Modules and modules

Sebastiaan

October 12th

UNIVERSITY OF MINNESOTA
Driven to Discover®

Outline

- Running example: exponentiation
- Generalization using functors

Multiplication on rationals

- Let’s assume this code:

```
module type Product = sig (* we’ll only use this part! *)
  type t
  val one : t
  val ( * ) : t -> t -> t
end
module Rational = struct
  type t = int * int
  let one = (1, 1)
  exception Division_by_zero
  let rec gcd a b = if b = 0 then a else
    if b > 0 then abs (gcd b (a mod b)) else - (gcd a (-b))
  let make a b = if b = 0 then raise Division_by_zero
   else let g = gcd a b in (a / g, b / g)
  let ( * ) (a, b) (c, d) = make (a * c) (b * d)
end
```

Exponentiation

- Here's how to do exponentiation using rational multiplication (naive method):

```
module RationalExponentiation = struct
  exception Negative_power
  let rec ( ** ) a b =
      if b > 0
      then Rational.( * ) a (a ** (b - 1))
      else (if b = 0 then Rational.one
            else raise Negative_power)
end
```

Exponentiation

- Here's how to do exponentiation using matrix multiplication (naive method):

```
module MatrixExponentiation = struct
  exception Negative_power
  let rec ( ** ) a b =
      if b > 0
      then Matrix.( * ) a (a ** (b - 1))
      else (if b = 0 then Matrix.one
            else raise Negative_power)
end
```

Exponentiation: the types

- Let’s look at the types in utop:
- RationalExponentiation.(**) : int * int -> int -> int * int
- (this could be ‘rational -> int -> rational’ if we bothered to use a proper type constructor)

- MatrixExponentiation.(**)
  : (int * int) array array -> int -> (int * int) array array
- (or something along those lines, depending on what a matrix is)

Generalization

- It’d be tempting to think we can write:
- MatrixExponentiation.(**)
  : ‘a -> int -> ‘a

- Indeed, if we passed a ‘one’ and ‘multiply’ function, we could write:
- MatrixExponentiation.(**)
  : (‘a -> ‘a -> ‘a) -> ‘a -> ‘a -> int -> ‘a
- … but ocaml has a nicer way!

Functor

- A functor is a module that takes a module as argument

Exponentiation, generalized

```
module type Product = sig
  type t
  val one : t
  val ( * ) : t -> t -> t
end

module Exponentiation (P : Product) = struct
  exception Negative_power
  let rec ( ** ) a b =
        if b > 0
        then P.( * ) a (a ** (b - 1))
        else (if b = 0 then P.one
              else raise Negative_power)
end
```

Exponentiation, generalized

- As with anything generalized, this doesn't actually run
- We need to create the proper instances:

```
module type Product = sig
  (…)
end

module Exponentiation (P : Product) = struct
  (…)
end

module RationalExponentiation = Exponentiation(Rational)
```

Exponentiation, generalized

```
module RationalExponentiation = Exponentiation(Rational)
```

- This gives us RationalExponentiation.(**) for exponentiation, and Rational.(*) for multiplication …

- Perhaps more conveniently:

```
module RationalWithExponentiation = struct
  include Exponentiation(Rational)
  include Rational
end
```

- This gives us a single module that has everything on rationals!

Exponentiation, improved

- Quicker way to calculate n ** 10:
- calculate n2 = n ** 2 = n * n
- calculate n4 = n ** 4 = n2 * n2
- calculate n8 = n ** 8 = n4 * n4
- calculate n ** 10 = (n ** 2) * (n ** 8) = n2 * n8
- (This uses four multiplications instead of 10.)

Exponentiation, improved

- Quicker way to calculate n ** b:
- calculate x = n ** (b/2)
- n ** b = x * x if (b/2) * (b/2) = b
  (note that / is integer division, which rounds down)
- n ** b = n * x * x otherwise

Exponentiation, improved

```
module Exponentiation (P : Product) = struct
  exception Negative_power
  let rec ( ** ) a b =
        if b > 1
        then let c = a ** (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
end
```

Testing it…

- Quickly testing it on integers:

```
module Integer = struct
  include Exponentiation(struct
      type t = int
      let one = 1
      let ( * ) = ( * )
    end)
end
```

- `Integer.(2 ** 10);;`
- `- : int = 1024`
- `(Yay!)`

Why ‘functor’?

