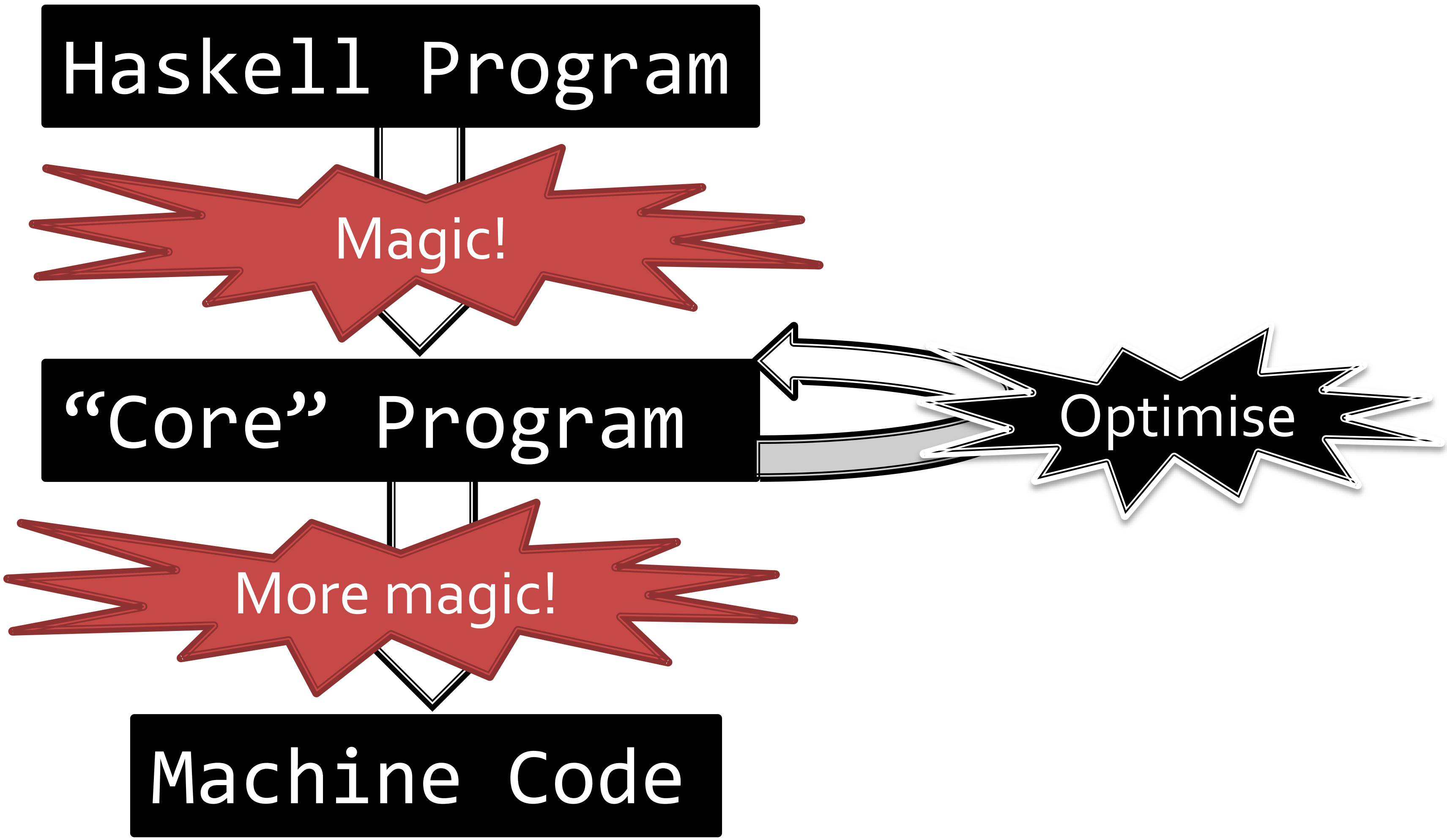
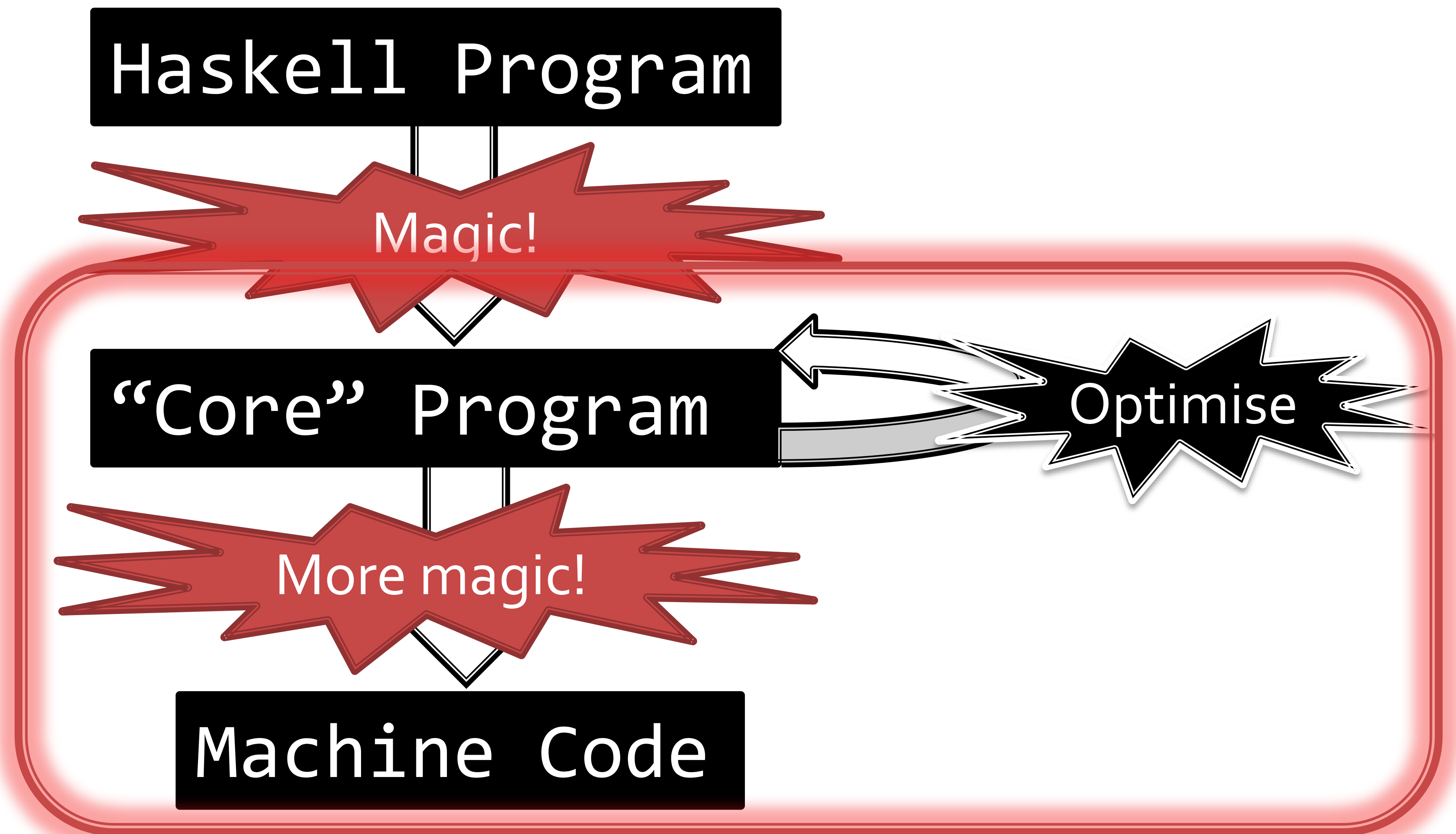
Max Bolingbroke
Simon Peyton Jones

Types Are Calling Conventions

The Glasgow Haskell Compiler



The Glasgow Haskell Compiler



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:

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
```

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
```

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
```

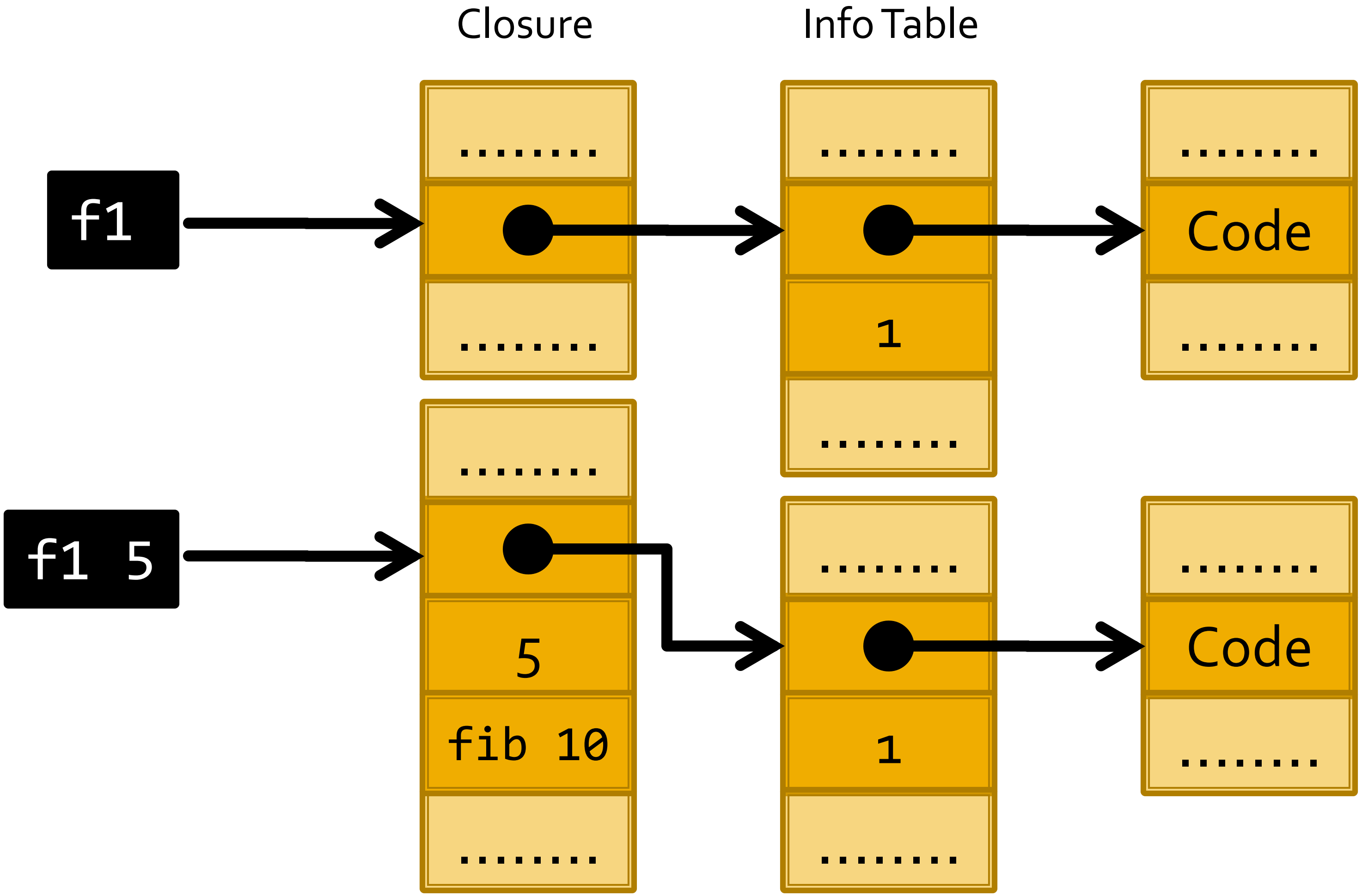
Zero!
```
f0 = let x = fib 10 in \y -> \z -> x + y + z
```

Arity is the number of arguments we can apply before the function does some “real work”

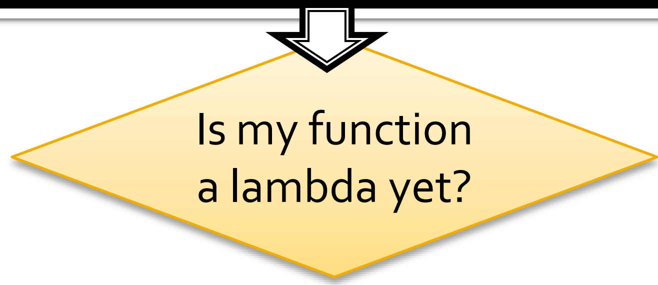
Arity depends on the **implementation** of the function, not its type!

