

Max Bolingbroke, University of Cambridge

Simon Peyton Jones, Microsoft Research Cambridge

# Types Are Calling Conventions

# Introduction

- GHC misses some fundamental optimisation opportunities!
  - We have to generate more **dynamic tests** than we would really like to when you use features like *laziness* and *higher order arguments*
- We present a new compiler intermediate language designed to recover those opportunities
- Essential observation: it is insufficient to derive a function's calling convention from its source language type in a higher-order lazy language
  - So we make sure that **our** intermediate language's types express all the calling conventions we are interested in

# The Glasgow Haskell Compiler

Haskell Program

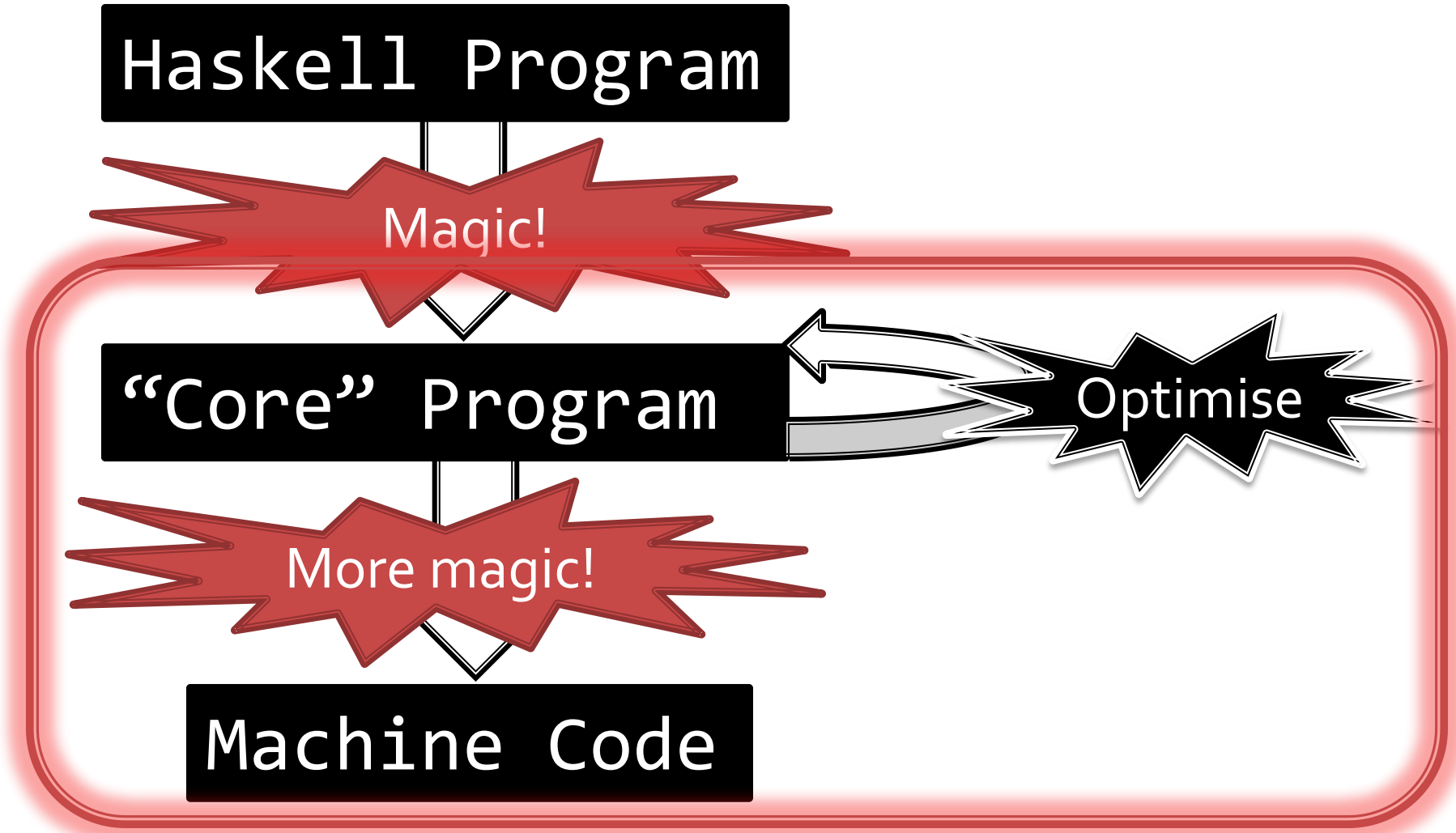
Magic!

