Haskell

The INFDEV@HR Team

Hogeschool Rotterdam
Rotterdam, Netherlands

Translating Lambda calculus to Haskell

Overview

- Haskell can be mapped to the lambda calculus as we did for F#
- It is slightly different than F# with respect to let-bindings
- It uses a different evaluation strategy than the straightforward beta reduction seen so far

Translating Lambda calculus to Haskell

Integers, booleans, floats, strings have the usual meaning, both in the lambda calculus, Haskell, and the languages you are used to:

```
((2 + 3) - 4)
```

```
(True && False)
```

Translating Lambda calculus to Haskell

- Conditionals behave just like in the lambda calculus
- This means that they return the evaluation of either of the two branches
- This differs from imperative languages, where we just jump into either of the two branches

```
if (True && False) then
   0
else
   1
```

Haskell

```
if (True && False) then
   0
else
   1
```

Lambda calculus

$$\text{if } (\mathrm{TRUE} \wedge \mathrm{FALSE}) \text{ then } 0 \text{ else } 1$$

$$((((\lambda p\ th\ el \rightarrow ((p\ th)\ el))\ (\mathrm{TRUE} \wedge \mathrm{FALSE}))\ 0)\ 1)$$

Translating Lambda calculus to Haskell

Functions look very similar, with \ instead of λ

```
(\x f -> (f x))
```

Translating Lambda calculus to Haskell

Functions look very similar, with \ instead of λ

```
(\x f -> (f x))
```

Just like function application

```
(((\x f -> (f x)) 3) (\x -> (3 + x)))
```

Translating Lambda calculus to Haskell

Haskell

```
(((\x f -> (f x)) 3) (\x -> (3 + x)))
```

Lambda calculus

$$(((\lambda x\ f \rightarrow (f\ x))\ 3)\ (\lambda x \rightarrow (3 + x)))$$

Translating Lambda calculus to Haskell

We can give names to functions, and code becomes much prettier as a result

```
let apply =
  \ x f -> (f x) in
((apply 3) (\x -> (3 + x)))
```

Haskell
The INFDEV@HR Team
Translating Lambda calculus to Haskell
Lazy evaluation
The "do" notation and IO monad
Conclusion

Haskell

```
let apply =
  \ x f -> (f x) in
((apply 3) (\x -> (3 + x)))
```

Lambda calculus

$$\texttt{let apply = } (\lambda \texttt{x f} \rightarrow (\texttt{f x})) \texttt{ in } ((\texttt{apply 3}) (\lambda \texttt{x} \rightarrow (\texttt{3} + \texttt{x})))$$

$$((\lambda \texttt{apply} \rightarrow ((\texttt{apply 3}) (\lambda \texttt{x} \rightarrow (\texttt{3} + \texttt{x})))) (\lambda \texttt{x f} \rightarrow (\texttt{f x})))$$

Translating Lambda calculus to Haskell

- In Haskell, unlike F#, there is no distinction between `let` and `let rec`. Everything is `let rec`.
- let-in is used to define bindings locally into a function body
- Global let bindings are simply defined by defining the function name and can be used recursively

```
fact =
  \ n ->
    if (n == 0) then
      1
    else
      ((fact (n - 1)) * n)

(fact 2)
```

Translating Lambda calculus to Haskell

Haskell

```
fact =
  \ n ->
    if (n == 0) then
      1
    else
      ((fact (n - 1)) * n)

(fact 2)
```

Lambda calculus

```
let fact = (fix (λf n→if (n = 0) then 1 else
    ((f (n − 1)) × n))) in (fact 2)
```

Translating Lambda calculus to Haskell

We can define tuples by just putting a comma between the values, with or without nesting for more than two values is done for us. Unlike F#, brackets are mandatory when defining tuples in Haskell

```
(1, True)
```

Translating Lambda calculus to Haskell

Haskell

```
(1, True)
```

Lambda calculus

```
(1, TRUE)
```

```
(((λx y→ (λf→((f x) y))) 1) TRUE)
```

Translating Lambda calculus to Haskell

- Functions such as π_1 and π_2, which both extract one item of a pair, also exist in Haskell
- They are called, respectively, `fst` and `snd`

```
(fst (1, True))
```

```
(snd (1, True))
```

Haskell

```
(fst (1, True))
```

```
(snd (1, True))
```

Lambda calculus

$(\pi_1 \ (1, \ \mathrm{TRUE}))$

$((\lambda p \rightarrow (p \ (\lambda x \ y \rightarrow x))) \ (1, \ \mathrm{TRUE}))$

$(\pi_2 \ (1, \ \mathrm{TRUE}))$

$((\lambda p \rightarrow (p \ (\lambda x \ y \rightarrow y))) \ (1, \ \mathrm{TRUE}))$

Translating Lambda calculus to Haskell

- Haskell also offers built-in discriminated unions
- Functions such as `inl` and `inr`, which both embed one item into the union, also exist in Haskell
- They are called, respectively, `Left` and `Right`

```
(Left 1)
```

```
(Right True)
```

Haskell

```
(Left 1)
```

```
(Right True)
```

Lambda calculus

```
(inl 1)
```

```
((λx→ (λf g→(f x))) 1)
```

```
(inr TRUE)
```

```
((λy→ (λf g→(g y))) TRUE)
```

Translating Lambda calculus to Haskell

We can, of course, perform matches on discriminated unions

```
case (Left 1) of
 Left x ->
   (Left x)
 Right y ->
  (Right y)
```

