Functional Programming in Haskell

The University of Glasgow

Get Unlimited learning

€23.99/month

TP

- Your learning
- Wishlist
- Profile
- Settings
- Account
- Sign out

Get Unlimited learning

€23.99/month

- To do
- Conversations
- Progress

View transcript

Video and captions loading.

Download video: standard or HD

Play Video

Replay

Mute

Current Time 9:11

/

Duration 9:11

Loaded: 100.00%

9:11

Stream Type LIVE

Seek to live, currently behind live LIVE

Remaining Time -0:00

1x

Playback Rate

- 2x
- 1.5x

0:05Skip to 0 minutes and 5 seconds WIM: Hello, everyone. In this optional video, I want to explain that, really, in functional languages there are only functions. What I want to do is give you a quick intuition how we can remove all the syntactic sugar from a functional language and reduce it to only lambda functions. So the full elaboration of how you can do this is explained in the article on the site. We will just cover the basic ideas. So let's start with the let construct. Suppose we have something like– so we a let block, where we have n is assigned to 10, and we define the function f of x being x plus 1. And we apply f to n.

1:02Skip to 1 minute and 2 seconds So basically the whole thing should return 11. So, we want to rewrite this so that there will only be lambda functions left. So the obvious first rewrite we can do is rewrite this function definition of the lambda. So, very simply.

1:26Skip to 1 minute and 26 seconds OK. So then the next thing we will do is, we have a let that contains two variables, and we will essentially do the equivalent of currying in the let blocks. So we will transform this let block in a nest of let blocks. So what we're doing is–

2:05Skip to 2 minutes and 5 seconds OK. So now we have a nested let. Now we have this let expression, with just a single variable here, and we can rewrite this as a lambda expression as follows. So I'll just put it on this side here.

2:32Skip to 2 minutes and 32 seconds So this whole let block here is actually entirely equivalent to a lambda function applied to the function that defines f. So we bind f to this function, and apply it to n. So all we have to do is do the same for other let. And that's quite simple. So we take the other let and transform it into a lambda expression, which says, basically, we have n applied to this construct here. And apply to 10. So if we just straighten it out a bit, we get

3:22Skip to 3 minutes and 22 seconds So we have transformed the whole nested let into a series of lambdas. This you can see, it's entirely equivalent, but it's a lot less easy to type, which is why the let syntax is available in the language of course.

3:42Skip to 3 minutes and 42 seconds So the next thing we want to do is see if we can define conditional expressions, and so on. And for that we need to define the values of true and false. You might wonder, how can we define the values of true and false without having any numbers, because remember we're going to use only functions, so even numbers are not defined yet. So it's actually quite easy. We define true and false as functions. So we say true is defined as a function of two elements, which returns the first element. And false is a function of two elements which returns the second elements. And that's all there is to it.

4:25Skip to 4 minutes and 25 seconds This is our– this is by definition, in our system, true and false. And now we can use these values of true and false, these definitions to define things like if then else constructs, or– Let's do that next. So, in if then else– so that in Haskell it looks if condition then if true else if false. So condition, if true and if false are expressions in their own right, condition evaluates to true or false. So the first thing is this if then else, we know it's an expression. And it's actually just syntactic sugar for a function, and we will call this function if then else. So we have if then else condition if true if false.

5:25Skip to 5 minutes and 25 seconds So this is our first step in removing some syntactic sugar. So now with these definitions for true and false, it turns out that the definition of this if then else is actually quite simple. It is very simply– because remember, condition evaluates to true or false. So it must return one of these two functions. These are functions of two elements, that return either the first or the second. So if it's this one, then condition will return this. If that one, condition will return that. So this way, we have defined if then else.

6:08Skip to 6 minutes and 8 seconds Neat, huh?

6:22Skip to 6 minutes and 22 seconds Finally, I want to show how to define lists. So let's say we have a list which is just 1, 2, 3. We already know from what we've seen earlier that this bracket syntax for list is actually syntactic sugar for a combination of the cons operation and the empty list. So we know that this is entirely equivalent to saying 1, 2, 3 empty list. Or written out more explicitly using the cons function.

7:06Skip to 7 minutes and 6 seconds So what we have to do now is define cons and the empty list using lambda functions and nothing else. And we will use the same trick again of using– defining a function that returns a function, because we have nothing else, remember? So we define cons as a lambda function of x and xs, which returns, in its own right, a function that operates on x and xs.

