Compiling Lazy Functional Languages

Why evaluation too early is *bad*

Suppose we have

```
loop x = if x > 0 then x else loop x
foo y z = if y > 0 then y else z
```

and consider the "program" `foo 1 (loop 0)` (assuming "standard" *call-by-value* or *eager evaluation*).

By *equational reasoning* (β-substitution) we ought to have

`foo 1 (loop 0)` $\equiv$ `if 1 > 0 then 1 else (loop 0)`

Yet, the first version does *not* terminate, while the second does.

Why?

Call-by-name

This suggests that we ought to pass arguments unevaluated, and evaluate expressions only when needed for computing the final result:

`foo 1 (loop 0)` $\Rightarrow$ `if 1 > 0 then 1 else (loop 0)` $\Rightarrow$ 1

This is known as *call-by-name*.

Call-by-name is inefficient

Consider

```
double x = x + x
main = double (3 * 7)
```

Call-by-value:

main ⇒ double (3 * 7) ⇒ double 21 ⇒ 21 + 21 ⇒ 42

Call-by-name:

main ⇒ double (3 * 7) ⇒ (3 * 7) + (3 * 7)
⇒ 21 + (3 * 7) ⇒ 21 + 21 ⇒ 42

Call-by-need or lazy evaluation

We would like to evaluate function arguments *at most once*. This is *call-by-need* or *lazy evaluation*.

Implementation idea:

- Represent program by a *graph*, thus allowing expressions to be *shared*.

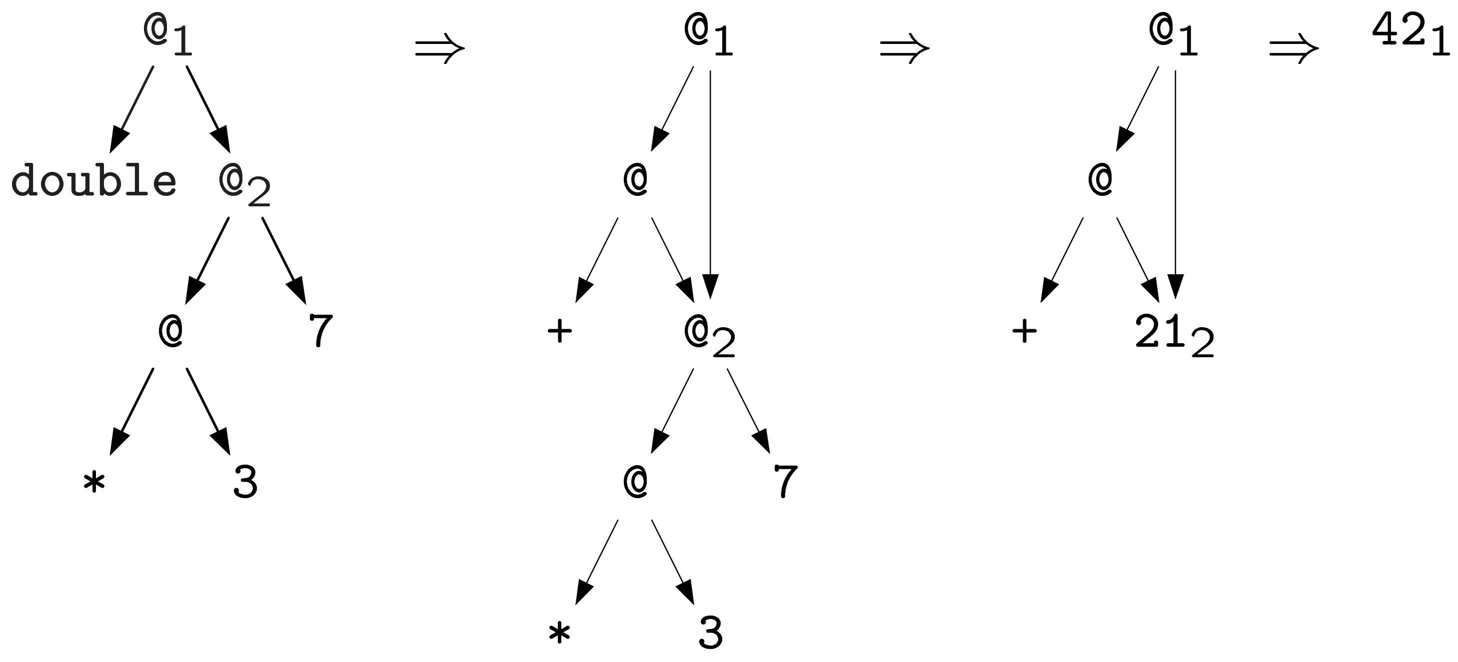
- When an expression has been evaluated, *replace* (update) it with the result to avoid re-computation.