“Core” Program

Optimise

More magic!

Machine Code



# How Many Parameters? 1

How many parameters can you supply to a function of this type?

```
f :: Int -> Int -> Int
```

As far as the user is concerned, the answer is obviously two:

```
f 1 2
```

# How Many Parameters? 2

However, the compiler actually makes a finer grained distinction:

Two: `f2 = \y -> \z -> let x = fib 10 in x + y + z`

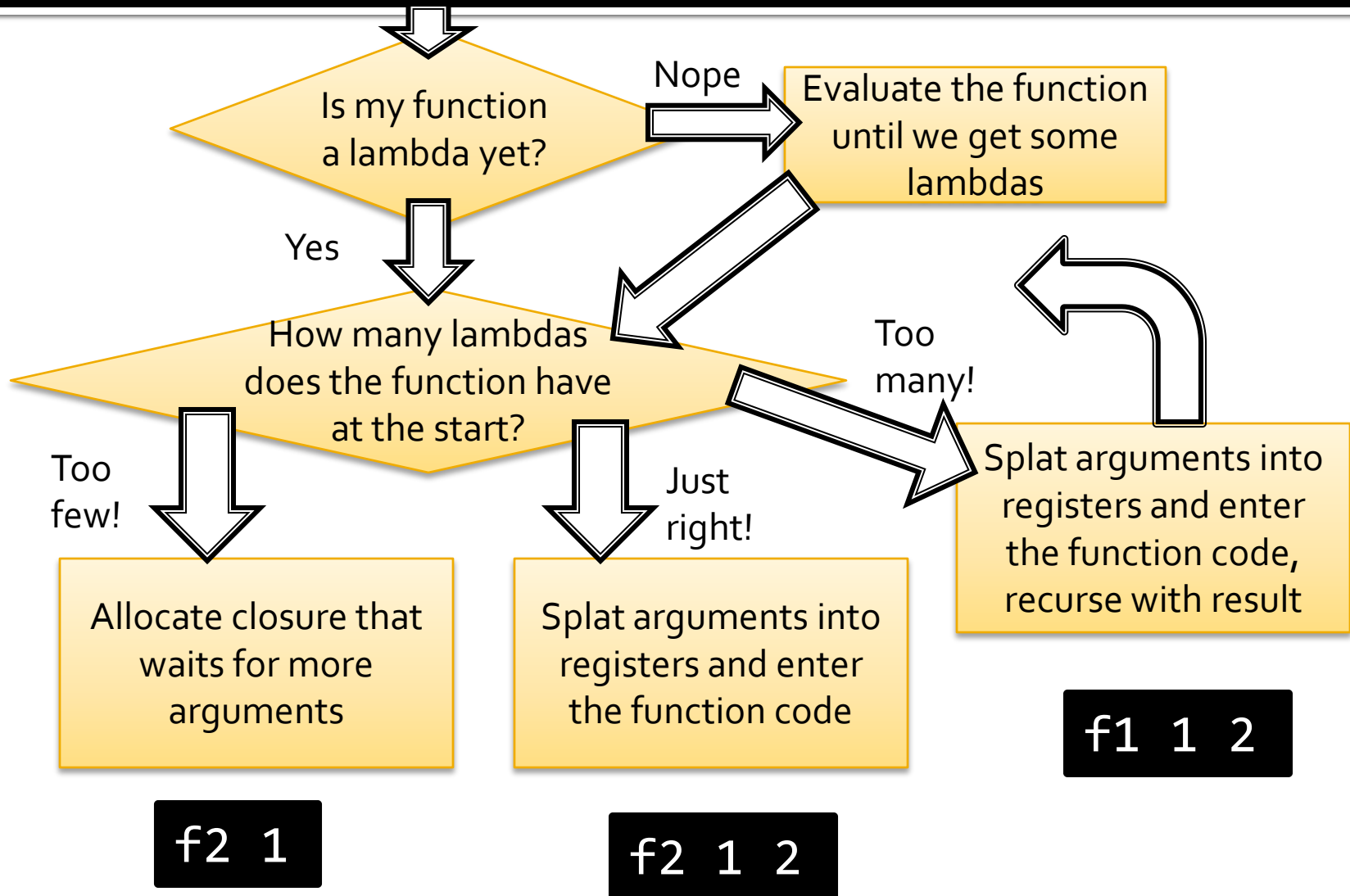
One: `f1 = \y -> let x = fib 10 in \z -> x + y + z`