7:42Skip to 7 minutes and 42 seconds So this is– because we don't want any syntactic sugar, so normally cons is of course a function of x and y. We've rewritten it as a variable binding to a lambda expression. And this lambda expression in x and xs returns this particular function. So all we need now is the empty list. And this is again very simple. So we define the empty list as a function of anything to true. And the reason for this, why it has to be true, is because we will work on from this and define things like test for emptiness of the list, and length of the list, and so on. And with this definition the whole system is consistent.

8:32Skip to 8 minutes and 32 seconds So in the same fashion you can define tuples and then you can go on to define list operations, recursions, folds, maps. And then you can go on to define numbers simply by saying that you have a 0, of some arbitrary starting point, and then a function that always defines the next number, and so on. All that is explained in the article in more detail. But in this way, we can define our complete language in terms of nothing else than lambda functions. So really, there are only functions. OK.

In a Functional Language, There are Only Functions

Although it might seems that a language like Haskell has a lot of different objects and constructs, we can express all of them in terms of functions. Watch the video to get a little insight, then read on to find out more …

Variables and `let`

```
let
  n = 10
  f x = x+1
in
  f n

-- One variable per let =>

let
  n = 10
in
  let
    f x = x+1
  in
    f n

-- Rewrite f as lambda =>

let
  n = 10
in
  let
    f = \x -> x+1
  in
    f n

-- Rewrite inner let as lambda =>

let
  n = 10
in
  (\f -> f n) (\x -> x+1)

-- Rewrite outer let as lambda =>

( \n -> ((\f -> f n) ( \x -> x+1 )) ) 10
```

So variables and let expressions are just syntactic sugar for lambda expressions.

Tuples

```
tp = (1,"2",[3])
```

The tuple notation is syntactic sugar for a function application:

```
tp = mkTup 1 "2" [3]
```

The tuple construction function can again be defined purely using lambdas:

```
mkTup = \x y z -> \t -> t x y z
```

The same goes for the tuple accessor functions:

```
fst tp = tp (\x y z -> x)
```

```
snd tp = tp (\x y z -> y)
```

Lists

Lists can be defined in terms of the empty lists `[]` and the `cons` operation `(:)`.

```
ls = [1,2,3]
```

Rewrite using : and [] =>

```
ls = 1 : 2 : 3 : []
```

Or using cons =>

```
ls = cons 1 (cons 2 (cons 3 []))
```

Defining cons

We can define `cons` using only lambda functions as

```
cons = \x xs ->
         \c -> c x xs
```

Thus

```
ls = cons 1 (...)
   = \c -> c 1 (...)
```

We can also define `head` and `tail` using only lambdas:

```
head ls = ls (\x y -> x)
tail ls = ls (\x y -> y)
```

The empty list

We can define the empty list as follows:

```
[] = \f -> true
```

The definitions for `true` and `false` are given below under Booleans.

Then we can check if a list is empty or not:

```
isEmpty lst = lst (\x xs -> false)
```

A non-emptylist is always defined as:

```
lst = x:xs
```

which with our defintion of `(:)` is

```
lst = (\x xs -> \c -> c x xs) x xs
 = \c -> c x xs
```

Thus,

```
isEmpty lst
 = isEmpty (\c -> c x xs)
 = (\c -> c x xs) (\x xs -> false)
 = false

isEmpty []
 = isEmpty (\f -> true)
 = (\f->true) (\x xs -> false)
 = true
```

Recursion on lists

Now that we can test for the empty list we can define recursions on lists such as `foldl`, `map` etc.:

```
foldl f acc lst =
  if isEmpty lst
    then acc
    else foldl f (f (head lst) acc) (tail lst)
```

and

```
map f lst  =
  let
    map' f lst lst' = if isEmpty lst then (reverse lst') else map' f (tail lst) (head lst: lst')
  in
    map' f lst []
```

with

```
reverse lst = (foldl (\acc elt -> (elt:acc)) [] lst
```

The definitions of `foldl` and `map` use an if-then-else expression which is defined below under Conditionals.

