Optimising Functional Programming Languages

Max Bolingbroke,
Cambridge University CPRG Lectures 2010

Objectives

- Explore optimisation of functional programming languages using the framework of *equational rewriting*
- Compare some approaches for *deforestation* of functional programs

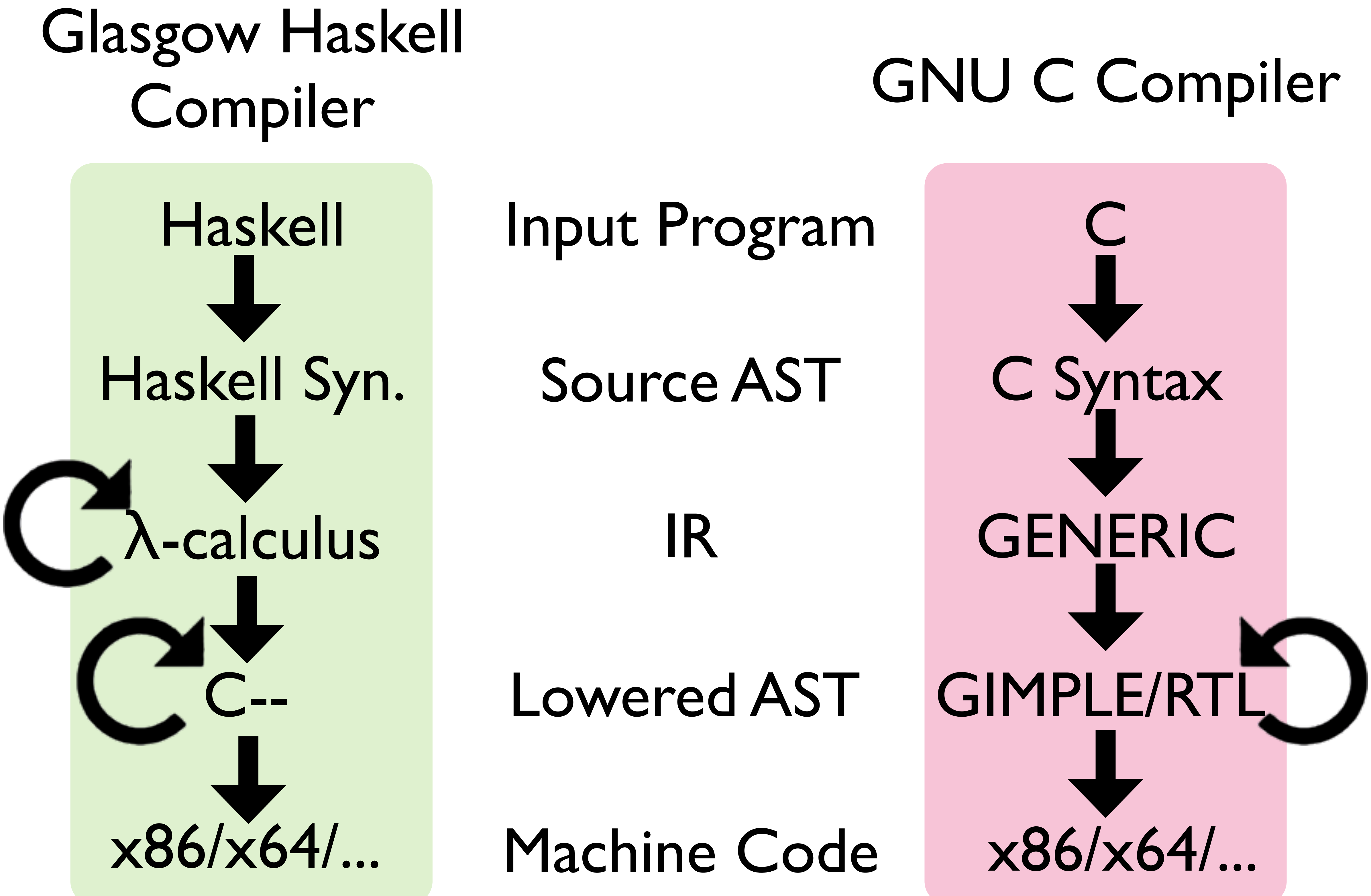
Making FP fast is important

- The great promise of functional programming is that you can write **simple**, **declarative** specifications of how to solve problems

```
wordCount :: String -> Int
wordCount = length . words

sumSquares :: String -> Int
sumSquares = sum . map square . words
  where square x = x * x
```

- Unfortunately, simple and declarative programs are rarely **efficient**
- If we want functional programming to displace the imperative style it needs to be somewhat fast

FP vs Imperative Optimisation



Pure λ-calculus is almost embarrassingly easy to optimise

- “Optimisation” consists of applying rules derived from the axioms of the calculus
- Things are much more complicated if the language has **impure** features such as reference cells (sorry, ML fans!)