Translating Lambda calculus to Haskell

We can, of course, perform matches on discriminated unions

```
case (Left 1) of
 Left x ->
   (Left x)
 Right y ->
  (Right y)
```

```
let i =
  case (Left 1) of
   Left x ->
     x
   Right y ->
    0
 in
(i * 2)
```

Translating Lambda calculus to Haskell

Discriminated unions, and their corresponding matches, can be nested as deep as we need

```
case (Left (Right True)) of
 Left x ->
   case x of
   Left x ->
     (Left x)
   Right y ->
    (Right y)

 Right y ->
  (Right y)
```

- Type annotations in haskell are quite different from F#
- The declaration of the function types is separated from the body

```
fact :: Integral->Integral
fact=(\n ->
  if (n == 0) then
    1
  else
    ((fact (n - 1)) * n)
)
```

Translating Lambda calculus to Haskell

- It is possible to use pattern matching to define functions instead of using a `match` or an `if-then-else`
- It is done just repeating the function definition with the specific arguments

```
length=(\[] -> 0)
length=(\(x:xs) -> (1 + (length xs)))
```

Translating Lambda calculus to Haskell

Since the type declaration is separated from the function body, type variables for generics can be written as normal variables

Note that in Haskell the type of a list is written as [a] where a is a concrete type or a type variable

```
length :: [a]->Integral
length=(\[] -> 0)
length=(\(x:xs) -> (1 + (length xs)))
```

Lazy evaluation

Lazy evaluation

- Haskell uses a mechanism of evaluation for expressions called *lazy evaluation*
- When binding an expression to a variable the expression is not evaluated immediately
- The binding contains a “recipe” to evaluate the expression
- The evaluation is delayed until the binding is actually used in the program
- Unevaluated values are called *thunks*

Lazy evaluation

Consider the following code:

```
let (x,y) =
  ((length [1,2]), (reverse [1,2])) in
...
```

- The variables `x` and `y` initially contain thunks, until at some point in the `in` body they are used
- If the values are never used, they will never be evaluated

Lazy evaluation

Consider the following code:

```
1 let
2   z = (length [1,2],reverse [1,2])
3   (n,s) = z
4 in ...
```

- At line 1 line `z` is simply a thunk
- At line 2 the compiler must know if `z` is actually a pair, because the pattern must match the let binding
- The compiler does not need to evaluate the content of the pair
- Thus `(n,s)` becomes a pair of thunks, i.e. `z = (thunk,thunk)`

Lazy evaluation

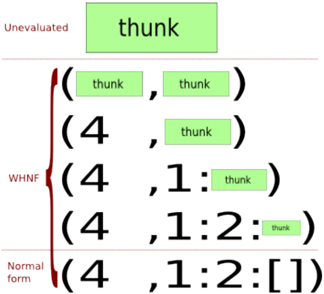
Consider the following code:

```
1 let
2   z = (length [1,2],reverse [1,2])
3   (n,s) = z
4   (1::ss) = s
5 in ...
```

- At line 4 the compiler must know if `s` is a list with the number 1 as head to match the pattern
- The compiler needs to know if `s` is a list, thus it evaluates the result of reverse as a list with a thunk as a head and another thunk as a tail, so we have thunk:thunk
- The compiler needs to know if the head of `s` matches the number 1, thus we have 1:thunk

Lazy evaluation

- The figure below shows the possible evaluation of (2, [1,2])
- WHNF = *Weak head normal form*, i.e. when the evaluation contains both values and thunks
- NF = *Normal form*, i.e. when the evaluation contains only values and no thunks



Lazy evaluation

- In the Haskell standard library we have a value called `undefined` which is used to capture errors in the program
- When the program evaluates `undefined`, execution halts and an error is returned
- Now consider the following code:

```
let
  failMiserably = \x -> undefined
  (x,y) = (4,failMiserably "Please crash")
in
  x
```

Does it crash?

Lazy evaluation

```
1 let
2   failMiserably = \x -> undefined
3   (x,y) = (4,failMiserably "Please crash")
4 in
5   x
```

The answer is no!

- At line 3 `(x,y) = (thunk,thunk)`
- At line 5 the expression only uses `x`, thus only 4 is evaluated.
- `y` is still a thunk, so the program will never know that it contains undefined
- You might have an evaluation that actually failed but you will never know because of the lazy evaluation!

The "do" notation and IO monad

The "do" notation and IO monad

- Haskell, unlike F#, is a pure funcional language
- This means we cannot make calls to imperative functions just like in F#. For example we cannot call something like `printf` because that is an imperative function
- How can we print a value to the standard output?

The "do" notation and IO monad

- In Haskell the main function is always[a] defined as a do block
- For example:

[a] For a large enough value of *always*

```
main = do
  putStr("Velociraptor\n")
  print (velociraptor 30.0 10)
```

The code allows you to print things on the shell

The "do" notation and IO monad

You are cheating!!! That is imperative code!!! So all this course is about nothing because you cannot have pure functional programming!

The "do" notation and IO monad

You are cheating!!! That is imperative code!!! So all this course is about nothing because you cannot have pure functional programming!

- No. The do notation is syntax to hide a functional structure called *Monad*
- We do not have time to explain monads in detail in this course, but they are structures that only use lambdas and a composition of lambdas to produce a result.
- It is possible to express imperative behaviours only with monads.
- If you are interested take a look at the State monad.
- In particular the IO Monad allows you to handle side effects, thus print on the shell

Conclusion

Conclusion

Closing up

- Haskell can be mapped to the lambda calculus as we did for F#: it looks mostly the same
- It does not feature non-recursive let-bindings
- It uses a lazy evaluation strategy that delays expanding values for as long as possible

This is it!

The best of luck, and thanks for the attention!