Talk Notes: Lazy Evaluation in Haskell

Date: 24.11.2017

```c
#include <stdio.h>

int add(int x, int y) {
  return x + y;
}

int main() {
  int five = add(1 + 1, 1 + 2);
  int seven = add(1 + 2, 1 + 3);

  printf("Five: %d\n", five);
  return 0;
}
```

Our function `add` is *strict* in both of its arguments. And its result is also strict. This means that:

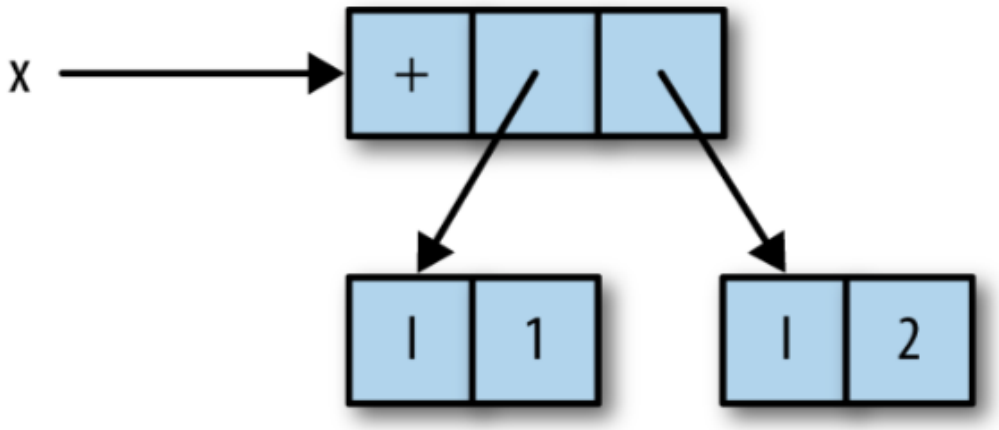
- Before `add` is called the first time, we will compute the result of both `1 + 1` and `1 + 2`.
- We will call the `add` function on `2` and `3`, get a result of `5`, and place that value in memory pointed at by the variable `five`.
- Then we'll do the same thing with `1 + 2`, `1 + 3`, and placing `7` in `seven`.
- Then we'll call `printf` with our `five` value, which is already fully computed.

```haskell
add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      seven = add (1 + 2) (1 + 3)

  putStrLn $ "Five: " ++ show five
```

- Instead of immediately computing 1 + 1 and 1 + 2, the compiler will create a *thunk* (which you can think of as a *promise*) for those computations, and pass those thunks to the add function.
- Except: we won't call the add function right away either: five will be a thunk representing the application of the add function to the thunk for 1 + 1 and 1 + 2.
- We'll end up doing the same thing with seven: it will be a thunk for applying add to two other thunks.

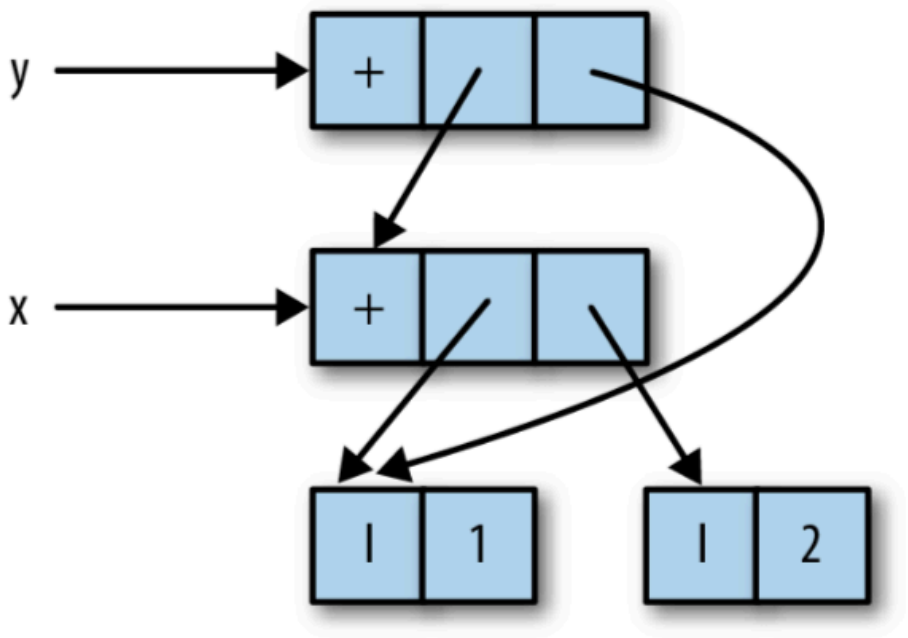
- When we finally try to print out the value five, we need to know the actual number. This is called *forcing evaluation*. We'll get into more detail on when and how this happens below, but for now, suffice it to say that when putStrLn is executed, it forces evaluation of five, which forces evaluation of 1 + 1 and 1 + 2, converting the thunks into real values (2, 3, and ultimately 5).
- Because seven is never used, it remains a thunk, and we don't spend time evaluating it.

```
λ let x = 1 + 2 :: Int
λ :sprint x
x = _
```




```
λ x
3
λ :sprint x
x = 3
```