λ-calculus

Pure | **Pure, CBV** | **Impure**

```
(\x -> e1) e2   ==>   let x = e2 in e1
```

```
case Just e1 of
  Just x -> e2    ==>   let x = e1 in e2
  Nothing -> e3
```

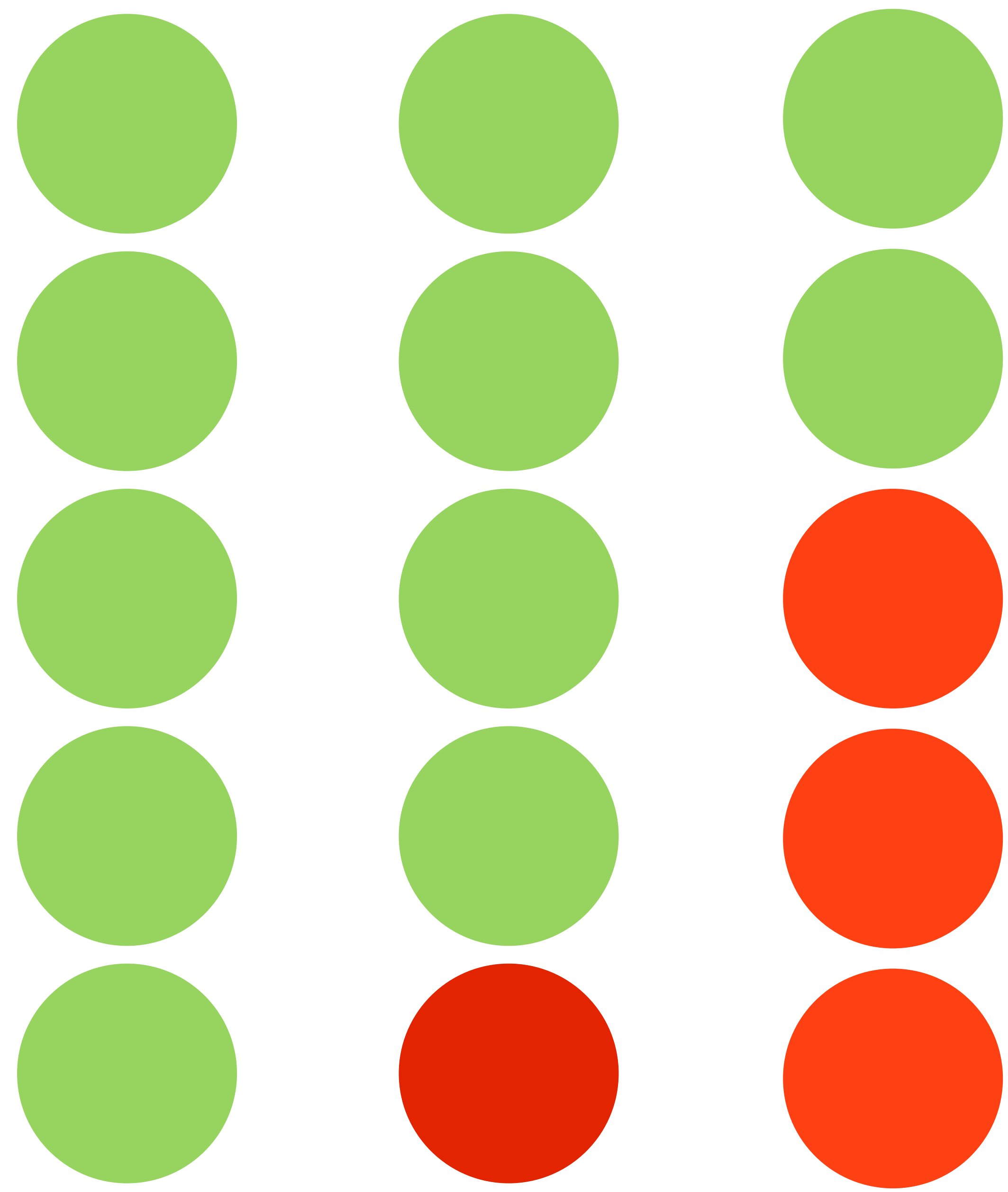
(x n.f. in e2)

```
let x = e1 in         let y = e2 in
let y = e2 in   ==>   let x = e1 in
e3                    e3
```

```
let x = e1            let x = e1
in ... x ...    ==>   in ... e1 ...
```

