *type-variables* *type-name* `=` *value-constructors*

1. the `type` keyword
2. any type variables, *e.g.*`'a`, or `('a, 'b)`
   These are given only for types that have parameters
3. the type name, this begins with a lower case letter, *e.g.* `tree`
4. the `=` sign
5. the value constructors, separated by `|`

Value constructors build new values

1. the constructor name: which begins with an upper case letter, *e.g.* `Empty` or `Fork`.
2. an optional `of` component with the type of value taken to construct the new value
   e.g. `int * tree * tree` or *e.g.* `'a * 'a tree * 'a tree`

e.g. `type 'a tree = Empty | Fork of 'a tree * 'a * 'a tree`

Drawing some pictures

- Pictures help us see the structure of the data.
- These are called data structures, after all.

Symbolic data

- Consider a binary tree with the values 2, 3, 4, 5, and 6.
- How might it be organized? Where would tree edges go?
- Draw them.

3

2 5

4 6

The realization in OCaml

```
type 'a tree = Empty
             | Fork of 'a tree * 'a * 'a tree

let t1 : int tree
  = Fork (Fork (Empty, 2, Empty),
          3,
          Fork (Fork (Empty, 4, Empty),
                5,
                Fork (Empty, 6, Empty)
            )
      )
```

How can we visualize this?


```
                        Fork

           (Fork          3,              Fork)

 (Emty,   2,   Emty),         (Fork          4,          Fork )

                        (Emty,   4,   Emty),        (Emty,   5,   Emty)
```

(`Empty` is shortened to `Emty` so that the tree fits on the slide.)

Draw in the tree edges. Then draw the OCaml commas and parenthesis.

Compare this to the OCaml definition of `t1` in `inductive.ml`

Exercise S1.5: #1: Collecting the leaves of a tree

Recall:

```
type 'a tree = Empty
             | Fork of 'a tree * 'a * 'a tree
```

Write a function named `flatten` that collects the values in a tree, returning them in a list.

It will have the following start.

```
let rec flatten (t: 'a tree) : 'a list = ...
```

Many varieties of trees

There is a plethora of different tree data structures.

We can consider trees with different numbers of child trees on different nodes.

For example:

- 2-3-4 trees
- Rose trees

See the development of these in `inductive.ml`.

Exercise S1.5: #2: Summing 2-3-4 trees

Recall the type for 2-3-4 trees:

```
type 'a tree234
  = Empty234
  | Node2 of 'a * 'a tree234 * 'a tree234
  | Node3 of 'a * 'a tree234 * 'a tree234 * 'a tree234
  | Node4 of 'a * 'a tree234 * 'a tree234 * 'a tree234 * 'a tree234
```

Write a function `sum234` of type `int tree234 -> int` that sums up all the integer values in a tree.

Exercise S1.5: #3: Summing Rose trees

Recall the type for Rose trees:

```
let rt1 = Rose (3, [Rose (1, []); Rose (2, []);
                    Rose (4, [Rose (5,[]); Rose (6,[])])
                   ] )
```

Write a function `sumRose` of type `int rosetree -> int` that sums up all the integer values in a tree.

Maps and folds over trees

We'll consider trees again, in `inductive.ml`.

We will also see higher order `map` and `fold` functions for trees.

Exercise S1.5: #4: Tree maps

These functions look very similar:

```
let rec string_tree (t: int tree) : string tree = match t with
  | Empty -> Empty
  | Fork (left, n, right) -> Fork (string_tree left,
                                   string_of_int n,
                                   string_tree right)

let rec square_tree (t: int tree) : int tree = match t with
  | Empty -> Empty
  | Fork (left, n, right) -> Fork (square_tree left,
                                   n * n,
                                   square_tree right)
```

Can you write a `map` function for trees, similar in spirit to the `map` function over lists?

It would start as follows:

```
let rec tree_map (f: 'a -> 'b) (t: 'a tree) : 'b tree = ...
```


Folding/reducing trees

How might we "fold-up" or "reduce" a tree?

For example, what might `reduce` look like if it is to be used as follows:

```
let t1 : int tree
  = Fork (Fork (Empty, 2, Empty),
          3,
          Fork (Fork (Empty, 4, Empty),
                5,
                Fork (Empty, 6, Empty)
             )
      )

let sum (t: int tree) : int = reduce ... ... t1
```

Folding up inductive types

There is a recipe for writing a `fold` function for inductive types like lists and trees.

If the inductive type has n different variants (constructors), then the `fold` function takes $n+1$ arguments:

- the tree
- a way to make values of the result type for each of the n constructors

  The type for these values follows directly from the types in the product for the variant.

See the reduce function for trees and `fold_right` for lists to see this pattern.

What is `reduce` for `tree234` or `rosetree`?

Recall lists in OCaml

3 ways of writing down the exact same list.

```
let l1 : int list = [2; 3; 4; 5]
let l2 : int list = 2 :: 3 :: 4 :: 5 :: []
let l3 : int list = 2 :: (3 :: (4 :: (5 :: [])))
```

How can we visualize this?


```
        ::

  2           ::

        3           ::

              4           ::

                    5           []
```

```
let l2 : int list = 2 :: 3 :: 4 :: 5 :: []
let l3 : int list = 2 :: (3 :: (4 :: (5 :: [])))
```

Draw in the edges showing the list structure.

Trees and lists

```
type 'a tree
  = Empty
  | Fork of 'a tree * 'a * 'a tree
```

If we remove one child we get

```
type 'a list
  = []
  | (::) of 'a * 'a list
```

Thus we see lists as trees with just one child that skew to one side.

