CSci 2041
Advanced Programming Principles
L9: Higher Order Functions

Eric Van Wyk

Fall 2014

Inductive values

We've just spent some time discussing inductive data types:

- how they are specified in OCaml
- how to write functions over them
- how to prove properties of these functions

Values

So now, we've seen 3 varieties of values (and types):

1. primitive values and types: `int`, `bool`, etc
   These are simple and easy to understand.

2. inductive values and types:
   `list`, `btree`, `expr`

3. functional values and types:
   ```
   let inc x = x + 1
   inc:  int -> int
   ```

Inductive and functional values

Inductive types specify the complete set of values for that type. For example, from

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

we know what all possible values of the type `int btree` are.

But functional types, such as `int -> int`, do not specify all the possible values that may "inhabit" (or have) that type.

There are many possible values of type `int -> int`:
`inc`, `square`, `fac`, `cube`, `negate` ...

Functional values

We now turn our attention to functions.

Specifically, languages in which functions are "first class citizens." They are "just values."

They can be

- defined and associated with a name (typical `let` expressions)
- passed as input to other functions
- returned as values from other functions
- specified as literal values that are not given a name (lambda expressions)

Our big questions are:

- How can we structure computations in such languages?
- How can code be easily reused? Since code reuse is a common goal in programming.

Topics

These slides (and others on higher order functions) cover the following topics:

- passing "helper functions" as arguments
  For example, consider a `find_by` function with the type

  ```
  find_by :  ('a -> 'a ->bool) -> 'a ->
             'a list -> 'a option
  ```

  that uses a helper function the check for equality when checking if an element appears in a list.
- specifying functional value using lambda expressions and curried functions
- higher order functions embodying computational patterns, for example

  ```
  map:  ('a -> 'b) -> 'a list -> 'b list
  fold:  'a list -> 'b -> ('b -> 'a -> 'b) -> 'b
  ```

Functions needing a form of equality check

Many function over lists require a check for some form of equality.

For example

- is-element-of, lookup
- grouping, partitioning
- splitting at a certain value

Let's consider a `lookup` function from homework 4 and how we can use a more general purpose `find` function instead.

We can then use `find` in another case to implement a is-element-of function.

See examples in `find_and_lookup.ml` in the code-examples directory.

Revisiting the type of find

- Our intention, was the `find` had the type
  `('a -> 'a ->bool) -> 'a -> 'a list -> 'a option`

- What did OCaml infer as the type? It was
  `('a -> 'b -> bool) -> 'a -> 'b list -> 'b option`
  What does this mean? Why are there two type variables?

- OCaml is telling us that we can use this function in more general ways than we maybe expected.

- The elements of the list don't have to be the same type as the value we are looking for.

- We can the use `find` a bit differently. See `lookup3`.

Other examples

There are other circumstances in which we may want to specify the function used for checking for some notion of equality:

- functions to group or partition a list of values
- set functions:
    - `union`, `intersect`, `setMinus`, `nub`
- btree `insert` function

Specifying these "helper" functions

- Functions like `find` and `splitBy` need to be passed some sort of equality checking function.
- How can we specify these?
  The specification of `equals` in the `is_elem` example is somewhat cumbersome.
- We have a few options
  - `let`-declared functions
  - lambda expressions
  - converting operators into functions
  - use of curried functions, sometimes called "partial application"

Lambda Expressions

- Lambda expressions let use write function values directly.
- (Historically, these are written as $\lambda x \to x + 1$, as part of Alonzo Church's "lambda calculus" for studying theoretical ideas in computation.)
- This is similar to writing integer, string, or list values directly without the need to give them a name.
  e.g. `1`, `[3.4; 5.6; 7.8 ]`
- Lambda expressions
  `fun` *formal parameters* `->` *body*
- *e.g.*
  - `fun x y -> x = y`
  - `fun x -> x + 1`
- Let's define the equality function in `is_elem` using a lambda-expression.

Converting operators into functions

- OCaml allows many infix operators to be used as functions by wrapping them in parenthesis.
  - `(+) :  int -> int -> int`
  - `(=) :  'a -> 'a -> bool`
  - However, `::` is not treated this way. So `(::)` does not work.
- Let's define the equality function in `is_elem` using a lambda-expression.

Use of curried functions

Recall the type of `find_by`

- `('a -> 'b -> bool) -> 'a -> 'b list -> 'b option`

We could define a "default" find function as follows:

- `let find = find (=)`

What is the type of `find`?

Consider the definition of `find_with` and its application.

```
let rec find_with f l =
  match l with
  | [] -> None
  | x::xs -> if f x then Some x else find_with f xs

let equals x y =  x = y

let res_1 = find_with (equals 4) [1;3;5;4;6]

let res_2 = find_with ((=) 4) [1;3;5;4;6]
```

Note the use of curried functions in using `find_with`.

(These examples are all in `find_and_lookup.ml`.)

Comparing `find_by` and `find_with`

Which one of these should a library provide?

Which provides more opportunities for reuse?

We can create `find_by` using `find_with`:

- `find_by f v l = find_with (f v) l`

but not the other way around.

This suggests that `find_with` is more "reusable" in some sense and would be the one to include in a library if, for some reason, only one could be provided.

We can:

- `let find v l = find_with ((=) v) l`

But, what if some function `f` takes a "find" function of type

- `'a -> 'a list -> option a`

as an argument?
We can easily give it `(find_by (=))` as the argument (if we've not defined `find` as above).

But find_with would require:

- `(fun v l -> find_with ((=) v l)`

or just

- `(fun v -> find_with ((=) v)`

but we need to mention the name of the argument to search for.

This is somewhat more cumbersome than using `find_by`.

So, neither `find_by` or `find_with` is obviously better than the other.

But we do want to understand the implications of the design of each one.

“partial application”

The term “partial application is not technically correct for a language like OCaml with curried functions.

With curried functions, the function type explicitly indicates that arguments are passed in one at a time.

- `add:  int -> int -> int`

Function application only takes one operation at a time.

- `add 3 4` is the same as `(add 3) 4`.
- (The parenthesis are not required.)

But these work seamlessly together so that it may feel like we are passing in more than one argument at once even though the mechanisms implementing functions don’t work that way.

"partial application"

If C allowed partial application, then for a function like `add`

- `int add (int x, int y) { return x + y; }`

then "partial application" might look like

- `add ( 3, _ )`

and evaluate to a function that takes and integer and returns an integer.

But this isn't possible in C.

The point is that "partial application" is not needed in a language with curried functions.

Ordering functions

There are also many examples of computations that require ordering values as equal, less than, or greater than.
For example,

- `min: ('a -> 'a -> int) -> 'a list -> 'a option`
- `max: ('a -> 'a -> int) -> 'a list -> 'a option`
- sort a list given an ordering function
  `sortBy: ('a -> 'a -> int) -> 'a list -> 'a list`
- merge two sorted lists
  `mergeBy: ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list`
- split a list into three
  `splitOnCompare: ('a -> 'a -> int) -> 'a -> 'a list -> ('a list, 'a list, 'a list)`

Additional examples

Functions drop_while, drop_until:

- These have the type
  `'a list -> ('a -> bool) -> 'a list`
- They return some portion of the original list, after dropping all items that return true (or false) when provided to the function.

Function take_while, take_until are similar.

More functions over functions

We can easily write functions

- change the order of arguments in a function
- compose two functions
- "curry" or "uncurry" a function

Consider `flip`:

```
let flip f a b = f b a
```