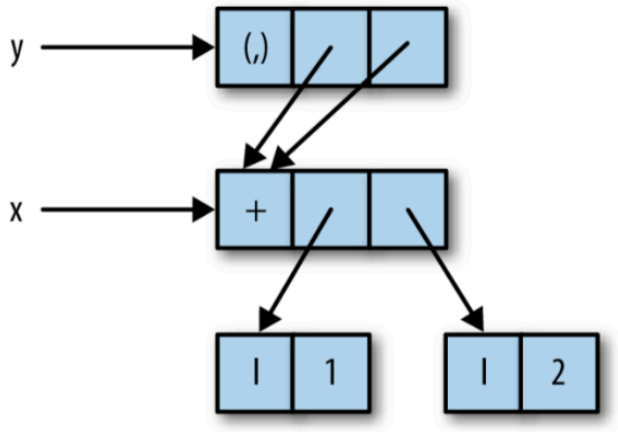
```
λ let x = 1 + 2 :: Int
λ let y = x + 1
λ :sprint x
x = _
λ :sprint y
y = _
```



`y` depends on `x`

- ```
  λ seq y ()
  ()
  λ :sprint x
  x = 3
  λ :sprint y
  y = 4
  ```
- Defining an expression causes a thunk to be built representing that expression.
- A thunk remains unevaluated until its value is required. Once evaluated, the thunk is replaced by its value.
- ```
  seq :: a -> b -> b
  ```

```
λ let x = 1 + 2 :: Int
λ let z = (x,x)
λ :sprint z
z = (_,_)
```




```
λ import Data.Tuple
λ let z = swap (x,x+1)
λ -- swap (a,b) = (b,a)
λ :sprint z
z = _
λ seq z ()
()
λ :sprint z
z = (_,_)
```

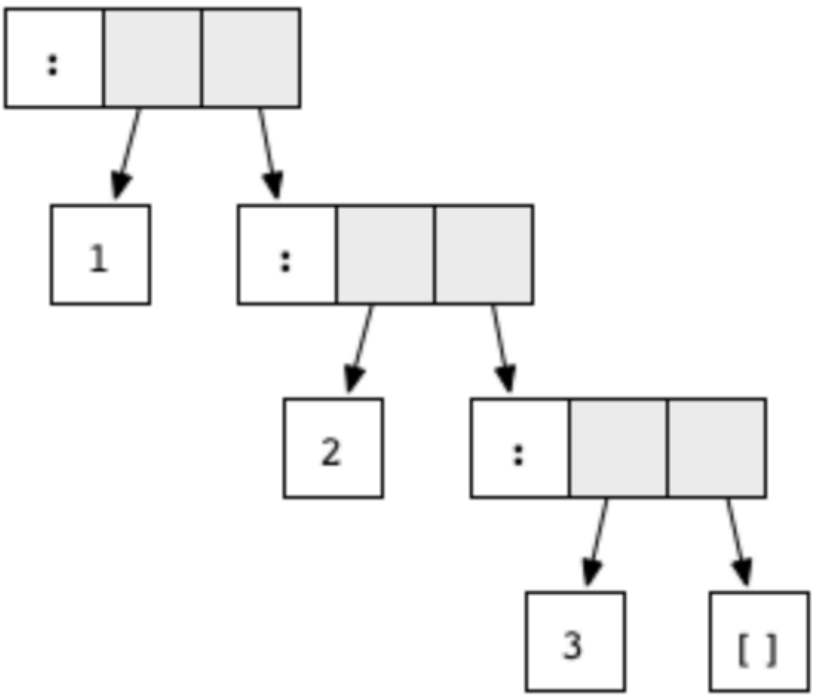
- Applying seq to z caused it to be evaluated to a pair, but the components of the pair are still unevaluated. The seq function evaluates its argument only as far as the first constructor, and doesn't evaluate any more of the structure.
- There is a technical term for this: We say that seq evaluates its first argument to weak head normal form (WHNF)
- The term normal form on its own means "fully evaluated"

```
λ seq x ()
()
λ :sprint z
z = (_,3)
```