A type for optional values

Recall the difficulty we had with the function `head` that is to return the head of a list.

```
let head (xs: 'a list) : 'a =
  match xs with
  | x::_ -> x
  | _ -> raise (Invalid_argument "head given empty list")
```

This function seems easy to misuse.

We might instead like to return a value that says

- yes, the list had a head element and here it is, or
- no, the list was empty, can't give you the head element since it doesn't exist.

We need a type for this. See `maybe` or `option` in `inductive.ml`.

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:

```
type '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.

In OCaml

- Note that OCaml provides something like this already. It is the `option` type.
- `type 'a option = None | Some of 'a`
- Can we now write a total function returning the head of a list if there is one?

Exercise S1.5: #5: Total list minimum function

Write a `minList` function that works even on empty lists and returns the miminum element of the list, if there is one using an `option` type.

Recall:

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


Trees, lists, and options

```
type 'a tree
  = Empty
  | Fork of 'a tree * 'a * 'a tree
```

If we remove one child we get

```
type 'a list
  = []
  | (::) of 'a * 'a list
```

If we remove the other child we get

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

The `option` type is not inductive. It contains no values of the same `option` type.

Disjoint Unions

Consider a message type for sending 3 different kinds of messages on a channel, each with an `int` time stamp.

```
type msg = StringMsg of string * int
         | BoolMsg of bool * int
         | FloatMsg of float * int
```

- These kinds of types are sometimes called "disjoint unions". They are not inductive. They are more like records or structures.
- The values of type `msg` are the "union" of `string * int`, `bool * int`, and `float * int`.
- But, they are kept disjoint.
- We may call these the different "variants" of the type.
- The values are "marked" and kept separate by the constructor. *e.g.* `StringMsg`, `IntMsg`, or `FloatMsg`

Disjoint Unions

Recall:

```
type msg = StringMsg of string * int
         | BoolMsg of bool * int
         | FloatMsg of float * int
```

We cannot mistake a `string` for a `int` because of the marker provided by the value constructor.

To process a message we must pattern match on it to get to the values "inside".

See this example in `inductive.ml` in the `Course-Resources/Sample-Programs` directory of the public class reop.

Sums of products

Recall:

```
type msg = StringMsg of string * int
         | BoolMsg of bool * int
         | FloatMsg of float * int
```

These kinds of types are also sometimes called "sums of products".

- Each variant often consists of a product type. *e.g.* `string * int`.
- Here "product" reminds us of the cross product of the sets (of values).
- The "disjoint union" aspect represents the sum (or union) of the different values of variants.

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 other languages.

(All examples in this section are in `inductive.ml` in the `Sample Programs` directory.)

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.

Exercise S1.5: #6: Enumerated Types

Define a type `weekday` that has as values the constants
`Mon`, . . . , `Sun`

Identify a type amongst the base types in OCaml that is actually an enumerated type like `color`.

Exercise S1.5: #7: Matching on enumerated types

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

Booleans as an enumerated type

We could define Boolean values as an enumerated type using the `type` declaration.

```
type boolean = False
             | True
```

OCaml builds in the `bool` type with values `false` and `true`.

(Note the lower case letters on the values here - user defined value constructors are capitalized.)

The point is that we can see Boolean values as a simple enumerated type.

How many values in an enumerated type?

We can think of a type as indicating a set of values.

- The `int` type indicates the set of integer values from `-4611686018427387904` to `4611686018427387903`.

  There are a lot of values in this set.

  (These are `min_int` and `max_int` in OCaml.
- The `color` type indicates a set of only 3 values: { `Red` , `Blue` , `Green` }
- Our `boolean` type indicates a set of only 2 value: { `False` , `True` }
- What about this one?
  `type what = What`
  Is there any use of a type `what` with only 1 value `What`?
  If a function returns a value of this type then we already know the value that will be returned, so why bother evaluating it?

The `unit` type

- OCaml has a type with only 1 value.
- The type is called `unit` and the value is also called "unit" but is written as `()`.
- Is it used when we do not care about the input our output values of functions / operators.
- `assert (sum [1;2;3] = 6)` has type `unit` since we do not care about the result - only the exception that it might raise.
- `print_endline` has the type `string -> unit` since this function has a side-effect of printing to the screen and there is no meaningful value to return.
- `read_line` has the type `unit -> string` since we only need to control when `read_line` is called by providing it some input, but we never have any interesting values to pass to it.
- `;` is an operator of type `unit -> 'a -> 'a`.
  It is how we might add a print expression to function.
- See some examples in `inductive.ml`

Other uses of `type`

The inductive types above show the full generality of creating new types with `type`.

But there are other ways to use it.

1. for type abbreviations / type synonyms
2. "simpler" types which don't use all the capabilities

Type abbreviations

We can introduce new names for existing types in OCaml.

For example

```
type myInt = int
type estring = char list
type length = float
```

The `type` keyword indicates a type declaration

Here, `length` is just a new name for `float`.

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

These do not introduce new values, just a type synonym.

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.

Exercise S1.5: #8: Consider the following types:

```
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`
  with expected meaning.

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
}
```

Recall: Inductive Datatypes

The most useful kinds of types are ones 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 2 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

We saw this pattern with `tree`, `rosetree`, and others.

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
- concatenating strings in a tree
- mapping a function over a tree

In all cases, the recursive structure of the function follows the inductive structure of the data.