Haskell

Lazy

Haskell, on the other hand, is lazy

Haskell
Lazy

Haskell, on the other hand, is lazy

It never evaluates anything until it is needed

Haskell

Lazy

Haskell, on the other hand, is lazy

It never evaluates anything until it is needed

In the assignment `ints = from 0` it does *not* evaluate the `from`

Haskell

Lazy

Haskell, on the other hand, is lazy

It never evaluates anything until it is needed

In the assignment `ints = from 0` it does *not* evaluate the `from`

In a sense it provides a "promise" (also called a "thunk", the same as the technique of delayed evaluation in Lisp using lambdas) that it will evaluate the elements of `ints` if you ever need them

Haskell

Lazy

Haskell, on the other hand, is lazy

It never evaluates anything until it is needed

In the assignment `ints = from 0` it does *not* evaluate the `from`

In a sense it provides a "promise" (also called a "thunk", the same as the technique of delayed evaluation in Lisp using lambdas) that it will evaluate the elements of `ints` if you ever need them

If not, it doesn't bother

If you ask `head ints` it will evaluate the `from` just enough to get you the head, namely 0

If you ask `head ints` it will evaluate the `from` just enough to get you the head, namely 0

If you ask `head (tail ints)` it will evaluate the `from` just a little bit further to get you the head of the tail, namely 1

Note that Haskell does not require `()` around the argument to a function call, but be careful as "`head tail ints`" is interpreted as
"`(head tail) ints`" and so is rejected as an error

```
> head tail ints
ERROR - Type error in application
*** Expression     : head tail ints
*** Term           : tail
*** Type           : [b] -> [b]
*** Does not match : [[Integer] -> a]
```

`tail` is of type `[b] -> [b]` (here `b` is a type variable different from the `a` below it), but Haskell was expecting something of type `[[Integer] -> a]`, a *list* of functions

Exercise. Explain why Haskell was expecting a list of functions each of which takes a list of integers and returns some object of unknown type `a`

Exercise. Explain why Haskell was expecting a list of functions each of which takes a list of integers and returns some object of unknown type `a`

So you need to have `head(tail ints)` in this case

Now try

```
sqs = map (\n -> n*n) ints
> head(tail(tail sqs))
= 4 :: Integer
```

Now try

```
sqs = map (\n -> n*n) ints
> head(tail(tail sqs))
= 4 :: Integer
```

We can manipulate this lazy structure as much as we wish: things only get evaluated if we need them

Now try

```
sqs = map (\n -> n*n) ints
> head(tail(tail sqs))
= 4 :: Integer
```

We can manipulate this lazy structure as much as we wish: things only get evaluated if we need them

One way of expressing the need for a value is to ask Haskell to print it

Infinite loops:

```
loopy n = loopy n
> :t loopy
= loopy :: a -> b
k x y = x
> :t k
= k :: a -> b -> a
```

Here `k` is a function that ignores its second argument (for the type of `k` see later)

Now

```
> k 1 (loopy 0)
= 1 :: Integer
```

Now

```
> k 1 (loopy 0)
= 1 :: Integer
```

But

```
> k (loopy 0) 1
```

goes into a busy loop. Hit ^C to interrupt

In the first `k 1 (loopy 0)` Haskell doesn't bother trying to evaluate the infinite loop as it's not needed for the answer

In the first `k 1 (loopy 0)` Haskell doesn't bother trying to evaluate the infinite loop as it's not needed for the answer

In the second `k (loopy 0) 1` you've asked to see the value of the first argument, so it has to try to evaluate the infinite loop

In the first `k 1 (loopy 0)` Haskell doesn't bother trying to evaluate the infinite loop as it's not needed for the answer

In the second `k (loopy 0) 1` you've asked to see the value of the first argument, so it has to try to evaluate the infinite loop

Exercise. What happens with `length [loopy 0,loopy 0]`?

In the first `k 1 (loopy 0)` Haskell doesn't bother trying to evaluate the infinite loop as it's not needed for the answer

In the second `k (loopy 0) 1` you've asked to see the value of the first argument, so it has to try to evaluate the infinite loop

Exercise. What happens with `length [loopy 0,loopy 0]`?