(x n.f. in e2)

```
let x = e1
in e2           ==>   e2
```



Artificial example of equational optimisation

```
let fst = \pair -> case pair of (a, b) -> a
    (.) = \f g x -> f (g x)
in (\y -> y + 1) . fst
```

(desugar operator)

```
let fst = \pair -> case pair of (a, b) -> a
    (.) = \f g x -> f (g x)
in (.) (\y -> y + 1) fst
```

(inline)

```
(\f g x -> f (g x)) (\y -> y + 1) (\pair -> case pair of (a, b) -> a)
```

(β-reduce)

```
let f = \x -> x + 1
    g = \pair -> case pair of (a, b) -> a
in \x -> f (g x)
```

(inline)

```
\x -> (\y -> y + 1) ((\pair -> case pair of (a, b) -> a) x)
```

(β-reduce)

```
\x -> let y = case x of (a, b) -> a
      in y + 1
```

(inline)

```
\x -> (case x of (a, b) -> a) + 1
```

(+ is strict)

```
\x -> case x of (a, b) -> a + 1
```

Equational optimisation is the bread and butter of a functional compiler

- Equational optimisation is the **number one** most important optimisation in a functional language compiler
- Inlining to **remove higher order functions** (e.g. in the arguments to the composition (.) function) is a particularly large win
  - Remove need to allocate closures for those functions
  - Eliminates some jumps through a function pointer (which are hard for the CPU to predict)
  - Allows some intraprocedural optimisation

Simple equational optimisation is not sufficient

Consider the following reduction sequence:

```
map (\y -> y + 1) (map (\x -> x + 1) [1, 2, 3, 4, 5])

map (\y -> y + 1) [2, 3, 4, 5, 6]

[3, 4, 5, 6, 7]
```

Input list

Intermediate list. Is this **really** necessary?

Output list

Idea: use higher level equations to optimise!

- We could build some facts about library functions into the compiler
- These can be in the form of extra equations to be applied by the compiler wherever possible, just like those derived from the axioms of λ-calculus

Example

Before, we had this expression, which allocates a useless intermediate list:

```
map (\y -> y + 1) (map (\x -> x + 1) [1, 2, 3, 4, 5])
```

However, if the compiler realises that:

```
∀ f g xs. map f (map g xs) = map (f . g) xs
```

It can then spot the (inefficient) original expression at compile time and equationally rewrite it to:

```
map ((\y -> y + 1) . (\x -> x + 1)) [1, 2, 3, 4, 5]
```

- Since there is only one call to `map` there is no intermediate list
- If `f` and `g` have side effects this rule isn't always true - **purity pays off**

Removing intermediate data is important for a FP compiler

- In a purely functional programming language, you can **never update** an existing data structure
  - Instead, the program is constantly allocating **brand new** data structures
  - A whole family of optimisations known as **deforestation** have sprung up to remove intermediate data structures (we just saw a very simple deforester)

Deforestation in practice

- Naively you might imagine that you need the compiler to know (at least) one equation for all possible pairs of composed functions (`map` of a `map`, `sum` of a `map`, `map` of a `enumFromTo`, etc.)
- The main implementation of the Haskell programming language implements a type of deforestation called **foldr/build fusion** based on a **single** equational rewrite rule
  - This is a (much) more general version of the map/map fusion I showed earlier
  - Knowing just a **single** equation, the compiler is able to deforest compositions of all sorts of list functions!

```
sumSq x = sum (map (\x -> x * x) (enumFromTo 1 x))
```

(deforestation)

```
sumSq x = if 1 > x then 0 else go 1
  where go y = if y == x then y * y else (y * y) + go (y + 1)
```

n.b: no lists - instead, we have a simple loop!

