More on proofs

Sebastiaan Joosten

The map functor

- Suppose we want to store the grades of students for some exercise in a datastructure.
- We could use pairs:
  type x500 = string
  type grade = int
  type grades = (string * int) list
- However, [("SJoosten",3),("SJoosten",5)] would be a valid value, even though my x500 occurs twice.
- Also, finding my grade can be slow.

The map functor

- In python, we could use a dictionary.
- In ocaml, we would use a map:
  - a map is a structure to store a partial function (with a finite domain of definition)
  - that is: we could store ‘grade : x500 -> grade option’ as a map if it is None everywhere except for a finite number of x500 inputs.
  - this could be of type “(x500, grade) map”
  - however, that’s not how map was implemented

The map functor

- To store the map, a sorted (somewhat balanced) binary tree is used.
- As a consequence, the implementation uses a ‘less than’ on the keys of our type (in this case, our keys are x500 values).
- Here’s the signature for that:

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

- Fortunately, x500 already comes with a ‘compare’ (recall that x500 = “String”): String.compare

```
utop # String.compare;;
- : string -> string -> int = <fun>
```

Here's how we make a map:

```
utop # module Grading = Map.Make(String);;
module Grading :
  sig
    type key = string
    type 'a t = 'a Map.Make(String).t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val mem : key -> 'a t -> bool
    val add : key -> 'a -> 'a t -> 'a t
(abbreviated)
    val to_seq : 'a t -> (key * 'a) Seq.t
    val to_seq_from : key -> 'a t -> (key * 'a) Seq.t
    val add_seq : (key * 'a) Seq.t -> 'a t -> 'a t
    val of_seq : (key * 'a) Seq.t -> 'a t
  end
```

Using the map

- let myMap = Grading.(add "VanWyk" 9 (add "Joosten" 3 (add "Moen" 10 empty))))
- ugh.. let's use pipelining:
- let myMap = empty
    |> add "Moen" 9
    |> add "Joosten" 3
    |> add "VanWyk" 10
- let printPair nm grade
   = print_endline nm ^ ": " ^ string_of_int grade
- Grading.mapi printPair myMap;;
  Joosten: 3
  Moen: 10
  VanWyk: 9

Some observations

- I never said that I'm storing 'int'
- The elements in the map are printed in order of the key!
- The order in which I want to sort a datatype varies:
  - for x500, all strings are all 8 characters, allowing for a slightly faster comparison than regular string comparison
  - depending on what you're trying to find out about your data, you might use a different order
- You can select which 'compare' function to use, hence you can determine the order that 'map' uses

Let’s play around with different orders!

- First, what is the ‘int’ in String.compare?

```
utop # String.compare "Joosten" "VanWyk";;
- : int = -1
utop # String.compare "VanWyk" "Joosten";;
- : int = 1
utop # String.compare "Joosten" "Joosten";;
- : int = 0
```

- … just like C and Java (sigh)

```
utop # Int.compare 5 7;;
- : int = -1
```

Can we return, say -2 and 2?

```
module MyC = struct
  type t = int
  let compare x y = x - y
end;;
```

- module MyCMap = Map.Make(MyC)
- let mm = empty |> add 3 3 |> add 5 5 |> add 1 1
- MyCMap.map print_int mm;;
- 135
- … hmm, seems okay (note: it’s not quite okay)

How about rock-paper-scissors?

- type rps = Rock | Paper | Scissors

```
let compare x y = match (x,y) with
 | (Rock,Paper) | (Paper,Scissors) | (Scissors,Rock) -> 1
 | (Paper,Rock) | (Scissors,Paper) | (Rock,Scissors)-> -1
 | _ -> 0
module Rps = Map.Make(struct
  type t = rps
  let compare = compare
 end)
let printrps x = match x with
  | Rock -> print_endline “Rock"
  | Paper -> print_endline “Paper"
  | Scissors -> print_endline "Scissors"
```

- What will be the problem?

Our module imposes conditions!