Haskell is *non-strict* in its arguments of functions

In the first `k 1 (loopy 0)` Haskell doesn't bother trying to evaluate the infinite loop as it's not needed for the answer

In the second `k (loopy 0) 1` you've asked to see the value of the first argument, so it has to try to evaluate the infinite loop

Exercise. What happens with `length [loopy 0,loopy 0]`?

Haskell is *non-strict* in its arguments of functions

Lisp, like most languages, is *strict*

A function is *strict* in an argument if it requires it to be evaluated before the function itself can be evaluated

A function is *strict* in an argument if it requires it to be evaluated before the function itself can be evaluated

If not, it is *non-strict*

A function is *strict* in an argument if it requires it to be evaluated before the function itself can be evaluated

If not, it is *non-strict*

Non-strictness is naturally associated with laziness, but they are slightly different concepts

A function is *strict* in an argument if it requires it to be evaluated before the function itself can be evaluated

If not, it is *non-strict*

Non-strictness is naturally associated with laziness, but they are slightly different concepts

Most languages are mostly strict

But most languages have significant non-strict exceptions

Haskell

Strictness

But most languages have significant non-strict exceptions

`or` and `and` are typically non-strict in most languages

But most languages have significant non-strict exceptions

`or` and `and` are typically non-strict in most languages

```
(or (= x 0.0) (= (/ 1.0 x) 2.0))
if (x == 0.0 || 1.0/x == 2.0) ...
```

Are valid Lisp and C/Java

Haskell

Strictness

But most languages have significant non-strict exceptions

`or` and `and` are typically non-strict in most languages

```
(or (= x 0.0) (= (/ 1.0 x) 2.0))
if (x == 0.0 || 1.0/x == 2.0) ...
```

Are valid Lisp and C/Java

We expect `or` only to evaluate as much as it needs to secure an answer

But most languages have significant non-strict exceptions

`or` and `and` are typically non-strict in most languages

```
(or (= x 0.0) (= (/ 1.0 x) 2.0))
if (x == 0.0 || 1.0/x == 2.0) ...
```

Are valid Lisp and C/Java

We expect `or` only to evaluate as much as it needs to secure an answer

In most languages, `or` is non-strict in its arguments, which is why it is a special form in Lisp and a syntactic form in other languages

Haskell

Lazy and Non-Strict

Haskell chooses to be lazy for a few reasons

Haskell chooses to be lazy for a few reasons

- it follows certain theoretical considerations from the Lambda Calculus (Exercise: look up *normal form* and *applicative* and *normal order* evaluation) that means Haskell can evaluate some expressions that other languages can't

Haskell chooses to be lazy for a few reasons

- it follows certain theoretical considerations from the Lambda Calculus (Exercise: look up *normal form* and *applicative* and *normal order* evaluation) that means Haskell can evaluate some expressions that other languages can't
- it is claimed to be more efficient: you don't evaluate stuff you don't need. However, in practice this is often offset by the extra mechanisms you need to support lazy evaluation: using promises rather than simple values

Haskell

Lazy and Non-Strict

Haskell chooses to be lazy for a few reasons

- it follows certain theoretical considerations from the Lambda Calculus (Exercise: look up *normal form* and *applicative* and *normal order* evaluation) that means Haskell can evaluate some expressions that other languages can't
- it is claimed to be more efficient: you don't evaluate stuff you don't need. However, in practice this is often offset by the extra mechanisms you need to support lazy evaluation: using promises rather than simple values
- it allows whackiness like infinite lists

Most languages are eager and strict

Most languages are eager and strict

- those expressions that Haskell can do but an eager language can't rarely turn up in real programs (but often turn up in papers about Haskell)

Haskell

Eager and Strict

Most languages are eager and strict

- those expressions that Haskell can do but an eager language can't rarely turn up in real programs (but often turn up in papers about Haskell)
- it is easy and efficient to implement and compile eager languages as modern hardware supports them well