Representing Functions

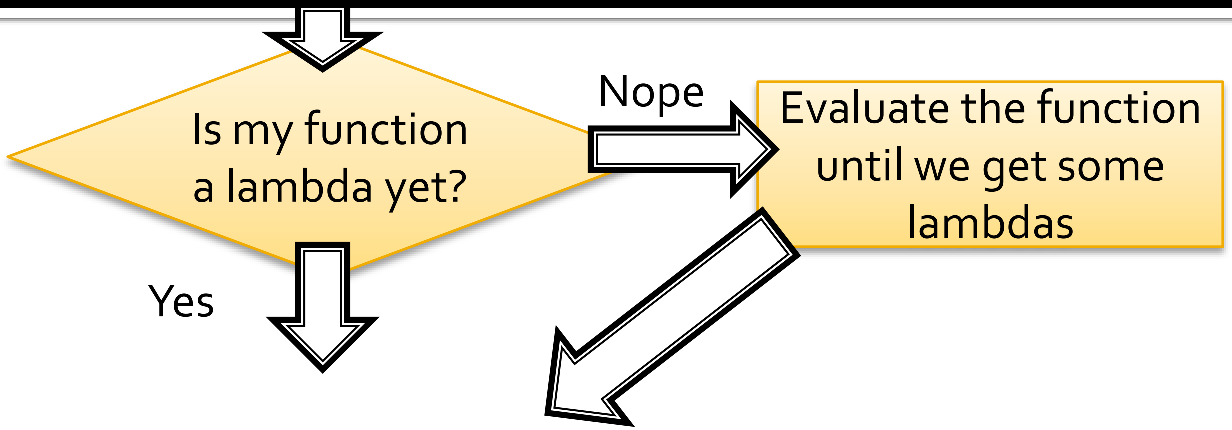
```
f1 = \y -> let x = fib 10 in \z -> x + y + z
```



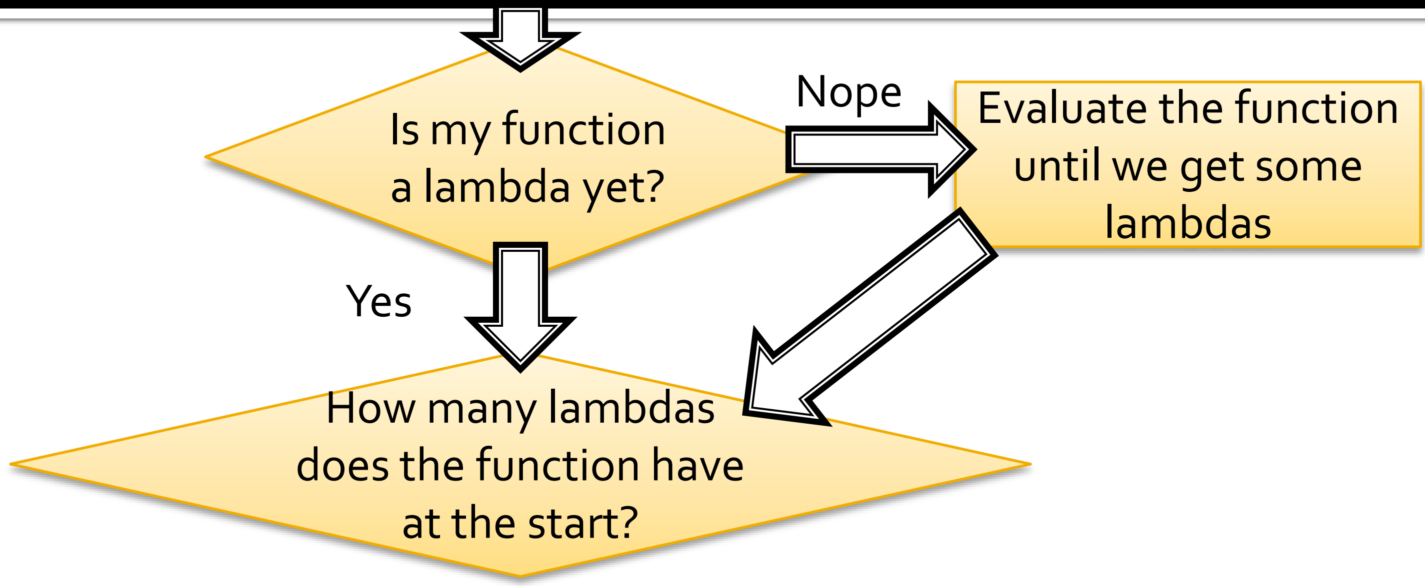
Function Application Today



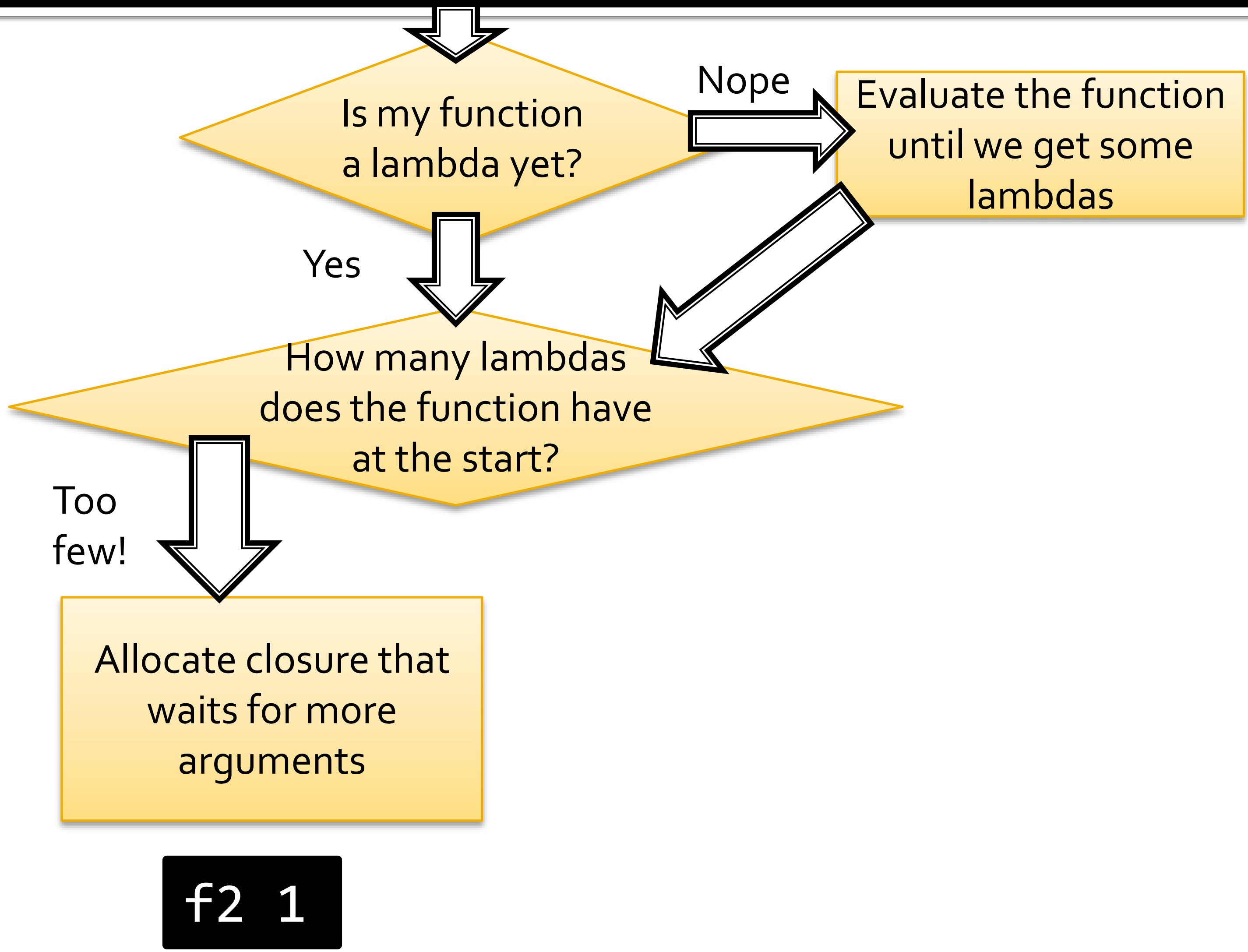
Function Application Today



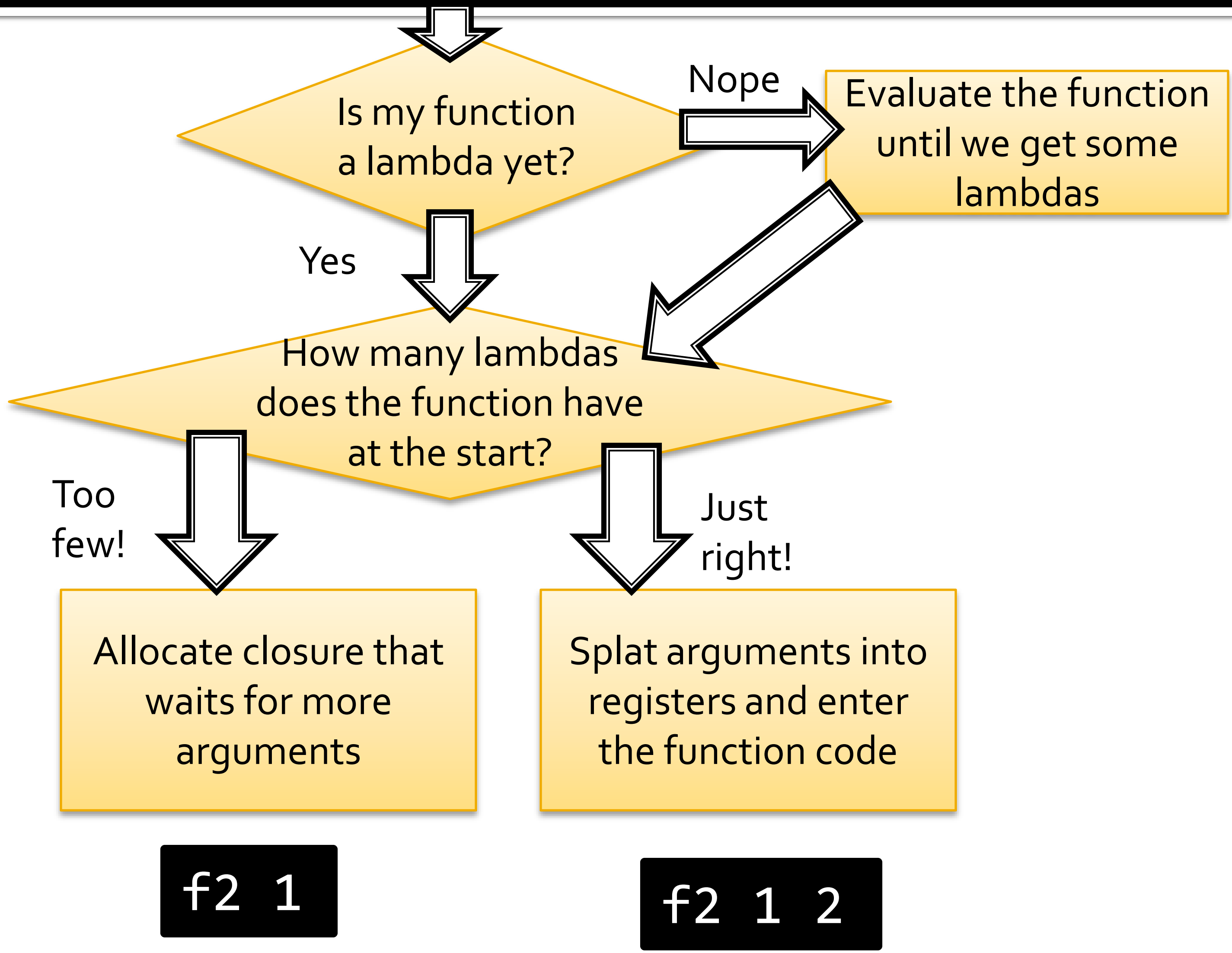
Function Application Today



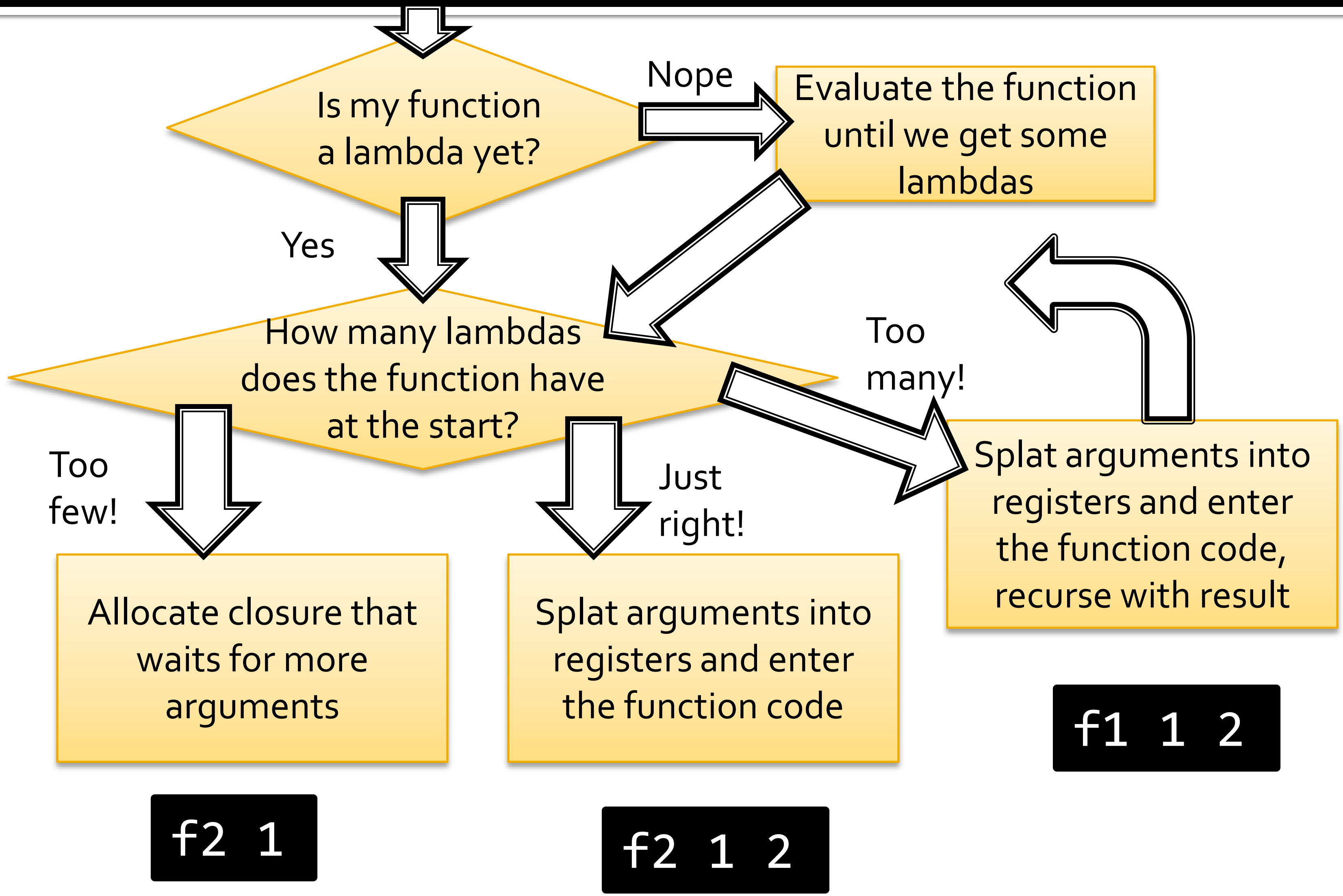
Function Application Today



Function Application Today



Function Application Today