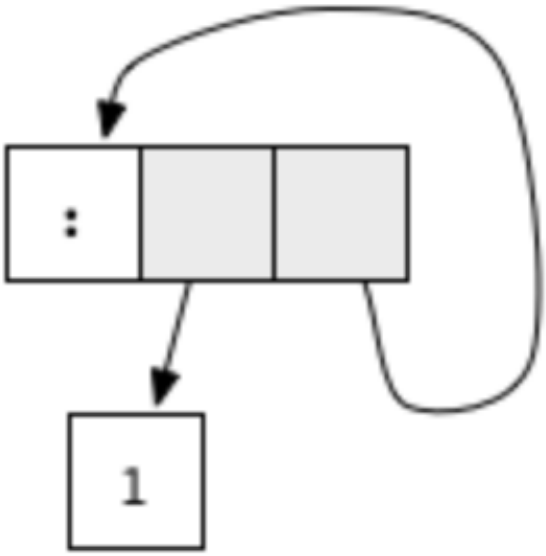
- Remember that z was defined to be `swap (x,x+1)`, which is `(x+1,x)`, and we just evaluated `x`, so the second component of `z` is now evaluated and has the value `3`.

```haskell
-- not really
data List a = Nil | Cons a (List a)

data [] a = [] | a : [a]

let xs = [1,2,3,4,5]
let ys = 1 : 2 : 3 : 4 : 5 : []
let zs = Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Nil))))
```

```haskell
let myList = [1, 2, 3]
```



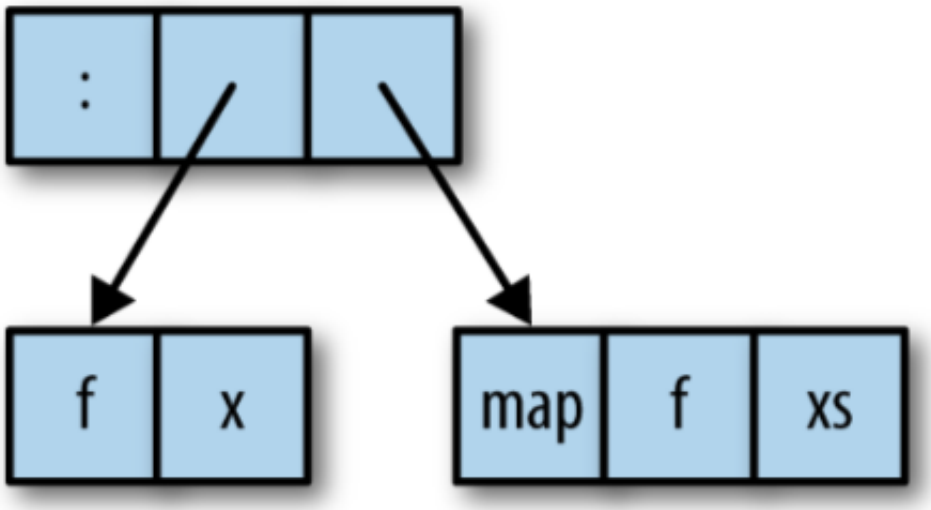

```haskell
cyclic = 1 : cyclic
```




```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

With explicit chunks:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) =
  let x' = fx
      xs' = map f xs
  in x' : xs'
```



```
λ let xs = map (+1) [1..10] :: [Int]
λ :sprint xs
xs = _
λ seq xs ()
()
λ :sprint xs
xs = _ : _
```

