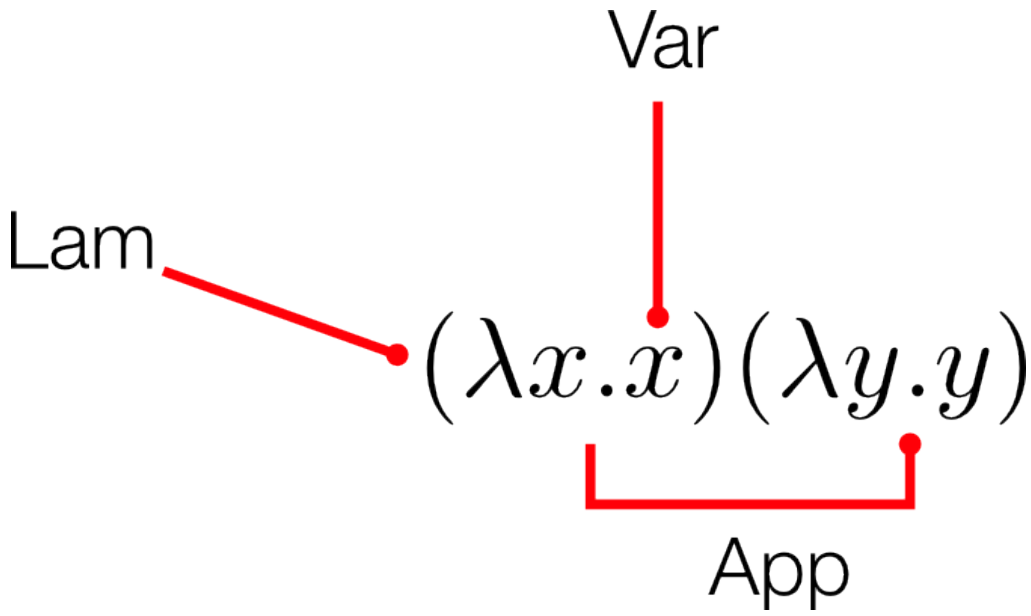
PROGRAMMING IN HASKELL

Chapter 4.2 The Lambda Calculus

Lambda Calculus

- lambda calculus is a model of computation devised in the 1930s by Alonzo Church.
- Functional programming languages all based on the lambda calculus
- Haskell is a *pure* functional language because all its features are translatable into lambda expressions
- allows higher degree of abstraction and composability

The structure of lambda terms



Alpha Equivalence

We have seen the function

$$\lambda x.x$$

The variable x here is not semantically meaningful except in its role in that single expression. Because of this, there's a form of equivalence between lambda terms called alpha equivalence. In other words,

$$\lambda x.x$$
$$\lambda apple.apple$$
$$\lambda orange.orange$$

all mean the same thing.

Beta Reduction

When we apply a function to an argument, we

- substitute the input expression for all instances of bound variables within the body of the abstraction
- eliminate the head of the abstraction (its only purpose was to bind a variable)

This process is called beta reduction.

More Beta reduction

$$\lambda x.x$$

More Beta reduction

$$\lambda x.x$$

- We apply the function above to 2

More Beta reduction

$$\lambda x.x$$

- We apply the function above to 2
- substitute 2 for each bound variable in the body of the function, and

More Beta reduction

$$\lambda x.x$$

- We apply the function above to 2
- substitute 2 for each bound variable in the body of the function, and
- eliminate the head:

More Beta reduction

$$\lambda x.x$$

- We apply the function above to 2
- substitute 2 for each bound variable in the body of the function, and
- eliminate the head:

$$\begin{array}{r} (\lambda x.x)\ 2 \\ 2 \end{array}$$

More Beta reduction

$$\lambda x.x$$

- We apply the function above to 2
- substitute 2 for each bound variable in the body of the function, and
- eliminate the head:

$$\begin{array}{r} (\lambda x.x)\ 2 \\ 2 \end{array}$$

The only bound variable is the single x, so applying this function to 2 returns 2. This function is the *identity* function.

More Beta reduction

Other Examples:

More Beta reduction

Other Examples:

$$(\lambda x.x + 1)$$

What happens if we apply this to 2? (Try this yourself)

More Beta reduction

Other Examples:

$$(\lambda x.x + 1)$$

What happens if we apply this to 2? (Try this yourself)
We can also apply our identity function to another lambda abstraction:

$$(\lambda x.x)(\lambda y.y)$$

More Beta reduction

Other Examples:

$$(\lambda x.x + 1)$$

What happens if we apply this to 2? (Try this yourself)
We can also apply our identity function to another lambda abstraction:

$$(\lambda x.x)(\lambda y.y)$$

In this case, we substitute the entire abstraction in for x. We use a new syntax here, $[x := z]$, to indicate that z will be substituted for all occurrences of x (here z is the function $(\lambda y.y)$). We reduce this application like this:

More Beta reduction

Other Examples:

$$(\lambda x.x + 1)$$

What happens if we apply this to 2? (Try this yourself)
We can also apply our identity function to another lambda abstraction:

$$(\lambda x.x)(\lambda y.y)$$

In this case, we substitute the entire abstraction in for x. We use a new syntax here, $[x := z]$, to indicate that z will be substituted for all occurrences of x (here z is the function $(\lambda y.y)$). We reduce this application like this:

$$\begin{aligned}(\lambda x.x)(\lambda y.y)\\ [x := (\lambda y.y)]\\ (\lambda y.y)\end{aligned}$$

Our final result is another identity function. There is no argument to apply it to, so we have nothing to reduce.

More Beta reduction

Once more, but this time we'll add another argument:

$$(\lambda x.x)(\lambda y.y)z$$

More Beta reduction

Once more, but this time we'll add another argument:

$$(\lambda x.x)(\lambda y.y)z$$

Applications in the lambda calculus are left associative. That is, unless specific parentheses suggest otherwise, they associate, or group, to the left. So, it can be rewritten as:

$$((\lambda x.x)(\lambda y.y))z$$

More Beta reduction

Once more, but this time we'll add another argument:

$$(\lambda x.x)(\lambda y.y)z$$

Applications in the lambda calculus are left associative. That is, unless specific parentheses suggest otherwise, they associate, or group, to the left. So, it can be rewritten as:

$$((\lambda x.x)(\lambda y.y))z$$

The β-reduction is as follows:

$$\begin{aligned} &((\lambda x.x)(\lambda y.y))z \\ &\quad [x := (\lambda y.y)] \\ &\qquad (\lambda y.y)z \\ &\qquad [y := z] \\ &\qquad\quad z \end{aligned}$$

More Beta reduction

Once more, but this time we'll add another argument:

$$(\lambda x.x)(\lambda y.y)z$$

Applications in the lambda calculus are left associative. That is, unless specific parentheses suggest otherwise, they associate, or group, to the left. So, it can be rewritten as:

$$((\lambda x.x)(\lambda y.y))z$$

The β-reduction is as follows:

$$\begin{aligned} &((\lambda x.x)(\lambda y.y))z \\ &\quad [x := (\lambda y.y)] \\ &\qquad (\lambda y.y)z \\ &\qquad\quad [y := z] \\ &\qquad\qquad z \end{aligned}$$

We can't reduce this any further as there is nothing left to apply, and we know nothing about z.

Variables

Variables can be

- bound or
- free

as the λ-calculus assumes an infinite universe of free variables. They are bound to functions in an environment, then they become bound by usage in an abstraction.

Variables

Variables can be

- bound or
- free

as the λ-calculus assumes an infinite universe of free variables. They are bound to functions in an environment, then they become bound by usage in an abstraction.
For example, in the λ-expression:

$$(\lambda x.x * y)$$

Variables

Variables can be

- bound or
- free

as the λ-calculus assumes an infinite universe of free variables. They are bound to functions in an environment, then they become bound by usage in an abstraction.
For example, in the λ-expression:

$$(\lambda x.x * y)$$

x is bound by λ over the body $x * y$, but y is a free variable.When we apply this function to an argument, nothing can be done with the y. It remains irreducible.

Variables contd.

Look at the following when we apply such a function to an argument:

Variables contd.

Look at the following when we apply such a function to an argument:

$$(\lambda x.x * y)z$$

Variables contd.

Look at the following when we apply such a function to an argument:

$$(\lambda x.x * y)z$$

We apply the lambda to the argument z.

$$\begin{aligned}(\lambda x.x * y)z \\ [x := z] \\ zy\end{aligned}$$

Variables contd.

Look at the following when we apply such a function to an argument:

$$(\lambda x.x * y)z$$

We apply the lambda to the argument z.

$$\begin{aligned}(\lambda x.x * y)z \\ [x := z] \\ zy\end{aligned}$$

The head has been applied away, and there are no more heads or bound variables. Since we know nothing about z or y, we can reduce this no further.

Multiple Arguments

Each lambda can only bind one parameter and can only accept one argument. Functions that require multiple arguments have multiple, nested heads. When you apply it once and eliminate the first (leftmost) head, the next one is applied and so on. This means that the following

$$\lambda xy.xy$$

Multiple Arguments

Each lambda can only bind one parameter and can only accept one argument. Functions that require multiple arguments have multiple, nested heads. When you apply it once and eliminate the first (leftmost) head, the next one is applied and so on. This means that the following

$$\lambda xy.xy$$

is simply syntactic sugar for

$$\lambda x(\lambda y.xy)$$

Lambda Caculus in Haskell

How do we write lambda expressions in Haskell?