- Here’s the type we get:

```
module Exponentiation :
  functor (P : Product) ->
    sig
      exception Negative_power
      val ( ** ) : P.t -> int -> P.t
    end
```

- It’s not quite a function, because it does not get a traditional argument and give a traditional value.
- Instead, it gets a module and yields a module!

Does this work?

- We’ve just optimized our *general* code.
- How can we be sure that this optimization is correct for all of our instances?

Does this work?

- We’ve just optimized our *general* code.
- How can we be sure that this optimization is correct for all of our instances?
  - We can test each instance (rational, matrix, …), or even better …
  - We can make a proof for each instance…
  - … but that’s still no guarantee for *future* instances
  - We can add more instances (prime fields, polynomials), but bugs arise in the instances we can’t foresee!
  - The solution is to **make assumptions explicit**!

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 0? exp1 a 0 = exp2 a 0
```

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 0? P.one = P.one  (good!)
```

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 0? P.one = P.one  (good!)
what does this give for b = 1? exp1 a 1 = exp2 a 1
```

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 0? P.one = P.one  (good!)
what does this give for b = 1? P.(a * one) = a
```

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 0? P.one = P.one  (good!)
what does this give for b = 1? P.(a * one) = a (assumption!)
```

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 3?
exp1 a 3 = …
```

Let’s find some assumptions…

```
let rec exp1 a b =
    if b > 0
    then P.( * ) a (exp1 a (b - 1))
    else (if b = 0 then P.one
          else raise Negative_power)
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 3?
exp1 a 3 = P.(a * (a * (a * one))
```

Let’s find some assumptions…

```
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 3?
exp1 a 3 = P.(a * (a * (a * one))
exp2 a 3 = …
```

Let's find some assumptions…

```
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 3?
exp1 a 3 = P.(a * (a * (a * one))
exp2 a 3 = P.(a * a * a) = …
```

Let's find some assumptions…

```
let rec exp2 a b =
        if b > 1
        then let c = exp2 a (b / 2) in
             if b mod 2 = 0 then P.(c * c)
             else P.(a * c * c)
        else (if b = 1 then a
              else (if b = 0 then P.one
                    else raise Negative_power))
what does this give for b = 3?
exp1 a 3 = P.(a * (a * (a * one))
exp2 a 3 = P.(a * a * a) = P.((a * a) * a)
```

Here are the assumptions we used:

- P.(a * one) = a
- P.((a * a) * a)
  = { from b=3 case }
  P.(a * (a * (a * one)))
  = { from previous assumption }
  P.(a * (a * a))
- … for larger numbers, there are more and more complicated assumptions, but here's one that generalizes it all:
- P.(a * (b * c)) = P.((a * b) * c)

Annotating a signature:

- Here's how we could write the product 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
```

- We could add more information, like the purpose behind needing 'one' and '*', but these assumptions are crucial for being able to optimize code that uses 'Product'.

Using it…

- Here’s optimized matrix multiplication:

```
module MatrixWithExponentiation = struct
  include Exponentiation(Matrix)
  include Matrix
end
```

Another optimization…

- Here's optimized integer multiplication:

```
module Integer = struct
  include Exponentiation(struct
      type t = int   let one = 1   let ( * ) = ( * )
    end)
end
```

- And here's how we can further improve our rational implementation:

```
module RationalWithExponentiation = struct
  include Exponentiation(Rational)
  include Rational
  let ( ** ) (x,y) n = Integer.(x ** n, y ** n)
end
```

Key take-aways

- We can take a modules as argument by writing the arguments before the = sign
- Module types (signatures) are required in the arguments.
- Document those signatures with your assumptions!
- The signature we get looks like that of a function: functor (Arg : argtype) -> sig … end
- (oh btw, the thing above is itself a signature!)

Outlook

- Proofs about code
- Proofs about recursive code
- Proofs about functors