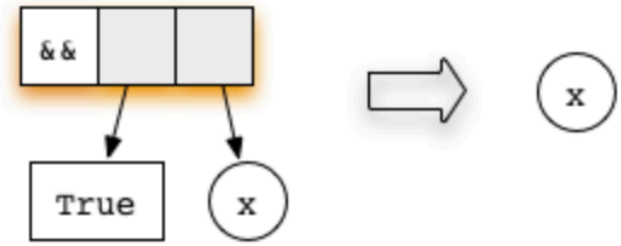
```
λ length xs
```

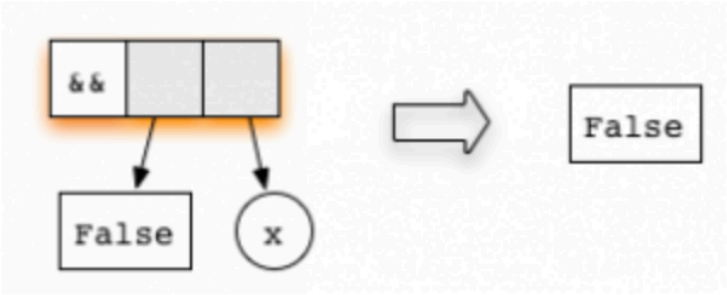
```haskell
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

- Note that length ignores the head of the list, recursing on the tail, xs.

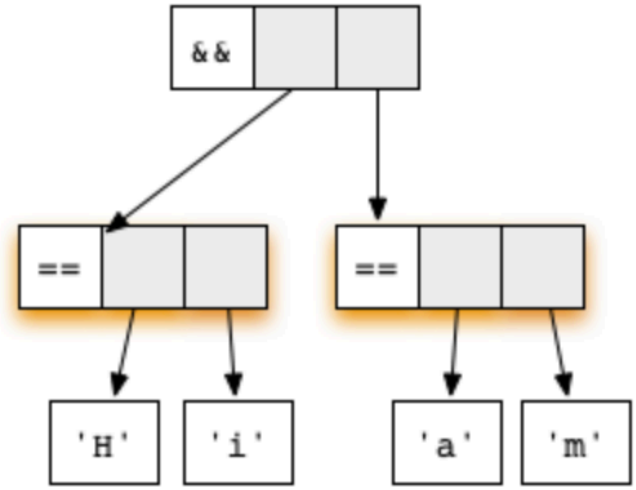
```
λ :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]
λ sum xs
65
λ :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

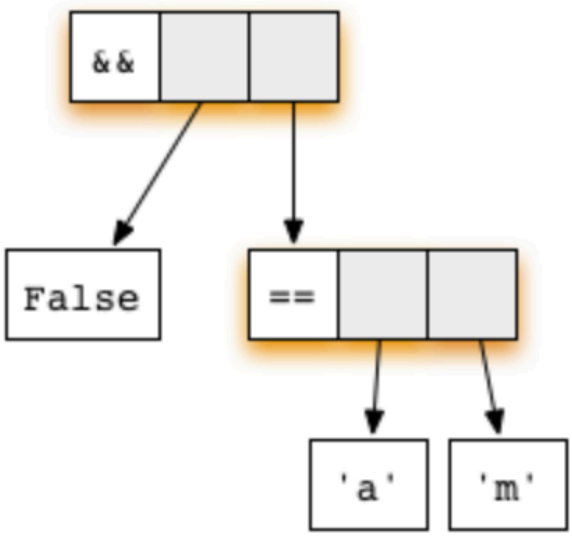
```haskell
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```





```
('H' == 'i') && ('a' == 'm')
```





- the second argument is not evaluated

```
⊥ && ⊥     = ⊥
⊥ && True  = ⊥
⊥ && False = ⊥

True && ⊥     = ⊥
True && True  = True
True && False = False

False && ⊥     = False <- this one is interesting!
False && True  = False
False && False = False
```

- Demo: fac.html, length-naive.html, length-acc.html?

Modular code

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

```
False && ((4*2 + 34) == 42)
```

- returns early

```haskell
and :: [Bool] -> Bool
and []     = True
and (b:bs) = b && and bs
```

```
and [False, True, True, True]
=>  False && and [True, True, True]
=>  False
```

```haskell
prefix :: Eq a => [a] -> [a] -> Bool
prefix xs ys = and (zipWith (==) xs ys)
```

- It returns `True` whenever one of the argument lists is a prefix of the other one

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys zipWith f _ _ = []
```

```
prefix "Haskell" "eager"
=>  and (zipWith (==) "Haskell "eager")
=>  and ('H' == 'e' : zipWith (==) "askell" "ager")
=>  and ( False     : zipWith (==) "askell" "ager")
=>  False && and (zipWith (==) "askell" "ager")
=>  False
```

- `zipWith` is completely generic
- its implementation knows nothing about the functions `and` and `(&&)`
- The three functions are independent and modular
- Yet, thanks to lazy evaluation, they work together and yield an efficient prefix test when combined

---

Inifinite lists and Corecursion

```
λ head [0..]
0
```

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```haskell
fibs = 0 : 1 : next fibs
  where
    next (a : t@(b:_)) = (a+b) : next t
```

User-defined control structures

```
myIf :: Bool -> a -> a -> a
myIf True  x y = x
myIf False x y = y

myIf (x /= 0) (100 / x) 0
```

Resolving issues

Bang!