Lambda Caculus in Haskell

How do we write lambda expressions in Haskell?

Named Function	Lambda Calculus (maths)	Lambda Calculus (Haskell)	Result
$f\ x = x + 1$	$(\ \lambda x.x + 1)\ 2$	$(\backslash x \rightarrow x + 1)\ 2$	3
$f\ x\ y = x * y$	$(\ \lambda x\ y.x * y)\ 2\ 3$	$(\backslash x\ y \rightarrow x * y)\ 2\ \ 3$	6
$f\ xs = `c` : xs$	$(\ \lambda xs.`c` : xs)\ \ \text{"at"}$	$(\backslash xs \rightarrow `c` : xs)\ \ \text{"at"}$	"cat"

Lambda Caculus in Haskell

How do we write lambda expressions in Haskell?

Named Function	Lambda Calculus (maths)	Lambda Calculus (Haskell)	Result
$f\ x = x + 1$	$(\ \lambda x.x + 1)\ 2$	$(\backslash x \rightarrow x + 1)\ 2$	3
$f\ x\ y = x * y$	$(\ \lambda x\ y.x * y)\ 2\ 3$	$(\backslash x\ y \rightarrow x * y)\ 2\ \ 3$	6
$f\ xs = `c` : xs$	$(\ \lambda xs.`c` : xs)\ \ \text{"at"}$	$(\backslash xs \rightarrow `c` : xs)\ \ \text{"at"}$	"cat"

Lambda functions are used extensively in Haskell, notably with Higher Order Functions.

Encoding Lambda Calculus

- Alonzo Church is probably most well known for inventing lambda calculus,

Encoding Lambda Calculus

- Alonzo Church is probably most well known for inventing lambda calculus,
- Church encodings were developed by Alonzo Church

Encoding Lambda Calculus

- Alonzo Church is probably most well known for inventing lambda calculus,
- Church encodings were developed by Alonzo Church
- Church encodings are a very interesting development arising from lambda calculus,

Encoding Lambda Calculus

- Alonzo Church is probably most well known for inventing lambda calculus,
- Church encodings were developed by Alonzo Church
- Church encodings are a very interesting development arising from lambda calculus,
- Church found out that every concept in programming languages can be represented using functions!

Encoding Lambda Calculus

- Alonzo Church is probably most well known for inventing lambda calculus,
- Church encodings were developed by Alonzo Church
- Church encodings are a very interesting development arising from lambda calculus,
- Church found out that every concept in programming languages can be represented using functions!
- Everything from boolean logic, conditional statements, numbers (natural, integer, real, complex, imaginary), and even loops (infinite loops also)!

Encoding Lambda Calculus.. contd.

- So how do we encode all these constructs using only functions?

Encoding Lambda Calculus.. contd.

- So how do we encode all these constructs using only functions?
- **Idea:** Encode the ***behavior*** of values and not their structure

Encoding Booleans in λ Calculus

- What can we do with a boolean?
- We can make a binary choice (***if*** expression)
- A boolean is a function that, given two choices, selects one of them:

$$TRUE = \lambda x.\lambda y.x$$

$$FALSE = \lambda x.\lambda y.y$$

Encoding Booleans in λ Calculus

So how do we encode **NOT**

if we think of the structure :

Encoding Booleans in λ Calculus

So how do we encode **NOT**

if we think of the structure :

if b then FALSE;
else TRUE

Encoding Booleans in λ Calculus

So how do we encode **NOT**

if we think of the structure :

$$\textit{if } b \textit{ then } \textit{FALSE};$$
$$\textit{else } \textit{TRUE}$$

$$\textit{NOT} = \lambda b.b\ \textit{FALSE}\ \textit{TRUE}$$

Encoding Booleans in λ Calculus

$$NOT = \lambda b.b\ FALSE\ TRUE$$

Let's test this on *NOT TRUE*:

$$NOT = \lambda b.b\ FALSE\ TRUE$$

$$NOT\ \ TRUE$$

$$(\lambda b.b\ FALSE\ TRUE)\ TRUE$$

$$(TRUE)\ FALSE\ TRUE$$

$$= FALSE$$

Encoding Booleans in λ Calculus

Next, let's test this on *NOT FALSE*:

$$NOT\ \ FALSE$$

$$(\lambda b.b\ FALSE\ TRUE)\ FALSE$$

$$(FALSE)\ FALSE\ TRUE$$

$$= TRUE$$

Encoding Lambda Calculus.. Exercise

Using the techniques above, encode the following using lambda calculus:

1. AND
2. OR

Encoding Lambda Calculus..video

For a very good introduction to lambda calculus and encoding boolean logic using lambdas, see link to Graham Hutton's video (Computerphile) here

(You'll need to view this pdf through Adobe Reader to use the link. Otherwise Google 'youtube Lambda Calculus Computerphile Graham Hutton')