`foldr`/`build` fusion

The idea (Gill et al., FPLCA 1993):

1. Write all your **list consumers** (`sum`, `length`, etc.) by using the `foldr` function
2. Write all your **list producers** (e.g. `enumFromTo`) by using the `build` function
3. Provide a clever equational optimisation rewriting an application of a `foldr` to `build` (i.e. a consumer to a producer), which will cause the intermediate list to be removed

Writing a foldr list consumer

In case you've forgotten:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n []       = n
foldr c n (x:xs) = c n (foldr c n xs)
```

Intuitively, `foldr c n` on a list replaces all the cons `(:)` in the list with `c` and nil `[]` with `n`:

```
foldr c n (e1 : e2 : ... : em : []) = foldr c n ((:) e1 ((:) e2 (... ((:) em []))))
                                    =           c    e1 (c   e2 (... (c   em n )))
```

Lots of useful list consumers can be written as a `foldr`:

```
sum :: [Int] -> Int
sum = foldr (\x y -> x + y) 0
```

```
sum (1 : 2 : 3 : [])
```

(inline)

```
foldr (\x y -> x + y) 0 (1 : 2 : 3 : [])
```

(`foldr` replaces cons and nil in the list)

```
1 + 2 + 3 + 0
```

Lots of useful list consumers can be defined using `foldr`

Another example:

```
length :: [a] -> Int
length = foldr (\_ y -> y + 1) 0
```

```
length (1 : 2 : 3 : [])
```

↓ (inline)

```
foldr (\_ y -> 1 + y) 0 (1 : 2 : 3 : [])
```

↓ (`foldr` replaces cons and nil in the list)

```
1 + 1 + 1 + 0
```

Another (more complicated) consumer:

```
unzip :: [(a, b)] -> ([a], [b])
unzip = foldr (\(a,b) (as,bs) -> (a:as,b:bs)) ([],[])
```

Writing a build list producer

The build function is apparently trivial:

```
build g = g (:) []
```

The real magic is in the type signature:

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
```

You might be wondering what that forall means.

Don't worry! You've secretly used it before:

```
id :: forall a. a -> a
map :: forall a b. (a -> b) -> [a] -> [b]
foldr :: forall a b. (a -> b -> b) -> b -> [a] -> b
(.) :: forall a b c. (b -> c) -> (a -> b) -> a -> c
```

The types written above are just the normal types for those functions, but with the forall quantification written in **explicitly**.

Writing a `build` list producer

```
id :: forall a. a -> a
map :: forall a b. (a -> b) -> [a] -> [b]
foldr :: forall a b. (a -> b -> b) -> b -> [a] -> b
(.) :: forall a b c. (b -> c) -> (a -> b) -> a -> c
```

The funny thing about these function types is that the `forall` quantification is always on the "left hand side" of the type.

- This is known as **rank-1** polymorphism
- The **person calling the function gets to choose what** `a`, `b`, `...` **are**
- For example, in an expression like `id 10`, the caller has chosen that `a` should be `Int`

Writing a `build` list producer

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
```

In `build`, the `forall` quantification is **nested within the argument type**.

- This is known as **rank-2** polymorphism
- **The function itself gets to choose what `b` is**
- (Types inferred by Hindley-Milner are always rank-1)

`build (\c n -> 1 : c 2 n)` ✗ Does not typecheck!

- `1 : c 2 n` requires that `c` returns a `[Int]`
- However, the rank-2 type enforces that all you know is that `c` returns a value of *some type that* `build` *chooses* (called `b`), which may or may not be `[Int]`!

`build (\c n -> c 1 (c 2 n))` ✓ Typechecks (builds a 2 element list)

Some intuition about `build`

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

- If we wrote list producers using `(:)` and `[]` all over the place, it would be hard for the compiler to spot and remove them if it wanted to stop an intermediate list being constructed
- Instead, **λ-abstract our list producer functions** over the "cons" and "nil" functions for building a list
  - Now, by simply *applying that function to different arguments* we are able to do make the producer do something other than heap-allocate a cons/nil
    - Just change the "cons" and "nil" we pass in
    - e.g. make "nil" be 0, and then have "cons" add 1 to its second argument (i.e. add 1 every time the producer tries to output a list cons cell) - this gives us the `length` function.
  - The `build` function takes something abstracted over the cons and nil, and "fills in" the real `(:)` and `[]`
    - (The rank-2 type ensures that our abstracted version hasn't cheated by using `(:)` and `[]` directly)

Writing a build list producer

```haskell
enumFromTo :: Int -> Int -> [Int]
enumFromTo from to = build (go from)
  where go from c n = if from > to then n else c from (go (from + 1) c n)
```

"Proof":