```
{-# LANGUAGE BangPatterns #-}
add :: Int -> Int -> Int
add !x !y = x + y

main :: IO ()
main = do
  let !five = add (1 + 1) (1 + 2)
      !seven = add (1 + 2) (1 + 3)

  putStrLn $ "Five: " ++ show five
```

- This code now behaves exactly like the strict C code
- As with many things in Haskell, however, bang patterns are just syntactic sugar for something else. And in this case, that something else is the `seq` function

Tracing

```
λ import Debug.Trace
λ :t trace
trace :: String -> a -> a
λ let a = trace "a" 1
λ let b = trace "b" 2
λ a + b
b
a
3
```

Bottom

- `undefined` is special in that, when it is evaluated, it throws a runtime exception

---

```
λ map (+1) [1, 2, undefined]
[2,3,*** Exception: Prelude.undefined
```

This works:

```
#!/usr/bin/env stack
-- stack --resolver lts-9.3 script
{-# LANGUAGE BangPatterns #-}

add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      seven = add (1 + 2) undefined -- (1 + 3)

  putStrLn $ "Five: " ++ show five
```

This blows up:

```
#!/usr/bin/env stack
-- stack --resolver lts-9.3 script
{-# LANGUAGE BangPatterns #-}

add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      !seven = add (1 + 2) undefined -- (1 + 3)

  putStrLn $ "Five: " ++ show five
```

```
$ ./blowup.hs
blowup.hs: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/Err.hs:79:14 in base:GHC.Err
  undefined, called at blowup.hs:11:28 in main:Main
```

Deepseq

```
force :: NFData a => a -> a
```

```
class NFData a where
  rnf :: a -> ()
  rnf a = a `seq` ()
```

- The rnf name stands for “reduce to normal-form.” It fully evaluates its argument and then returns ()

```
instance NFData Bool

data Tree a = Empty | Branch (Tree a) a (Tree a)

instance NFData a => NFData (Tree a) where
  rnf Empty = ()
  rnf (Branch l a r) = rnf l `seq` rnf a `seq` rnf r
```

- The idea is to just recursively apply rnf to the components of the data type

---

```
-- Control.DeepSeq

deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b

force :: NFData a => a -> a
force x = x `deepseq` x
```

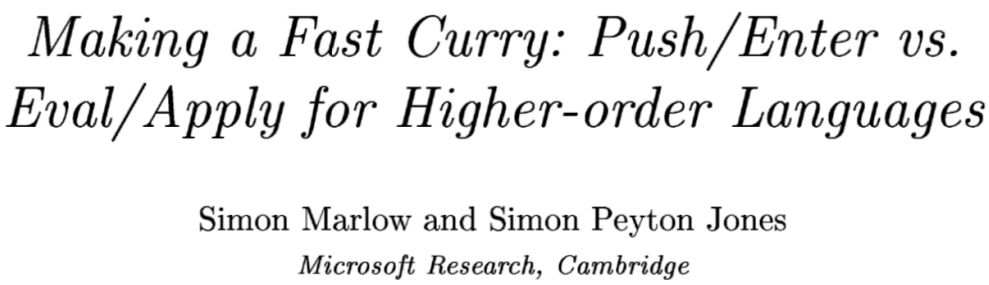
Deepseq "confidence points"

```haskell
let adIds = [...]
forM_ adIds $ \adId -> do
  r <- generateReport
  storeReport $!! r -- same as deepseq r (storeReport r)
```

Language Strict