Zero! `f0 = let x = fib 10 in \y -> \z -> x + y + z`

**Arity** is the number of arguments we can apply before the function does some “real work”. The arity changes how we call the function.

Arity depends on the **implementation** of the function, not its type! In stark contrast to C, where all calling convention information **must** be derivable from the type!

# Function Application Today



# What Sucks

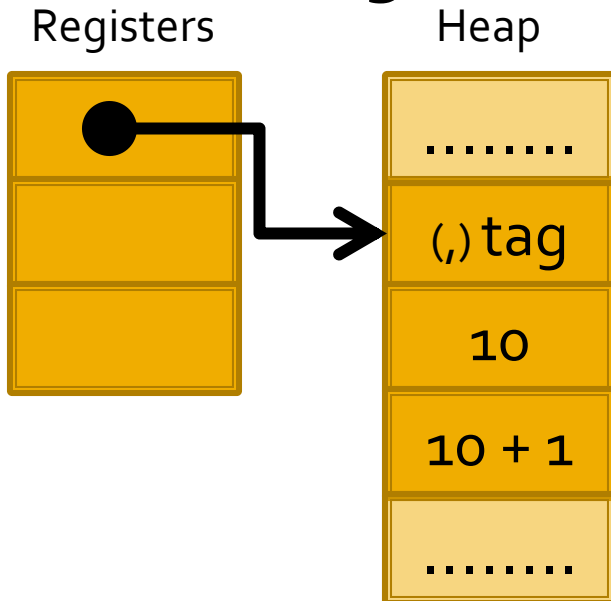
- Mandatory **dynamic** check of the function's runtime *arity* against the number of available arguments
- Mandatory **dynamic** check that the function has actually been evaluated to some lambdas
  - Things are actually worse than this: for example, there is no way to encode the fact that e.g. a `Bool` argument must have been evaluated by the caller!
  - So **unnecessary thunks are being allocated**—bad!
- In effect we decide the precise function calling convention at runtime, on a per-function basis!