List concatenation

```
(++) lst1 lst2 = foldl (\acc elt -> (elt:acc)) lst2 (reverse lst1)
```

The length of a list

To compute the length of a list we need integers, they are defined below.

```
length lst = foldl calc_length 0 lst
  where
    calc_length _ len = inc len
```

Conditionals

We have used conditionals in the above expressions:

```
if cond then if_true_exp else if_false_exp
```

Here `cond` is an expression returning either `true` or `false`, these are defined below.

We can write the if-then-else clause as a pure function:

```
ifthenelse cond if_true_exp if_false_exp
```

Booleans

To evaluate the condition we need to define booleans:

```
true = \x y -> x
false = \x y -> y
```

With this definition, the if-then-else becomes simply

```
ifthenelse cond if_true_exp if_false_exp = cond if_true_exp if_false_exp
```

Basic Boolean operations: `and`, `or` and `not`

Using `ifthenelse` we can define `and`, `or` and `not`:

```
and x y = ifthenelse x (ifthenelse y true) false
or x y = ifthenelse x true (ifthenelse y true false)
not x = ifthenelse x false true
```

Boolean equality: `xnor`

We note that to test equality of Booleans we can use `xnor`, and we can of course define `xor` in terms of `and`, `or` and `not`:

```
xor x y = (x or y) and (not x or not y)
xnor x y = not (xor x y)
```

Signed Integers

We define an integer as a list of booleans, in thermometer encoding, and with the following definitions:

We define usigned 0 as a 1-element list containing `false`. To get signed integers we simply define the first bit of the list as the sign bit. We define unsigned and signed versions of 0:

```
u0 = false:[]
0 = +0 = true:u0
-0 = false:u0
```

For convenience we define also:

```
isPos n = head n
isNeg n = not (head n)
isZero n = not (head (tail n))
sign n = head n
```

Integer equality

The definition of `0` makes the integer equality `(==)` easier:

```
(==) n1 n2 = let
  s1 = head n1
  s2 = head n2
  b1 = head (tail n1)
  b2 = head (tail n2)
  if (xnor s1 s2) then
    if (and (not b1) (not b2))
      then true
      else
        if (and b1 b2)
          then (==) (s1:(tail n1)) (s2:(tail n2))
          else false
    else false
```

Negation

We can also easily define negation:

```
neg n = (not (head n)):(tail n)
```

Increment and Decrement

For convenience we define also define increment and decrement operations:

```
inc n = if isPos n then true:true:(tail n) else if isZero n then 1 else false:(tail (tail n))
dec n = if isZero n then -1 else if isNeg n then false:true:(tail n) n else true:(tail (tail n))
```

Addition and Subtraction

General addition is quite easy:

```
add n1 n2 = foldl add_if_true n1 n2
  where
    add_if_true elt n1 = if elt then true:n1 else n1
```

In the same way, subtraction is also straightforward:

```
sub n1 n2 = foldl sub_if_true n1 n2
  where
    sub_if_true elt n1 = of elt then (tail n1) else n1
```

Multiplication

An easy way to define multiplication is by defining the `replicate` and `sum` operations:

```
replicate n m =
  let
    repl n m lst = if n==0 then lst else repl (dec n) m (m:lst)
  in
    repl n m []
sum lst = foldl add 0 lst
```

Then multiplication simply becomes

```
mult n m = sum (replicate n m)
```

In a similar way integer division and modulo can be defined.

Floats, Characters and Strings

We note that floats and chars use an integer representation, and strings are simply lists of chars. So we now have a language that can manipulate lists and tuples of integers, floats, chars and strings.

Share this video:

-
-
-
-

2 comments

Previous

Mark as complete

Next

Mark as complete

Comments

TP

Tomas P.

Add a comment... (plain text and markdown available)

0/1200

Post Learn more about markdown

Show:

All comments

Sort by:

Newest

Filter

- GY
  Gwee YC
  Follow
  31 OCT31 OCT
  Yes, got lost along the way, after a while. Computer science formal stuff always seems not written for humans to read or process! O Blimey!
  (edited)
  Like
  Reply
  Bookmark
  Report
- 
  Xavier Góngora
  Follow
  22 OCT22 OCT
  I found this lesson quite hard. Some references would be nice.
  Like
  Reply
  Bookmark
  Report

- Help Centre
- Child safety
- Privacy
- T&Cs

Contact FutureLearn for Support

Video