This technique is known as *graph reduction*.

Lazy evaluation/graph reduction is an *efficient implementation technique* for achieving call-by-need or *normal order* semantics.

Graph reduction

Evaluation of `double (3 * 7)` through graph reduction:



(Subscripts indicate physical memory locations.)

Applications of lazy evaluation

Some uses of lazy evaluation:

- “Infinite” data structures.
- Attribute grammar evaluation.
- Dynamic programming.
- Circular programming — programming with “unknowns”.

Lazy evaluation example 1: Prime numbers (1)

Canonical example of use of "infinite" lists.

```
from :: Int → [Int]
from n = n : from (n+1)

take :: Int → [a] → [a]
take 0     _      = []
take _     []     = []
take (n+1) (x:xs) = x : take n xs
```

Lazy evaluation example 1: Prime numbers (2)

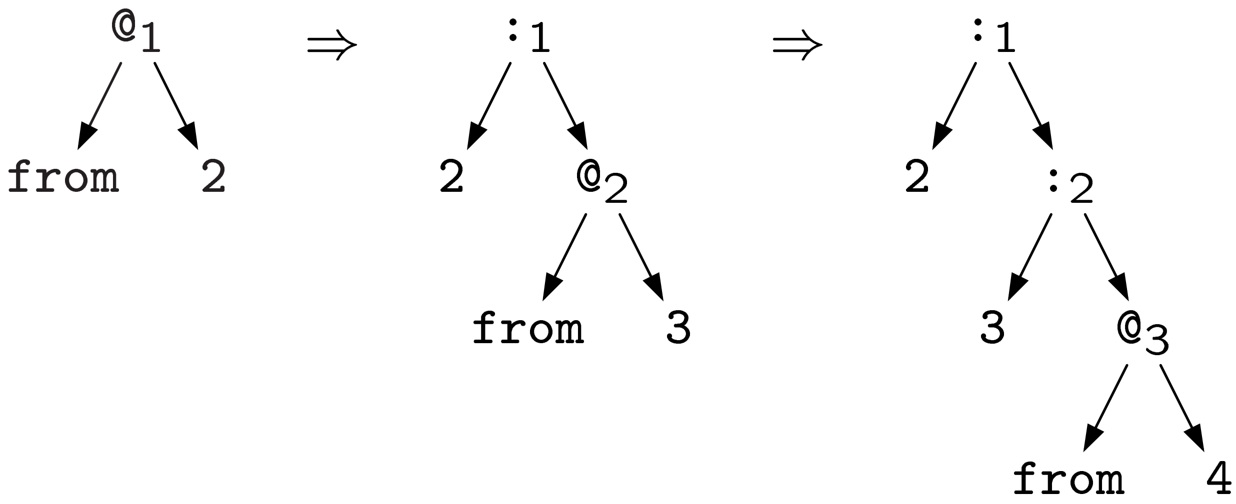
```
filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) =
    if p x then x : filter p xs else filter p xs

sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve (filter x_not_divisor xs)
    where
        x_not_divisor y = y `mod` x /= 0

primes = sieve (from 2)

main = take 100 primes
```

Lazy evaluation example 1: Prime numbers (3)



Note how *reducible expressions* or *redexes* are replaced by instances of the function body.

A redex is only evaluated when needed, and then only to *Weak Head Normal Form* (WHNF), e.g. until it is a list cell, but head and tail are not evaluated.