# Multiple Results

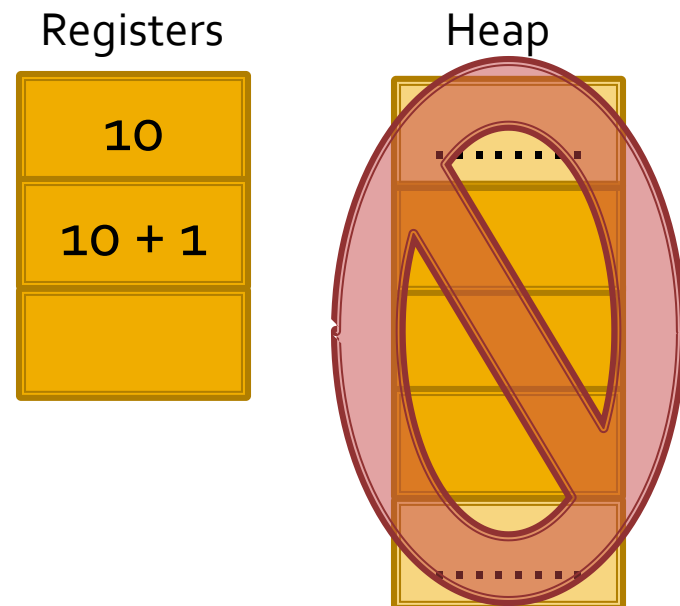
A “symmetric” source of slowness: returning multiple **results**:

```
silly x = (x, x + 1)
result = silly 10
```

What we **get**:



What we **want**:



# The Question

A vanilla functional language like GHC's current Core language doesn't let us express optimisations that act to remove these sources of inefficiency.

Is there a language in which we can express the differences in calling convention arising from:

1. Precisely how many arguments the function takes before doing "real work"
2. Whether or not we have decided to return results from a function in several registers

(...and hence express **optimisation** of these two properties!)

- In a way that the compiler can chew on all that information statically?
- That doesn't impose unbearable bureaucratic overhead when building terms and types?
- Which still has a functional flavour?

# A Possible Answer: Strict Core

**Idea:** just put the juicy operational information **in the type system of the compiler intermediate language**

- Invariants enforced via the type system are **stable** – if you break them, your program doesn't typecheck any longer!  
Easy to **detect compiler bugs**
- Types are a convenient place to encode all statically known information about a value:
  - Types (and their implied invariants) easily available to the code generator
  - Seamless treatment of invariants on **higher order arguments!**

**Slogan:** (intermediate language) types **are** calling conventions

# Strict Core Is Small!

## Binders

$b ::= x:\tau$  Value binding  
|  $\alpha:\kappa$  Type binding

## Types

$\tau, \nu, \sigma ::= \mathbf{T}$  Type constructors  
|  $\alpha$  Type variable references  
|  $\bar{b} \rightarrow \bar{\tau}$  Function types  
|  $\tau \nu$  Type application

## Atoms

$a ::= x$  Term variable references  
|  $\ell$  Literals

## Atoms In Arguments

$g ::= a$  Value arguments  
|  $\tau$  Type arguments

## Multi-value Terms

$e ::= \bar{a}$  Return multiple values  
|  $\mathbf{let} \overline{x:\tau} = e \mathbf{in} e$  Evaluation  
|  $\mathbf{valrec} \overline{x:\tau} = \bar{v} \mathbf{in} e$  Allocation  
|  $a \bar{g}$  Application  
|  $\mathbf{case} a \mathbf{of} \overline{p \rightarrow e}$  Branch on values

## Heap Allocated Values

$v ::= \lambda \bar{b}. e$  Closures  
|  $\mathbf{C} \bar{\tau}, \bar{a}$  Constructed data

# Strict Core Has Simple Types!

The type system should be very familiar, apart from a single changed production:

## Types

$\tau, \upsilon, \sigma$  ::=  $\mathbf{T}$  Type constructors  
          |  $\alpha$  Type variable references  
          |  $\bar{b} \rightarrow \bar{\tau}$  Function types  
          |  $\tau \upsilon$  Type application

$\langle \text{Int} \rangle \rightarrow \langle \text{Int}, \text{Int} \rangle$

$\langle \text{Int}, \text{Int} \rangle \rightarrow \langle \text{Int} \rangle$

$\langle a :: *, a \rangle \rightarrow \langle a \rangle$

If you have ***n* types in <brackets>** to the left (right) of a function arrow then ***n* arguments (results) are passed into (out of) the function call in registers**

# Strict Core Has Simple Expressions!

A strict functional language which we present in A-normal form:

## Multi-value Terms

```
e ::=  $\bar{a}$ 
    | let  $\bar{x}:\bar{\tau} = e$  in e
    | valrec  $\bar{x}:\bar{\tau} = \bar{v}$  in e
    | a  $\bar{g}$ 
    | case a of  $\bar{p} \rightarrow e$ 
```

`<1, x, 10>`

`let <x, y> = <1, 2>
in <x>`

Return multiple values  
Evaluation  
Allocation  
Application  
Branch on values

`valrec
 x = Just y
 y = \`

`id <Int, 10>`

- All allocation is done by **valrec**
- Stack frames are pushed by **let**
- **case** is only responsible for branching, not forcing of the scrutinee

# Thunks

- Thunks are **zero-argument functions**
  - Very natural – lambdas somehow induce a **delay** in evaluation
  - Similar to classic lazy lists in ML
- However, we want to **share the result of the function** between several calls
  - Add special cases to the operational semantics that **update** thunks with what they evaluate to

$$\{\tau, \dots, \tau\} \triangleq \langle \rangle \rightarrow \langle \tau, \dots, \tau \rangle$$
$$\{e\} \triangleq \backslash \langle \rangle . e$$

# From Haskell To Strict Core

Given this Haskell function:

```
twice :: (a -> a) -> a -> a
twice = \f -> \x -> f (f x)
```

We produce this Strict Core one:

```
twice :: <{<{a}> -> <a>}>
        -> <<{a}> -> <a>>
twice = \<f>. \<x>.
        f <> <{f <> <{x <>}>}>>
```

# Reprise

- *Mandatory* dynamic evaluatedness checks:
  - **Gone!** Can see **statically** if something is certainly evaluated
- *Mandatory* dynamic arity checks:
  - **Gone!** Can see **statically** how many arguments a functional value is expecting
- Returning several things in registers:
  - **Expressible!**
- **But** the functions I'm getting directly from Haskell make *little or no use* of these capabilities!

# Simple Equational Optimisations

Remember this?

```
twice = \<f>. <\<x>.  
  f <> <\{f <> <\{x <>}\>\>>
```

Apply **eta-contraction** to remove redundant thunk allocation:

$$\{x \ \langle \rangle\} \triangleq \backslash \langle \rangle . x \ \langle \rangle = x$$

# Arity Definition Site Analysis 1

```
twice = \<f>. <\<x>. ...>
```

Hang on - we take an argument and then **immediately return a function**? That's stupid! **Let's optimize:**

```
twice'
```

```
:: <{\<a> -> <a>}, {a}> -> <a>
```

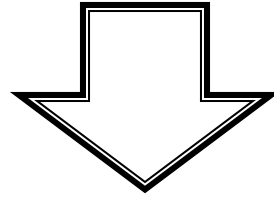
```
twice' = \<f, x>. ...
```

```
twice =
```

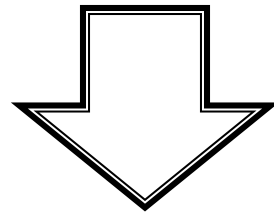
```
\<f>. <\<x>. twice' <f, x>>
```

# Worker/Wrapper In Action

```
twice <succ> <0>
```



```
(\<f>. <\<x>. twice' <f, x>>) <succ> <0>
```



```
twice' <succ, 0>
```

# Arity Definition Site Analysis 2

Q: Can we **always** improve the function arity like that?

```
h :: <Int> -> <<Int> -> <Int>>  
h = \<x>. let <z> = fib <x>  
           in \<y>. y + z
```

```
let h' = h <10000> in h' <2> + h' <1>
```

A: **No!** If we change the type of `h` to `<Int, Int> -> <Int>` then we won't be able to **share the partial application** any longer. Result: `fib <10000>` would be **run twice!**