```haskell
enumFromTo from to = build (go from)
  where go from c n = if from > to then n else c from (go (from + 1) c n)
```

↓ (inline build)

```haskell
enumFromTo from to = go from (:) []
  where go from c n = if from > to then n else c from (go (from + 1) c n)
```

↓ (notice that c and n are invariant in the recursion)

```haskell
enumFromTo from to = go from
  where go from = if from > to then [] else from : go (from + 1)
```

So our enumFromTo **does** do the right thing. The version without build is easier to understand, but our deforestation equational rewrite will only understand list producers using build, so we use that version of enumFromTo.

The magic `foldr`/`build` rule

```
∀ c n g. foldr c n (build g) = g c n
```

- Intuitively:
  - Take a `g` which has been *abstracted over the cons and nil functions*
  - Where the list produced by `build g` is being immediately consumed by a `foldr c n`
  - Finally, instead of `build`ing to produce that intermediate list and then consuming it, just *instantiate the list "constructors" in the producer with the thing doing the consuming*

The magic foldr/build rule

```
∀ c n g. foldr c n (build g) = g c n
```

```
sum (enumFromTo 1 10)
```

⬇ (inline sum and enumFromTo)

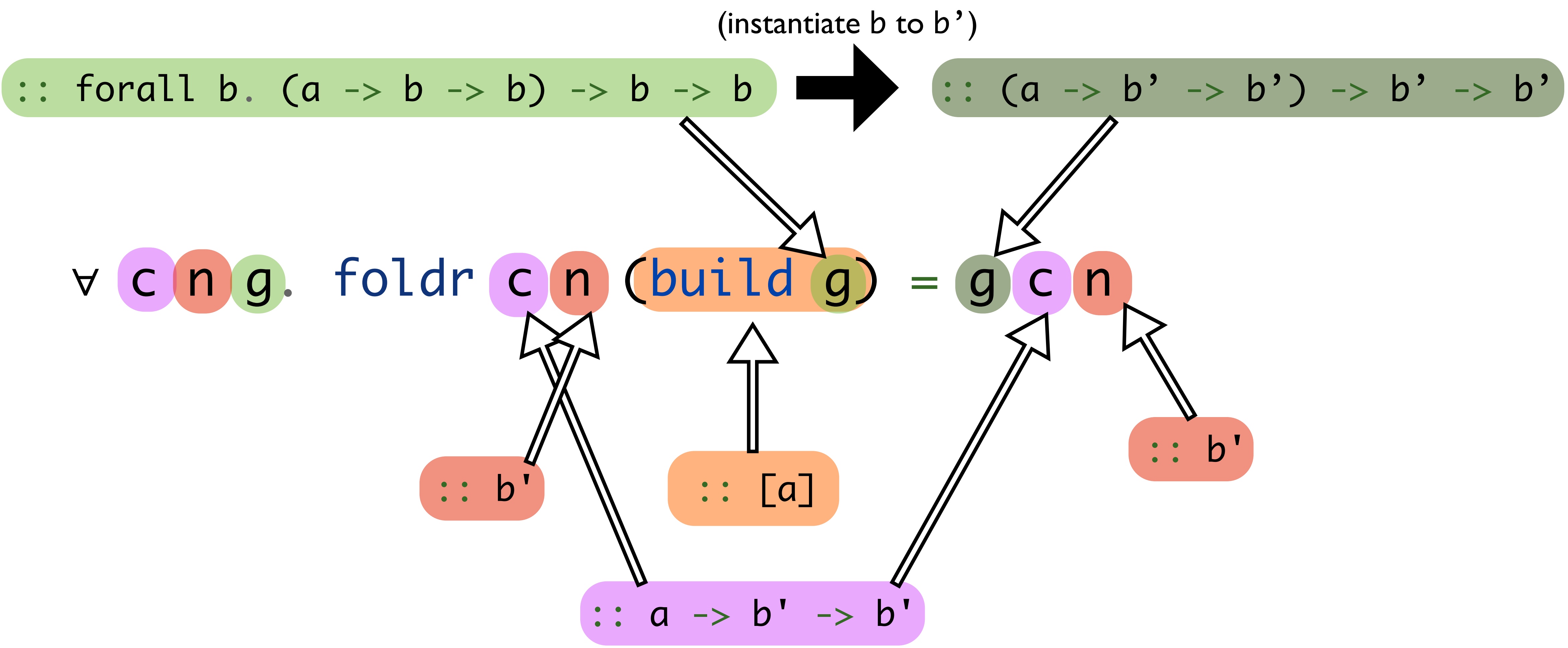
```
foldr (\x y -> x + y) 0 (build (go 1))
  where go from c n = if from > 10
                      then n
                      else c from (go (from + 1) c n)
```

⬇ (apply the foldr/build equation)