Lazy evaluation example 1: Prime numbers (4)

The evaluation of `primes` roughly goes like this
(`xnd` means `x_not_divisor`):

```
primes
⇒ sieve (from 2)
*
⇒ 2 : sieve (filter (xnd where x=2) (from 3))
*
⇒ 2 : 3 : sieve (filter (xnd where x=3)
                        (filter (xnd where x=2)
                                (from 3)))
*
⇒ 2 : 3 : 5 : sieve (filter (xnd where x=5)
                            (filter (xnd where x=3)
                                    (filter (xnd where x=2)
                                            (from 6))))
```

Lazy evaluation example 2: Spreadsheet (1)

Consider a simple spreadsheet evaluator:

	A	B	C
1	B2+A3	B3+A3	3
2	2	C1*A3	4
3	A2*C3	5	C1+C2

$\Longrightarrow$

	A	B	C
1	56	19	3
2	2	42	4
3	14	5	7

One possible evaluation order: . . . , C3, A3, B1, B2, A1.

Lazy evaluation example 2: Spreadsheet (2)

A simple evaluator can be constructed along the following lines:

```
type SpreadSheet = Array Expression
type Result      = Array Double
type Environment = Result

amap :: (a → b) → Array a → Array b
evalExp :: Environment → Expression → Double

evalSheet :: SpreadSheet → Result
evalSheet sheet = result
    where
        result = amap (evalExp result) sheet
```

Lazy evaluation example 2: Spreadsheet (3)

```
evalSheet :: SpreadSheet → Result
evalSheet sheet = result
    where
        result = amap (evalExp result) sheet
```

Note how `result` is used as the environment in the computation of itself. An example of *circular programming*.

The demand-driven nature of lazy evaluation takes care of finding a suitable evaluation order (if one exists).

Effectively, a *topological sort* according to the data dependences is carried out dynamically.

Compiled graph reduction

How can we implement graph reduction efficiently? Idea:

- Convert all functions to *supercombinators*, i.e. functions *without free variables*, using *lambda lifting*.

- Compile a *fixed* code sequence for each supercombinator that replaces (updates the *root* of) a reducible application (redex) of that function with an *instance* of the function body.

We will compile to the *G-machine*: a stack-based, virtual machine for graph reduction.

G-machine code can be interpreted or easily turned into native code.

Lambda lifting (1)

Consider `sieve` again:

```
sieve :: [Int] → [Int]
sieve (x:xs) = x : sieve (filter x_not_divisor xs)
    where
        x_not_divisor y = y ‘mod‘ x /= 0
```

The local function `x_not_divisor` has a free variable `x`.

Thus we *cannot* compile any fixed code sequence for it!

Lambda lifting (2)

Lambda lifting:

- Abstract out all free variables as *extra arguments*.
- Replace all calls to original function with the new supercombinator applied to the free variables.

```
$x_not_divisor x y = y `mod` x /= 0

$sieve (x:xs) = x : $sieve (filter ($x_not_divisor x) xs)
```

Note: partial application of supercombinators is how we represent *closures*. (There are other ways, though.)

Lambda lifting (3)

Here is one way to handle local, (mutually) recursive functions.

```
f x y = g 1 where
    g z = (x * z) : h (z + 1)
    h z = (y * z) : g (z + 1)
```