- Recall when we said the signature was:

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

- We should have said:

```
module type OrderedType = sig
  type t
  (** [compare x y >= -1], [compare x y <= 1], [compare a a = 0].
     [compare x y = -(compare y x)], and compare must be transitive:
     if [compare a b = 1] and [compare b c = 1] then [compare a c = 1] **)
  val compare : t -> t -> int
end
```

Another proof

- Preconditions like these pop up all over.
- Next week you'll see an example that reaches back to this signature:

```
(** A module to implement a product **)
module type Product = sig
  type t
  (** Must satisfy [x * one = x] for all x : t **)
  val one : t
  (** Must satisfy [x * (y * z) = (x * y) * z]
      for all x, y, z : t **)
  val ( * ) : t -> t -> t
end
```

- It will be called "plus" and "zero" and there's one extra property

Another proof

- This week we'll use this signature instead:

```
(** A module to implement a product **)
module type Product = sig
  type t
 (** Must satisfy [x * (y * z) = (x * y) * z]
      and [x * y = y * x] **)
  val ( * ) : t -> t -> t
end
```

Proving two list-product implementations

```
module Listprod (P : Product) = struct
  let rec prod_acc acc lst = match lst with
    | [] -> acc
    | h::tl -> prod_acc P.(acc * h) tl
  let rec prod acc lst = match lst with
    | [] -> acc
    | h::tl -> P.(h * prod acc tl)
end
```

- Using it over built-in integers:

```
module LP = Listprod(struct type t = int let ( * ) = ( * ) end)
```

- Testing:

```
utop # LP.prod 2 [3;4;5];;
- : int = 120
utop # LP.prod_acc 2 [3;4;5];;
- : int = 120
```

A note on the textbook

- … the textbook proves that these functions are the same for a fold_left (prod_acc) and a fold_right (prod).
- I’m not touching higher order functions in proofs, but it helps if you see that these are the same.
- We prove:
  prod_acc a lst = prod a lst
  by induction on lst
- Two cases:
  …

A note on the textbook

- … the textbook proves that these functions are the same for a fold_left (prod_acc) and a fold_right (prod).
- I’m not touching higher order functions in proofs, but it helps if you see that these are the same.
- We prove:
  prod_acc a lst = prod a lst
  by induction on lst
- Two cases:
  case 1: lst = []
  case 2: lst = (h::tl), IH: …

A note on the textbook

- … the textbook proves that these functions are the same for a fold_left (prod_acc) and a fold_right (prod).
- I'm not touching higher order functions in proofs, but it helps if you see that these are the same.
- We prove:
  prod_acc a lst = prod a lst
  by induction on lst
- Two cases:
  case 1: lst = []
  case 2: lst = (h::tl), IH: prod_acc a tl = prod a tl
  (here tl is a constant)

Proof by induction, case lst = []

- prod_acc a lst
  = {case}
  prod_acc a []
  = {definition}
  match [] with | [] -> a | h::tl -> prod_acc P.(a * h) tl
  = {match}
  a
  = {match}
  match [] with | [] -> a | h::tl -> P.(h * prod a tl)
  = {definition}
  prod a []
  = {case}
  prod a lst

Proof by induction, case lst = h::tl IH: prod_acc a tl = prod a tl

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc P.(a * h) tl
  = {match}
  …
  = {match}
  match (h::tl) with | [] -> a | h::tl -> P.(h * prod a tl)
  = {definition}
  prod a (h::tl)
  = {case}
  prod a lst

Proof by induction, case lst = h::tl
IH: prod_acc a tl = prod a tl

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc P.(a * h) tl
  = {match}
  prod_acc P.(a * h) tl
  …
  P.(h * prod a tl)
  = {match}
  match (h::tl) with | [] -> a | h::tl -> P.(h * prod a tl)
  = {definition}
  prod a (h::tl)
  = {case}
  prod a lst

Proof by induction, case lst = h::tl
IH: prod_acc a tl = prod a tl

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc P.(a * h) tl
  = {match}
  prod_acc P.(a * h) tl

  Both sides allow us to use the IH !

  …
  P.(h * prod a tl)
  = {match}
  match (h::tl) with | [] -> a | h::tl -> P.(h * prod a tl)
  = {definition}
  prod a (h::tl)
  = {case}
  prod a lst

Proof by induction, case lst = h::tl
IH: prod_acc a tl = prod a tl

- prod_acc P.(a * h) tl
  = {IH: let a := a * h (note that 'a' is the only variable!)}
  prod P.(a * h) tl
  …
  P.(h * prod_acc a tl)
  = {IH: let a := a}
  P.(h * prod a tl)

Now we have four things that should be equal,
two of which are pairs that we can prove equal.

Time for a helper lemma

- Two ideas would help us finish the proof:
- h * prod_acc a tl = prod_acc (h * a) tl
- h * prod a tl = prod (h * a) tl

- I *always* find right-folds easier to prove with!