```
go 1
  where go from = if from > 10
                  then 0
                  else from + go (from + 1)
```

The intermediate list has been **totally eliminated**

The `foldr/build` equation is type correct



The rank-2 polymorphism in the type of `build` is essential!

- If it weren't polymorphic, we could only fuse if `b = b'`
- This would break most interesting deforestations
  - e.g. when deforesting `sum (enumFromTo 1 10)` we need `b = [Int]` and `b' = Int`

Functions which are both producers and consumers are defined using both `build` and `foldr`

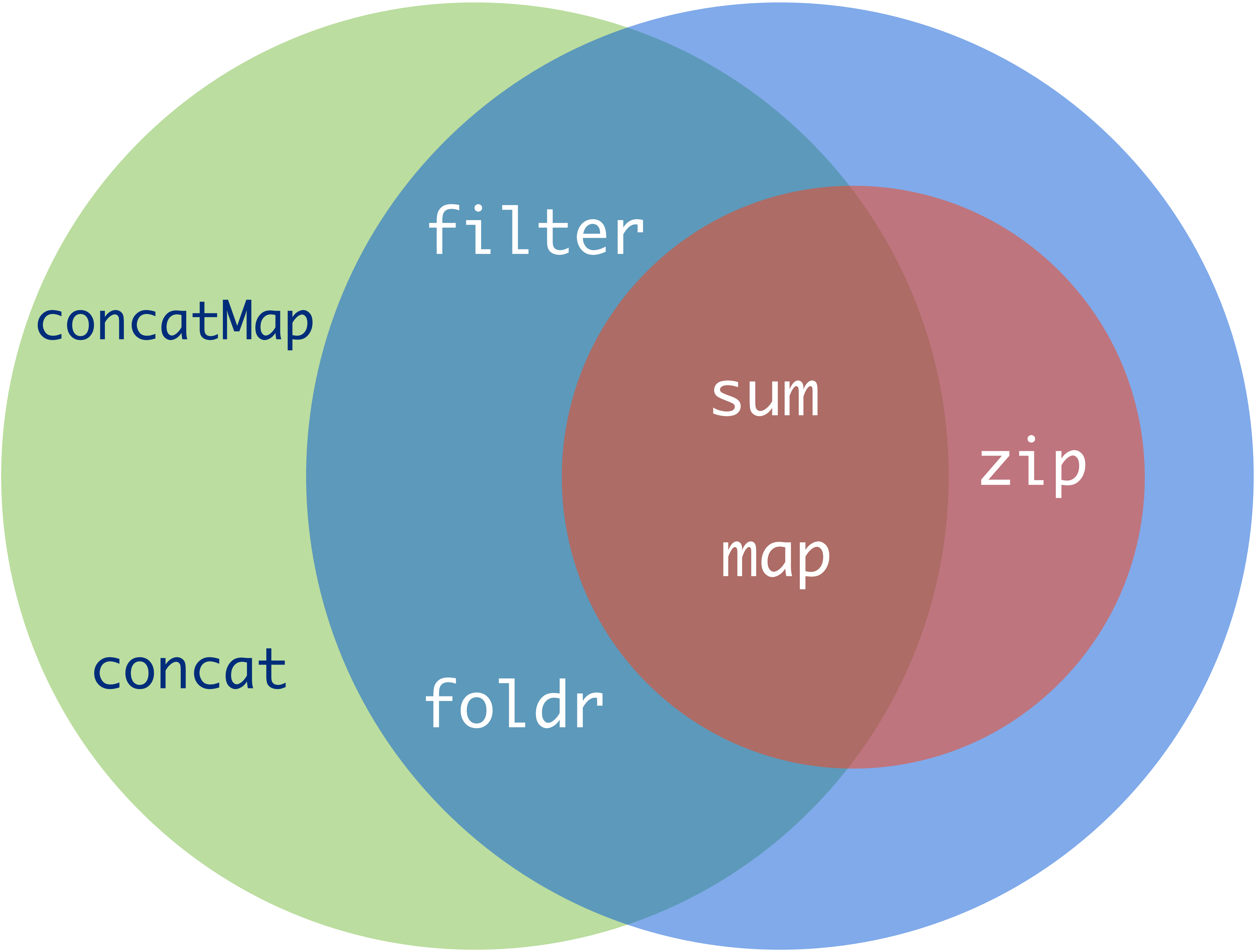
```
map :: (a -> b) -> [a] -> [b]
map f xs = build (\c n -> foldr (\x ys -> c (f x) ys) n xs)