```haskell
{-# LANGUAGE Strict #-}
{-# OPTIONS_GHC -funbox-strict-fields #-}

-- your code goes here...
```

Making a Fast Curry



Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages

Simon Marlow and Simon Peyton Jones

Microsoft Research, Cambridge

Variables	x, y, f, g			
Constructors	C			Defined in data type declarations
Literals	lit	::=	$i \mid d$	Unboxed integer or double
Atoms	a, v	::=	$lit \mid x$	Function arguments are atomic
Function arity	k	::=	$\bullet$	Unknown arity
		$\mid$	n	Known arity $n \geq 1$
Expressions	e	::=	a	Atom
		$\mid$	$f^k\ a_1 \ldots a_n$	Function call ($n \geq 1$)
		$\mid$	$\oplus\ a_1 \ldots a_n$	Saturated primitive operation ($n \geq 1$)
		$\mid$	`let` x `=` obj `in` e	
		$\mid$	`case` e `of` $\{alt_1; \ldots; alt_n\}$	($n \geq 1$)
Alternatives	alt	::=	$C\ x_1 \ldots x_n \rightarrow e$	($n \geq 0$)
		$\mid$	$x \rightarrow e$	Default alternative
Heap objects	obj	::=	$FUN(x_1 \ldots x_n \rightarrow e)$	Function (arity $= n \geq 1$)
		$\mid$	$PAP(f\ a_1 \ldots a_n)$	Partial application (f is always a FUN with $arity(f) > n \geq 1$)
		$\mid$	$CON(C\ a_1 \ldots a_n)$	Saturated constructor ($n \geq 0$)
		$\mid$	$THUNK\ e$	Thunk
		$\mid$	$BLACKHOLE$	[only during evaluation]
Programs	$prog$	::=	f_1 `=` $obj_1; \ldots; f_n$ `=` obj_n	

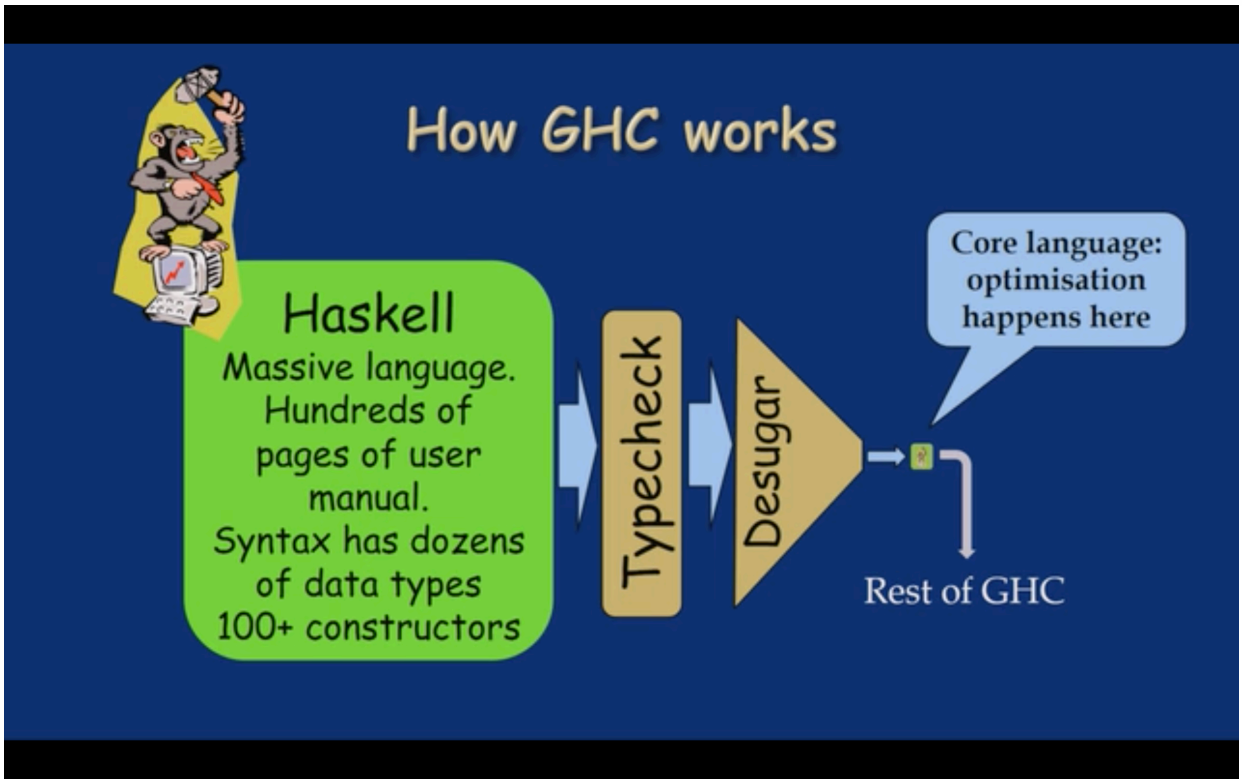
Fig. 1. Syntax

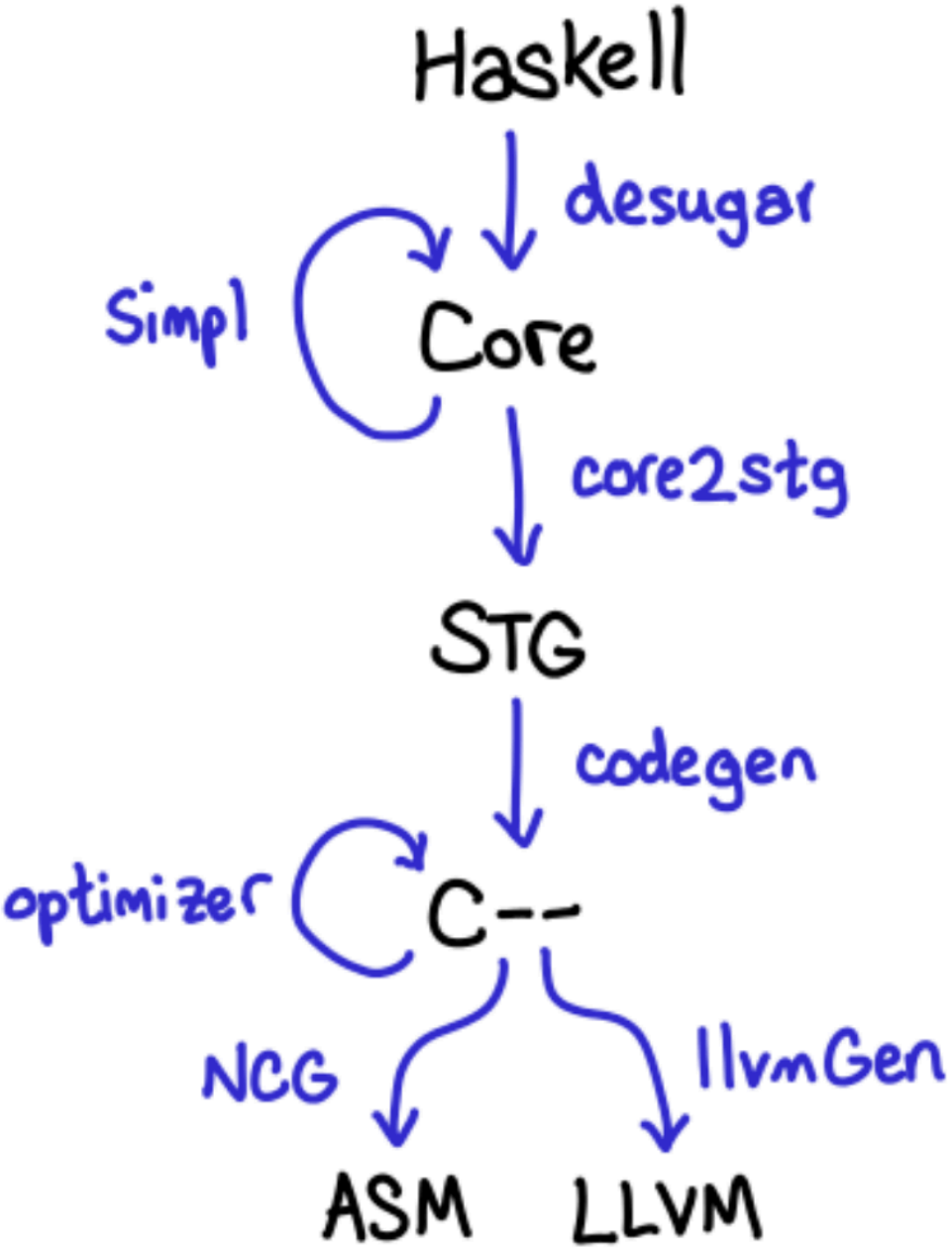
To give the idea, here is the Haskell definition of the `map` function:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

and here is its rendition into our intermediate language:

```
nil = CON Nil

map = FUN (f xs ->
        case xs of
          Nil -> nil
          Cons y ys -> let h = THUNK (f y)
                           t = THUNK (map f ys)
                           r = CON (Cons h t)
                       in r
      )
```


How GHC works
Haskell
Massive language.
Hundreds of
pages of user
manual.
Syntax has dozens
of data types
100+ constructors
Typecheck
Desugar
Core language:
optimisation
happens here
Rest of GHC



Strictness analysis

Strictness analysis - a practical approach

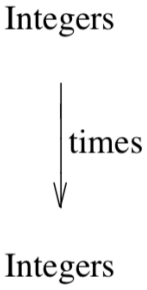
Chris Clack and Simon L Peyton Jones
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, United Kingdom

Abstract interpretation example

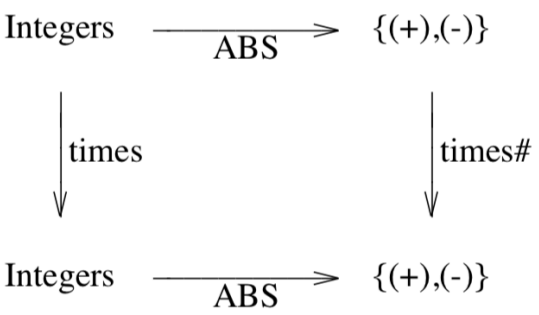
"rule of signs"

(+) times# (+)	= (+)	(-) times# (+)	= (-)
(+) times# (-)	= (-)	(-) times# (-)	= (+)

The "times" function is a function from pairs of Integers to Integers, thus:



The "times#" function models the "times" function, **but in the abstract domain,** thus:



From this diagram we can see that what we require from "times#" is that

(□) ABS (times a b) = times# (ABS a) (ABS b)

3.3 Abstracting termination information

To use this approach for strictness analysis we want to know the answer to "Does this function application terminate?". Hence a natural abstract domain is the two-point domain

$$\mathbf{T} = \{\mathbf{0}, \mathbf{1}\} \text{ ordered by } \mathbf{0} \leq \mathbf{1}$$

We intend that

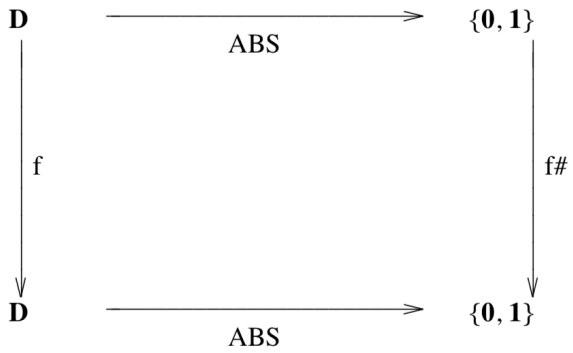
ABS x = **1** iff x MAY PERHAPS terminate

or, equivalently,

ABS x = **0** iff x DEFINITELY fails to terminate

---

and also, of course, that ABS satisfies the basic commutativity equation for any function f, depicted by the diagram:



Supposing that we could find such an ABS, and we calculate that

f# **0** = **0**

Then we are sure that

f ⊥ = ⊥

that is, we have discovered at compile time that f is strict.

Optimistic evaluation

Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs

Robert Ennals
Computer Laboratory, University of Cambridge
Robert.Ennals@cl.cam.ac.uk

Simon Peyton Jones
Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

Linear types (and their dependent lollipop)

I Got Plenty o' Nuttin'

Witheld
Witheld
witheld

Abstract

Work to date on combining linear types and dependent types has deliberately and successfully avoided doing so. Entirely fit for their own purposes, such systems wisely insist that types depend only on the replicable sublanguage, thus sidestepping the issue of counting uses of limited-use data either within types or in ways which are only really needed to shut the typechecker up. As a result, the linear implication ('lollipop') stubbornly remains a non-dependent $S \multimap T$. This paper defines and establishes the basic metatheory of a type theory supporting a 'dependent lollipop' $(x : S) \multimap T[x]$, where what the input used to be is in some way commemorated by the type of the output. Usage is tracked with resource annotations belonging to an arbitrary rig, or 'riNg without Negation'. The key insight is to use the rig's zero to mark information in contexts which is present for purposes of contemplation rather than consumption, like a meal we remember fondly but cannot eat twice. We can have

Linear dependent types are a hot topic, but for all concerned bar me, linearity stops when dependency starts. The application rules (for twain they are, and never shall they meet) from Krishnaswami and friends illustrate the puzzle.

$$\frac{\Gamma; \Delta \vdash e : A \multimap B \quad \Gamma; \Delta' \vdash e' \multimap A}{\Gamma; \Delta, \Delta' \vdash e\, e' : B}$$

$$\frac{\Gamma; \Delta \vdash e : \Pi x : X.\, A \quad \Gamma \vdash e' : X}{\Gamma; \Delta \vdash e\, e' : A[e'/x]}$$

Judgments have an intuitionistic context, Γ, shared in each premise, and a linear context Δ, carved up between the premises. Later types in Γ may depend on earlier variables, so Γ cannot be freely permuted, but in order to distribute resources in the linear application, the linear context *must* admit permutation. Accordingly, types in Δ can depend on variables from Γ, but not on linear variables. How

instead of

```
fst :: (a,b) -> a
fst (x,_) = x
```

we'll get

```
fst :: (a,b) ⊸ a
fst (x,_) = x
```

```
map :: (a ⊸ b) → [a] ⊸ [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

But please don't panic!

- Strict analysis is very smart usually

Thanks

Kostiantyn Rybnikov http://twitter.com/ko_bx

Papers:

- Peyton-Jones strictness analysis practical approach
- The Impact of Laziness on Parallelism and the Limits of Strictness Analysis (1995)
- How Much Non-strictness do Lenient Programs Require?" (Klaus E. Schauser, Seth C. Goldstein)
- Optimistic evaluation - an adaptive evaluation strategy for non-strict programs
- Cheap eagerness - speculative evaluation in a lazy functional language
- Making a fast curry
- I Got Plenty o' Nuttin' by McBride

Literature used:

- http://alpmestan.com/posts/2013-10-02-oh-my-laziness.html
- https://hackhands.com/non-strict-semantics-haskell/
- https://www.well-typed.com/blog/2017/09/visualize-cbn/
- https://www.fpcomplete.com/blog/2017/09/all-about-strictness
- http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/
- HaskellBook (http://haskellbook.com)