# Arity Use Site Analysis

Q: Can we improve the arity of higher-order arguments?

```
h :: <<Int> -> <<Int> -> <Int>>>  
    -> <Int>  
h = \<f>. f <1> <2> + f <3> <4>
```

A: **Yes!** The function `h` only ever applies `f` to two arguments at once – and **no partial application is shared!** Let's optimize:

```
h' :: <<Int, Int> -> <Int>>  
     -> <Int>  
h' = \<f'>. f' <1, 2> + f' <3, 4>  
h = \<f>. h' <\<x,y>. f <x> <y>>
```

# Constructed Product Result

```
silly :: <{Int}> -> <({Int}, {Int})>  
silly = \<x>. <(x, {x <> + 1})>
```

We take an argument and then **immediately** return a **product type**? We could cancel that with a use site! **Let's optimize:**

```
silly' :: <{Int}> -> <{Int}, {Int}>  
silly' = \<x>. <x, {x <> + 1}>
```

```
silly = \<x>. let <y, z> = silly' <x>  
              in <(y, z)>
```

# Strictness

```
i :: <{Bool}> -> <Int>  
i = \<b>. case b <> of True -> <...>  
                        False -> <...>
```

We have a {thunked} argument that we **immediately evaluate**? That's stupid! **Let's optimize:**

```
i' :: <Bool> -> <Int>  
i' = \<b'>. case b' of True -> <...>  
                        False -> <...>  
  
i = \<b>. i' <b <>>
```

# Deep Unboxing

```
i :: <{({Int}, {Int})}> -> <Int>
i = \<p>. case ... of
  True -> <1>
  False -> case p <> of
    (x, y) -> <x <> + y <>>
```

This is more subtle! If `i` ever evaluates `p` then it certainly evaluates both components of the pair. **Let's optimize:**

```
i' :: <{Int, Int}> -> <Int>
i' = ...
i = ...
```