(++) :: [a] -> [a] -> [a]
xs ++ ys = build (\c n -> foldr c (foldr c n ys) xs)
```

- It all looks very weird, but it works!
- Upshot is that you can eliminate superfluous intermediate lists (and hence reduce allocation) for compositions of *almost all* of the common list functions
- The `map`/`map` deforestation example I showed at the start is a special case of the `foldr`/`build` rule

Extensions and alternatives

- The `foldr`/`build` framework can be generalised to data types other than simple lists (but that is not so useful in practice)
  - The main issue with the framework is that `zip`-like functions (that consume two or more lists) cannot be deforested
- There is a categorically dual framework called `unfoldr`/`destroy` (Svenningsson, ICFP 2002) which **can** deal with such functions
  - However, it can turn non-terminating programs into terminating ones (i.e. the `unfoldr`/`destroy` rule is not actually an equivalence)
  - Fails to deforest some functions that *will* deforest with `foldr`/`build` (such as `filter` and `concatMap`)
- Yet another approach is Stream Fusion (Coutts et al., ICFP 2007), which relies on the equation `stream . unstream = id`
  - Can fuse everything that the above approaches can, except for `concatMap`

The deforestation landscape



foldr/build

stream/unstream

unfoldr/destroy

Deforestation by supercompilation

- There are many other approaches to deforestation, many of which don't use a simple equational rewriting
- One such method is **supercompilation** (Turchin, PLS 1986)
    - A supercompiler is based around an evaluator which is capable of evaluating expressions *containing free variables*

Deforestation by supercompilation

```
h0 xs = map (\x -> x + 1) (map (\x -> x + 2) xs)
```

(inline outer map)

```
h0 xs = case map (\x -> x + 2) xs of
          []     -> []
          (y:ys) -> y + 1 : map (\x -> x + 1) ys
```

(inline inner map)

```
h0 xs = case (case xs of
                []     -> []
                (z:zs) -> z + 2 : map (\x -> x + 2) zs) of
            []     -> []
            (y:ys) -> y + 1 : map (\x -> x + 1) ys
```

(case-of-case)

```
h0 xs = case xs of
          [] -> case [] of
                  []     -> []
                  (y:ys) -> y + 1 : map (\x -> x + 1) ys
          (z:zs) -> case z + 2 : map (\x -> x + 2) zs of
                      []     -> []
                      (y:ys) -> y + 1 : map (\x -> x + 1) ys
```

Deforestation by supercompilation