Haskell

Eager and Strict

Most languages are eager and strict

- those expressions that Haskell can do but an eager language can't rarely turn up in real programs (but often turn up in papers about Haskell)
- it is easy and efficient to implement and compile eager languages as modern hardware supports them well
- most programmers don't expect their language to be that clever and they can barely cope with finite datastructures

Haskell

Eager and Strict

Most languages are eager and strict

- those expressions that Haskell can do but an eager language can't rarely turn up in real programs (but often turn up in papers about Haskell)
- it is easy and efficient to implement and compile eager languages as modern hardware supports them well
- most programmers don't expect their language to be that clever and they can barely cope with finite datastructures

But eager languages still want non-strictness when it suits them

Haskell

> *"We will encourage you to develop the three great virtues of a programmer: Laziness, Impatience, and Hubris."*

Larry Wall

Before we move on from Haskell there is one more item of note

Before we move on from Haskell there is one more item of note

All functions in Haskell take exactly one argument

Before we move on from Haskell there is one more item of note

All functions in Haskell take exactly one argument

On the face of it this seems wrong: what about +?

Before we move on from Haskell there is one more item of note

All functions in Haskell take exactly one argument

On the face of it this seems wrong: what about +?

What about the example earlier: `k x y = x`?

The type of `k` is a clue

The type of `k` is a clue

```
k :: a -> b -> a
```

The type of `k` is a clue

```
k :: a -> b -> a
```

We should read this as `a -> (b -> a)`

Slightly confusingly (at first), the association of function application `k x y` is `(k x) y` but the association of type signatures `a -> b -> a` is `a -> (b -> a)`. You can always put brackets in if you are uncertain

Slightly confusingly (at first), the association of function application `k x y` is `(k x) y` but the association of type signatures `a -> b -> a` is `a -> (b -> a)`. You can always put brackets in if you are uncertain

After a while you realise it has got to be like this as it's an artifact of the way we write functions!

Thus: `k :: a -> (b -> a)` is a function *of one argument* and it returns a function of type `b -> a`

So `k 1` is a valid thing to write and it returns a function of type `a -> Integer`

So `k 1` is a valid thing to write and it returns a function of type `a -> Integer`

Then `k 1 1.0`, which we read as `(k 1) 1.0`, applies that function to the argument 1.0

So `k 1` is a valid thing to write and it returns a function of type `a -> Integer`

Then `k 1 1.0`, which we read as `(k 1) 1.0`, applies that function to the argument 1.0

Returning 1 in this case

So `k 1` is a valid thing to write and it returns a function of type `a -> Integer`

Then `k 1 1.0`, which we read as `(k 1) 1.0`, applies that function to the argument 1.0

Returning 1 in this case

`k 1` returns a function that takes an argument, ignores it and returns 1

In Haskell, wrapping + in parentheses makes it a non-infix function

In Haskell, wrapping + in parentheses makes it a non-infix function

```
(+) ::  Num a => a -> a -> a
```

In Haskell, wrapping + in parentheses makes it a non-infix function

```
(+) ::  Num a => a -> a -> a
```

```
(+) 2 ::  Num a => a -> a
```

In Haskell, wrapping + in parentheses makes it a non-infix function

```
(+) ::  Num a => a -> a -> a

(+) 2 ::  Num a => a -> a

(2 +) ::  Num a => a -> a
```

In Haskell, wrapping + in parentheses makes it a non-infix function

```
(+) ::  Num a => a -> a -> a

(+) 2 ::  Num a => a -> a

(2 +) ::  Num a => a -> a

(+ 2) ::  Num a => a -> a
```

In Haskell, wrapping + in parentheses makes it a non-infix function

```
(+) :: Num a => a -> a -> a
```

```
(+) 2 :: Num a => a -> a
```

```
(2 +) :: Num a => a -> a
```

```
(+ 2) :: Num a => a -> a
```

Exercise. Why is `(+) 2 :: Num a => a -> a` and not `Integer -> Integer`?