Time for a helper lemma:
P.(x * prod a lst = prod (x * a) lst)

- case lst = []
- x * prod a lst
  = {definition}
  x P.* match lst with | [] -> a | h::tl -> P.(h * prod a tl)
  = {case}
  x P.* match [] with | [] -> a | h::tl -> P.(h * prod a tl)
  = {match}
  P.(x * a)
  = {match}
  match [] with | [] -> P.(x * a) | h::tl -> P.(h * prod (x * a) tl)
  = {definition}
  prod P.(x * a) []
  = {case}
  prod P.(x * a) lst

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- case lst = h :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  =

  … here I need to get the h with the a,
  what is the IH anyways?

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  =

IH: x * prod a tl = prod (x * a) tl
(with tl a constant)

If I apply the IH now, I'd get the h with the a:

x * (prod (h * a) tl)

Time for a helper lemma:
x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  = {(a * (b * c) = (a * b) * c)}
  (x * h) * prod a tl

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  = {(a * (b * c) = (a * b) * c)}
  (x * h) * prod a tl
  = …

Time for a helper lemma:
x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  = {(a * (b * c) = (a * b) * c)}
  (x * h) * prod a tl
  = {(a * b) = (b * a)}
  (h * x) * prod a tl

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  = {(a * (b * c) = (a * b) * c)}
  (x * h) * prod a tl
  = {(a * b) = (b * a)}
  (h * x) * prod a tl
  = …

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  = {(a * (b * c) = (a * b) * c)}
  (x * h) * prod a tl
  = {(a * b) = (b * a)}
  (h * x) * prod a tl
  = {(a * (b * c) = (a * b) * c)}
  h * (x * prod a tl)

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- case lst = h2 :: tl
- x * prod a lst
  = {definition}
  x * match lst with | [] -> a | h::tl -> (h * prod a tl)
  = {case}
  x * match h :: tl with | [] -> a | h::tl -> (h * prod a tl)
  = {match}
  x * (h * prod a tl)
  = {(a * (b * c) = (a * b) * c)}
  (x * h) * prod a tl
  = {(a * b) = (b * a)}
  (h * x) * prod a tl
  = {(a * (b * c) = (a * b) * c)}
  h * (x * prod a tl)

Now is a good time for the IH !

Time for a helper lemma: x * prod a lst = prod (x * a) lst

- h * (x * prod a tl)
  = {IH}
  h * prod (x * a) tl
  = {match}
  match h::tl with | [] -> (x * a) | h::tl -> (h * prod (x * a) tl)
  = {definition}
  prod (x * a) (h::tl)
  = {case}
  prod (x * a) lst

Back to our proof…

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc (a * h) tl
  = {match}
  prod_acc (a * h) tl
  (this is where we got stuck)

Back to our proof…

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc (a * h) tl
  = {match}
  prod_acc (a * h) tl
  = {IH}

Back to our proof…

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc (a * h) tl
  = {match}
  prod_acc (a * h) tl
  = {IH}
  prod (a * h) tl
  = {(a * b = b * a)}
  prod (h * a) tl
  = {helper lemma}
  (h * prod a tl)

Inductive case of prod_acc a lst = prod a lst

- prod_acc a lst
  = {case}
  prod_acc a (h::tl)
  = {definition}
  match (h::tl) with | [] -> a | h::tl -> prod_acc (a * h) tl
  = {match}
  prod_acc (a * h) tl
  = {IH}
  prod (a * h) tl
  = {(a * b = b * a)}
  prod (h * a) tl
  = {helper lemma}
  (h * prod a tl)

- = {match}
  match (h::tl) with | [] -> a | h::t
  = {definition}
  prod a (h::tl)
  = {case}
  prod a lst

On this specific proof…

- This proof has many opportunities for a wrong turn:
  - choosing the wrong helper lemma
  - applying the right helper lemma too early
  - not realizing the x and h are in the wrong places
- Proving that fold_left = fold_right is tricky…
- … and can be done in multiple ways
- … but it helps to realize that that's what we're doing!
- Seeing many proofs helps us come up with proofs.

If you need proof practice

- Write a proper comparison function and prove the properties.
- Depending on the comparison function, this might be easy or hard:
  - toy: for the types unit and bool
  - easy: use built-in > and < and its properties for ‘int’
  - harder: on self-defined natural numbers, type nat = ..
  - hardest: on lists of comparable elements (using correctness of the compare of the elements in the list)
- Note: it is crucial to write ***simple*** function definitions!