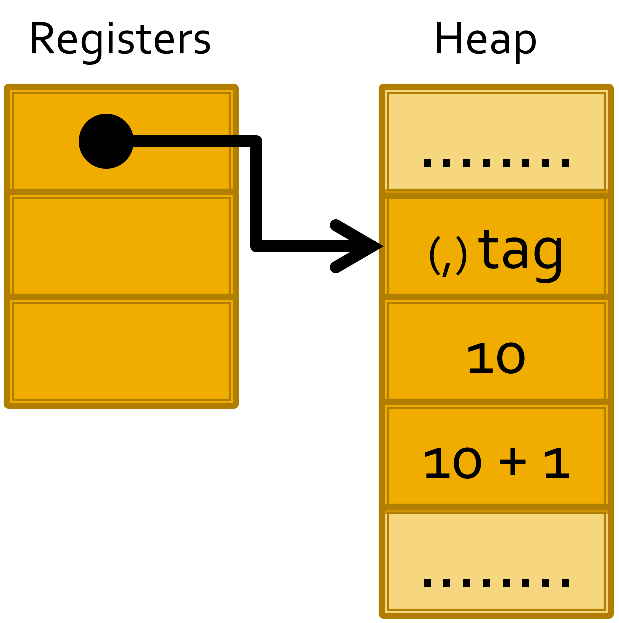
What Sucks

- 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!

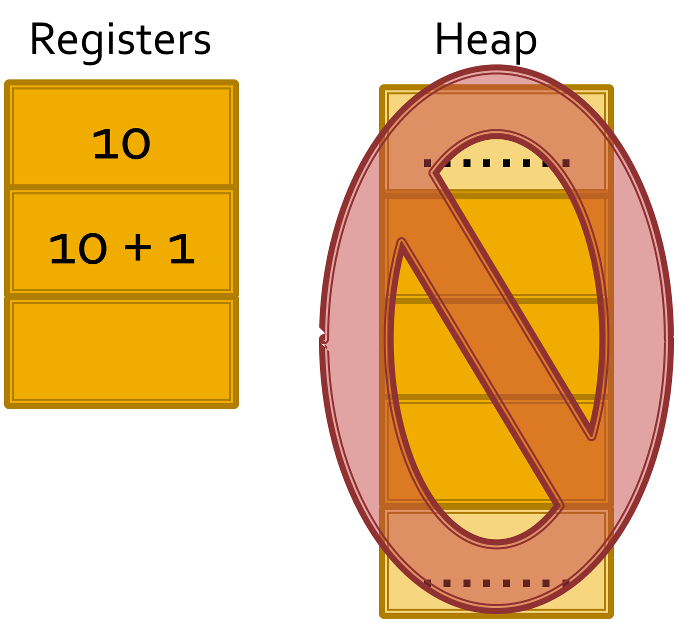
Multiple Results

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

What we **get**:



What we **want**:



The Idea

Key idea: put juicy operational information relating to *arity* , *thunking* and *unboxing* **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

How We Do It: Strict Core

Binders

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

Types

τ, υ, σ	$::=$	$\mathbf{T}$	Type constructors
	$\mid$	α	Type variable references
	$\mid$	$\bar{b} \rightarrow \bar{\tau}$	Function types
	$\mid$	$\tau\ \upsilon$	Type application

Basically: 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**

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>
```

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

$$\{\tau,\ldots,\tau\} \triangleq \langle\rangle \rightarrow \langle\tau,\ldots,\tau\rangle$$

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

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)>
```

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>
```

Arity Definition Site Analysis 1

```
g2 :: <Int> -> <<Int> -> <Int>>
g2 = \<x>. <\<y>. ...>
```

We take an argument and then **immediately return a function**? That's stupid! **Let's optimize:**

```
g2' :: <Int, Int> -> <Int>
g2' = \<x, y>. ...

g2 = \<x>. <\<y>. <g2' <x, y>>>
```

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>>
```

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 big win**, optimization-wise
  - Entirely new optimisation opportunities (arity use-site analysis, deep unboxing)
  - Existing *ad hoc* optimizations put on a sound footing
- You can have a simple, strong **operational model** and syntactic **beauty** at the **same time!**
- 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!