In Haskell, wrapping + in parentheses makes it a non-infix function

```
(+) :: Num a => a -> a -> a
```

```
(+) 2 :: Num a => a -> a
```

```
(2 +) :: Num a => a -> a
```

```
(+ 2) :: Num a => a -> a
```

Exercise. Why is `(+) 2 :: Num a => a -> a` and not `Integer -> Integer`?

Exercise. Find out what Haskell *actually* says for the last

Confusingly, you might see Haskell code like `f(1,2)`

Confusingly, you might see Haskell code like `f(1,2)`

`f` is *not* a function of two arguments

Confusingly, you might see Haskell code like `f(1,2)`

`f` is *not* a function of two arguments

It is a function of *one* argument of type `(Integer, Integer)`

Confusingly, you might see Haskell code like `f(1,2)`

`f` is *not* a function of two arguments

It is a function of *one* argument of type `(Integer, Integer)`

Here `(Integer, Integer)` is a *product* type, often written `Integer` $\times$ `Integer`

Confusingly, you might see Haskell code like `f(1,2)`

`f` is *not* a function of two arguments

It is a function of *one* argument of type `(Integer, Integer)`

Here `(Integer, Integer)` is a *product* type, often written `Integer` $\times$ `Integer`

Much like a structure in C that contains two integers

The syntax “`(,)`” is a constructor function that makes an object of the appropriate product type from the two elements

The syntax "(,)" is a constructor function that makes an object of the appropriate product type from the two elements

```
(1,2) :: (Integer,Integer)
((1, 2.0), "hello") :: ((Integer,Double),[Char])
```

The syntax "`(,)`" is a constructor function that makes an object of the appropriate product type from the two elements

```
(1,2) :: (Integer,Integer)
((1, 2.0), "hello") :: ((Integer,Double),[Char])
```

`fst` and `snd` extract the elements

The syntax “`(,)`” is a constructor function that makes an object of the appropriate product type from the two elements

```
(1,2) ::  (Integer,Integer)
((1, 2.0), "hello") ::  ((Integer,Double),[Char])
```

`fst` and `snd` extract the elements

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

The syntax “`(,)`” is a constructor function that makes an object of the appropriate product type from the two elements

```
(1,2) ::  (Integer,Integer)
((1, 2.0), "hello") ::  ((Integer,Double),[Char])
```

`fst` and `snd` extract the elements

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

N.B. `(,)` is about constructing elements of new types and has nothing to do with arrays `[]`

Product types are extremely common in computer languages, often called structure types

```
struct pairint {
  int fst;
  int snd;
};

struct pairint foo;
foo.fst = 1;
foo.snd = 2;
```

Or classes

```
class pairint {
  int fst;
  int snd;
};

pairint foo = new pairint(1,2);
```

Or classes

```
class pairint {
  int fst;
  int snd;
};

pairint foo = new pairint(1,2);
```

Whatever the construction, they take two types and produce a new single type that is a composite of the two types: a *product type* in Haskell terms

Sometimes it helps to think of `a -> b -> a` as the type of a function that takes something of type `a`, then something of type `b` and then returns something of type `a`

Sometimes it helps to think of `a -> b -> a` as the type of a function that takes something of type `a`, then something of type `b` and then returns something of type `a`

While `(a,b) -> a` is the type of a function that takes a single object of type `(a,b)` and then returns something of type `a`

Sometimes it helps to think of `a -> b -> a` as the type of a function that takes something of type `a`, then something of type `b` and then returns something of type `a`

While `(a,b) -> a` is the type of a function that takes a single object of type `(a,b)` and then returns something of type `a`

Very different functions

Sometimes it helps to think of `a -> b -> a` as the type of a function that takes something of type `a`, then something of type `b` and then returns something of type `a`

While `(a,b) -> a` is the type of a function that takes a single object of type `(a,b)` and then returns something of type `a`

Very different functions

Correct use of parentheses is essential here

Sometimes it helps to think of `a -> b -> a` as the type of a function that takes something of type `a`, then something of type `b` and then returns something of type `a`

While `(a,b) -> a` is the type of a function that takes a single object of type `(a,b)` and then returns something of type `a`

Very different functions

Correct use of parentheses is essential here

`k2(x,y) = x` has type `(a,b) -> a`

Why one argument?