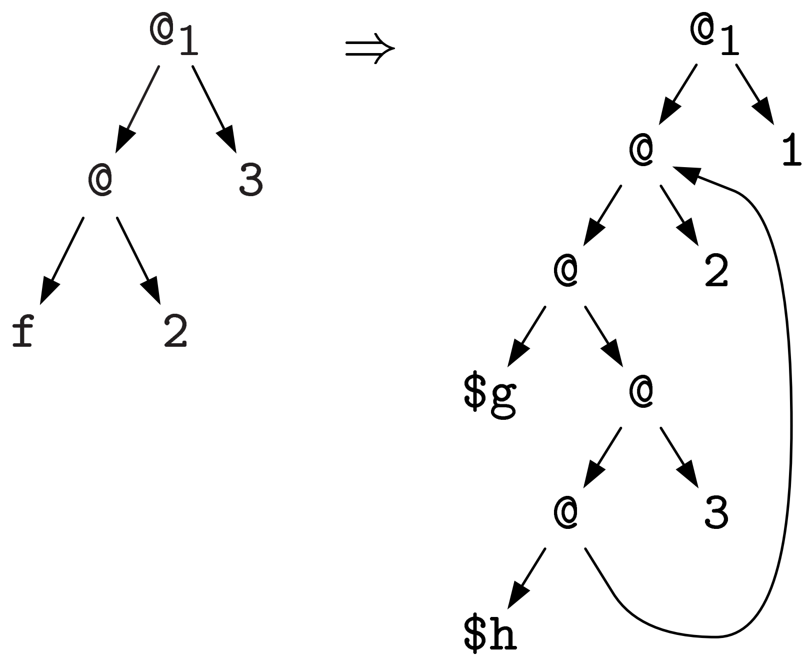
Supercombinator form:

```
$g h x z = (x * z) : h (z + 1)
$h g y z = (y * z) : g (z + 1)
$f x y = g 1 where
    g = $g h x
    h = $h g y
```

Note that the *recursive functions* are treated as free variables!

Lambda lifting (4)

Thus the instance of the body of `f` becomes a *circular graph*:



This captures the notion of a *fixed point* as the meaning of a recursive definition.

Lambda lifting (5)

Another way. We exploit that supercombinators have a name (known fixed address) and can refer to each other that way:

```
f x y = g 1 where
    g z = (x * z) : h (z + 1)
    h z = (y * z) : g (z + 1)
```

Supercombinator form:

```
$g x y z = (x * z) : $h x y (z + 1)
$h x y z = (y * z) : $g x y (z + 1)
$f x y = $g x y 1
```

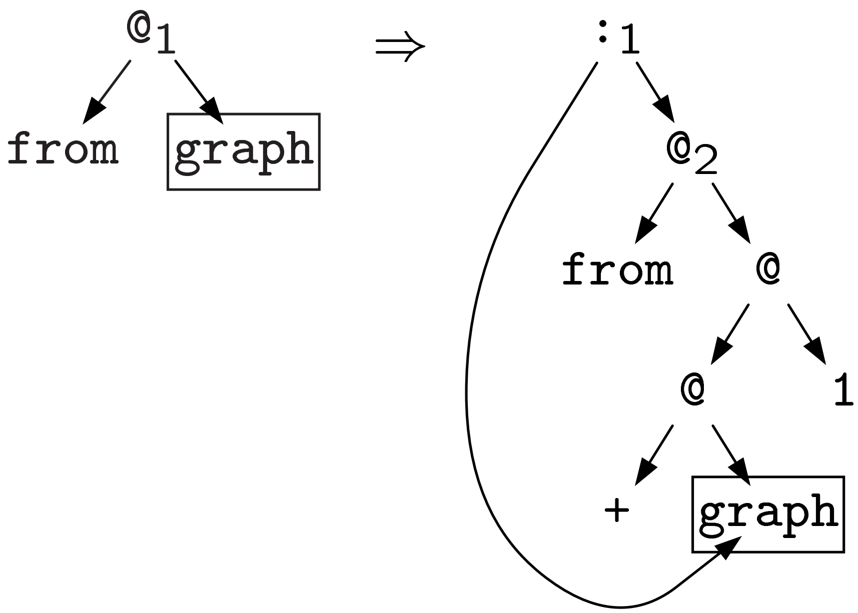
Note that the *union* of the free variables in `f` and `g` needs to be abstracted out!

The G-machine (1)

Recall the definition of `from`:

```
from n = n : from (n + 1)
```

The code for `from` should perform the graph transformation:



The G-machine (2)

G-code for `from`. On the stack: ptr. to arg., ptr. to redex root:

```
PUSHINT 1
PUSH 1          -- Argument
PUSHFUN $plus
MKAP            -- Build application node
MKAP
PUSHFUN $from
MKAP
PUSH 1          -- Argument
CONS            -- Build CONS-cell (list cell)
UPDATE 2        -- Update redex root
POP 1           -- Clean up stack (remove argument)
RETURN          -- Redex root in WHNF, done!
```