```
h0 xs = case xs of
          [] -> case [] of
                  []     -> []
                  (y:ys) -> y + 1 : map (\x -> x + 1) ys
          (z:zs) -> case z + 2 : map (\x -> x + 2) zs of
                      []     -> []
                      (y:ys) -> y + 1 : map (\x -> x + 1) ys
```

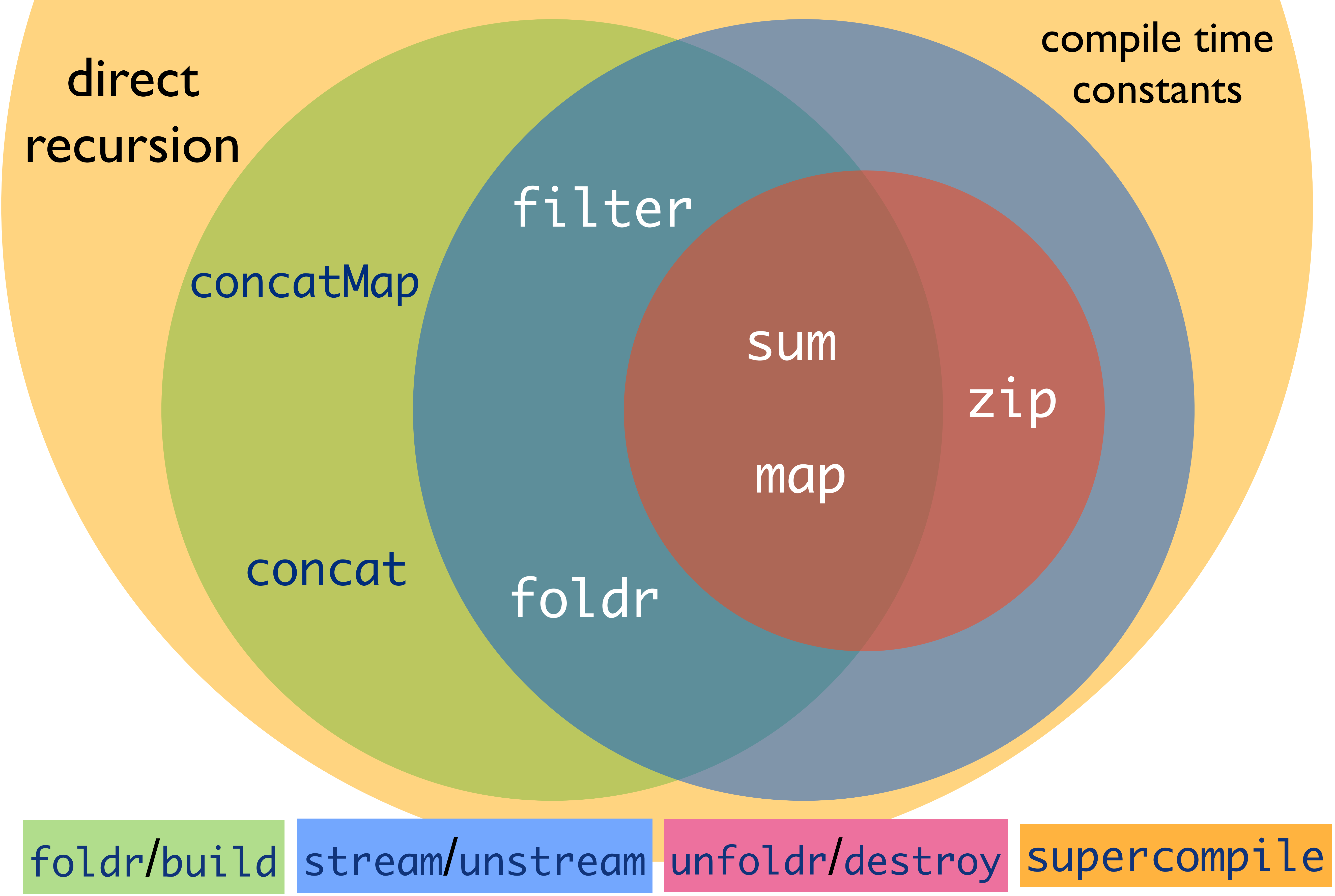
(evaluate both inner cases)

```
h0 xs = case xs of
          [] -> []
          (z:zs) -> z + 2 + 1 : map (\x -> x + 1) (map (\x -> x + 2) zs)
```

(tie back, since the case branch which recursively calls `map` is just a renaming of what we started with, and thoughtfully called `h0`)

```
h0 xs = case xs of
          [] -> []
          (z:zs) -> z + 2 + 1 : h0 zs
```

The deforestation landscape



Supercompilation

- Supercompilation is a very powerful transformation, which can achieve much more than just deforestation
  - No need to define your library functions in a stylised way (e.g. in `foldr`/`build` you carefully use `foldr` for everything)
  - Closely related to **partial evaluation**
- Currently, supercompilers are too slow to be practical (i.e. they can take on the order of *hours* to compile some simple examples)

Conclusion

- You **can** write beautiful, declarative functional programs which compile to very fast code!
- **Deforestation** is an important optimisation for achieving this, and can be achieved in practice using `foldr`/`build`
- The fact that we are optimising a pure and functional language makes such optimisations reliable and simple to do
    - Removing intermediate data structures from e.g. C programs is much harder (but possible!)
- All programs should be written in Haskell :-)

Further Reading

- Shrinking lambda expressions in linear time (Appel et al., JFP 7:5)
- Call-pattern specialisation for Haskell programs (Peyton Jones, ICFP 2007)
- The worker/wrapper transformation (Gill et al., JFP 19:2)
- The source code to GHC! (http://hackage.haskell.org/trac/ghc/)