Why one argument?

- again theoretical considerations from the Lambda Calculus

Why one argument?

- again theoretical considerations from the Lambda Calculus
- it allows more factoring of code: e.g., `inc = (+) 1`

Why one argument?

- again theoretical considerations from the Lambda Calculus
- it allows more factoring of code: e.g., `inc = (+) 1`
- it makes analysis of code easier

Why one argument?

- again theoretical considerations from the Lambda Calculus
- it allows more factoring of code: e.g., `inc = (+) 1`
- it makes analysis of code easier

Most importantly, there is a process called *currying* (after its inventor, Curry) that converts functions of multiple arguments into a nest of functions of a single argument

Why one argument?

- again theoretical considerations from the Lambda Calculus
- it allows more factoring of code: e.g., `inc = (+) 1`
- it makes analysis of code easier

Most importantly, there is a process called *currying* (after its inventor, Curry) that converts functions of multiple arguments into a nest of functions of a single argument

So a function of type `(a,b) -> c` can be converted into an equivalent function of type `a -> b -> c`

And *uncurrying* for the other direction

And *uncurrying* for the other direction

A function of type `a -> b -> c` can be converted into an equivalent function of type `(a,b) -> c`

And *uncurrying* for the other direction

A function of type `a -> b -> c` can be converted into an equivalent function of type `(a,b) -> c`

There is no loss of expressiveness: everything you can do with multiple argument functions you can do with single argument functions; and vice versa

And *uncurrying* for the other direction

A function of type `a -> b -> c` can be converted into an equivalent function of type `(a,b) -> c`

There is no loss of expressiveness: everything you can do with multiple argument functions you can do with single argument functions; and vice versa

Exercise. Write functions `kurry` and `unkurry` in Haskell that do the above (`curry` and `uncurry` are actually already defined). Hint: what are the types of `kurry` and `unkurry`?

Haskell

There is a huge amount of Haskell we have omitted to describe: modules for structuring programs, *monads* (special structures that facilitate programming kinds of things that are traditionally difficult in pure functional languages, like state and I/O), abstract datatypes, object orientation and classes of types, and more

Haskell

It is claimed that some compilers for Haskell produce code that is equal in speed to that from a C program even though you have the power of functional programming. It doesn't seem to be about to replace traditional languages, though

Haskell

It is claimed that some compilers for Haskell produce code that is equal in speed to that from a C program even though you have the power of functional programming. It doesn't seem to be about to replace traditional languages, though

Another functional language with similar principles is *Erlang*, and this *is* used is real life situations

Haskell

Exercise. What is the type of `+`? And `/`?

Exercise. What is the type of `map`?

Exercise. What is the type of `s` where
`s x y z = x z(y z)`?

Exercise. Array `[]` and pairing `(,)` are essentially *type constructors* in Haskell, i.e., functions on types returning types. What is the type of `((,))`? Investigate other type constructors

Haskell

Exercise. Some languages have a *sum* type constructor as well as a product type constructor. For example, `union` in C. Investigate, and find out why they are called sums and products

Exercise. Is there anything like C's `union` types in Haskell?

Haskell

A product type $\alpha \times \beta$ is a type that contains an α *and* a β. A sum type $\alpha + \beta$ is a type that contains an α *or* a β

Haskell

A product type $\alpha \times \beta$ is a type that contains an α *and* a β. A sum type $\alpha + \beta$ is a type that contains an α *or* a β

There is a strong connection between types and logic

Haskell

Exercise. Look up *Curry-Howard Correspondence*. It explains how the types of functions are related to theorems in logic

Exercise. From this correspondence, explain while we can have a function of type $\alpha \to (\beta \to \alpha)$ there cannot exist a function of type $(\alpha \to \beta) \to \alpha$

Exercise. Learn about the functional language Erlang

Exercise. Think about how the functional style can help with parallel programming: see Google's *MapReduce*