# Conclusions

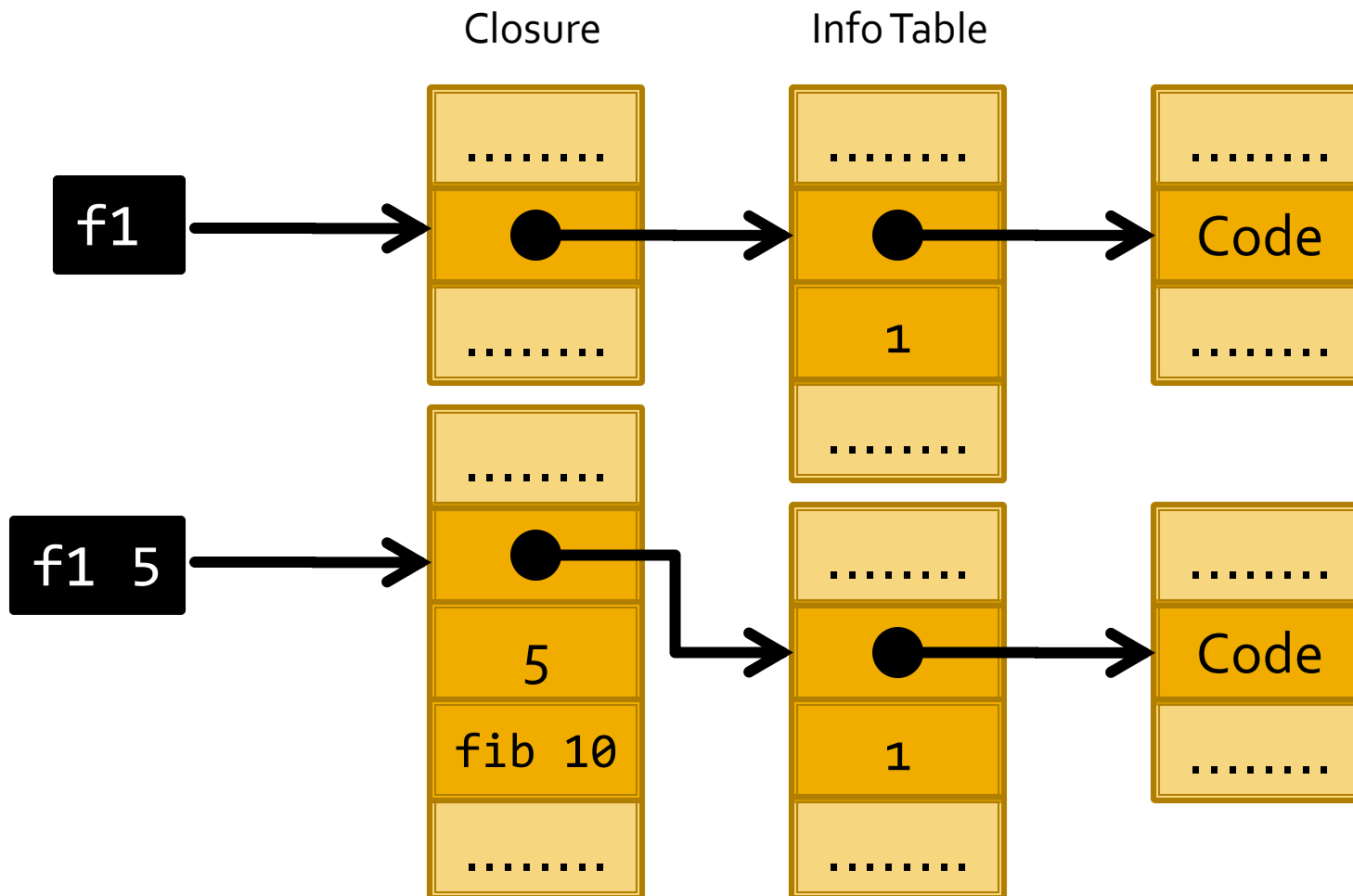
- Identifying **types** with **calling conventions** *looks* like a win, optimization-wise
  - Entirely new optimisation opportunities (arity use-site analysis, deep unboxing)
  - Existing *ad hoc* optimizations put on a sound footing
- A **nice point in the design space**: we couldn't seem to get much more optimisation opportunity without adding a lot of additional complexity
- Surprising (to us): the best choice for the intermediate language of a compiler for a lazy language is **not** *itself* a lazy language
- Exciting direction for future work: push the ability to write strict programs into Haskell!

# Extra Material

---

# Representing Functions

```
f1 = \y -> let x = fib 10 in \z -> x + y + z
```



# Examples

Expressing the number of **results** we expect:

```
duplicate :: <Int> -> <Int, Int>  
duplicate = \<x> -> <x, x>
```

Expressing the number of **arguments** we expect:

```
add2 :: <Int, Int> -> <Int>  
add2 = \<x, y> -> <x + y>
```

Polymorphism:

```
id :: <a :: *, a> -> <a>
```

# Example Translation 2

```
twice :: <{<{a}> -> <a>}>  
      -> <<{a}> -> <a>>  
twice = \<f> -> <\<x> ->  
        f <> <{f <> <{x <>}}>>>
```

Removing the (notational) line noise:


```
twice :: {{a} -> a}  
      -> {a} -> a  
twice = \<f> -> <\<x> ->  
        f <> {f <> {x <>}}>
```

# Let's Optimize!


- I've shown you some of the calling conventions we would **like** to have for our Haskell source functions
- The straightforward translation from Haskell will **not** give you these optimized calling conventions directly
  - Instead, we need to have some optimisations that improve calling conventions through program transformation

# Worker/Wrapper In Action

```
case silly <{10}> of  
  (x, y) -> <x + y>
```



```
case let <a, b> = silly' <{10}>  
  in <(a, b)> of  
  (x, y) -> <x + y>
```



```
let <a, b> = silly' <{10}>  
in <a + b>
```