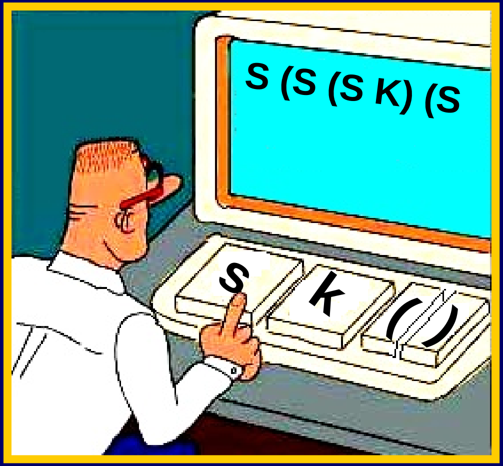
Three improvements to the Reduceron

Matthew Naylor and Colin Runciman
University of York

(Talk given at HFL, March 2009)

The Reduceron

A custom computer designed to run functional programs,



not restricted by conventional architectural constraints,

implemented on an FPGA using a functional language.

This talk

1. Graph reduction and widening the von Neumann bottleneck.

2. Three improvements to the Reduceron since September 2008.

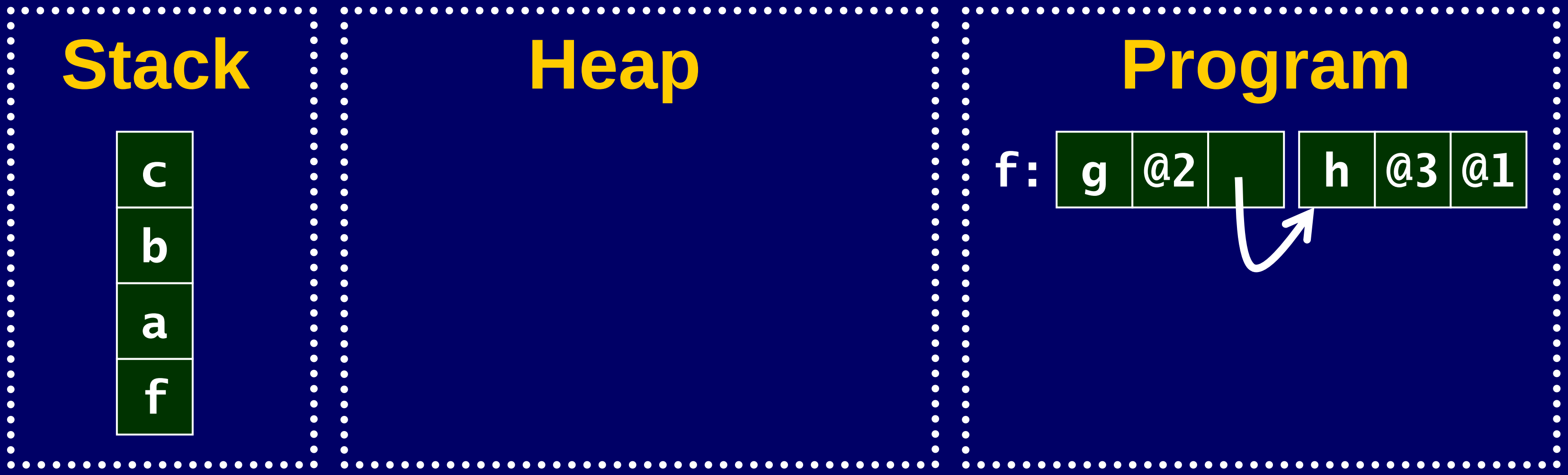
3. How the Reduceron is described.

Graph Reduction

Suppose that **`f`** is defined by

```
f x y z   =   g y (h z x)
```

where **g** and **h** are functions and the following machine-state arises during reduction.



Stack

c
b
a
f

Heap

Program

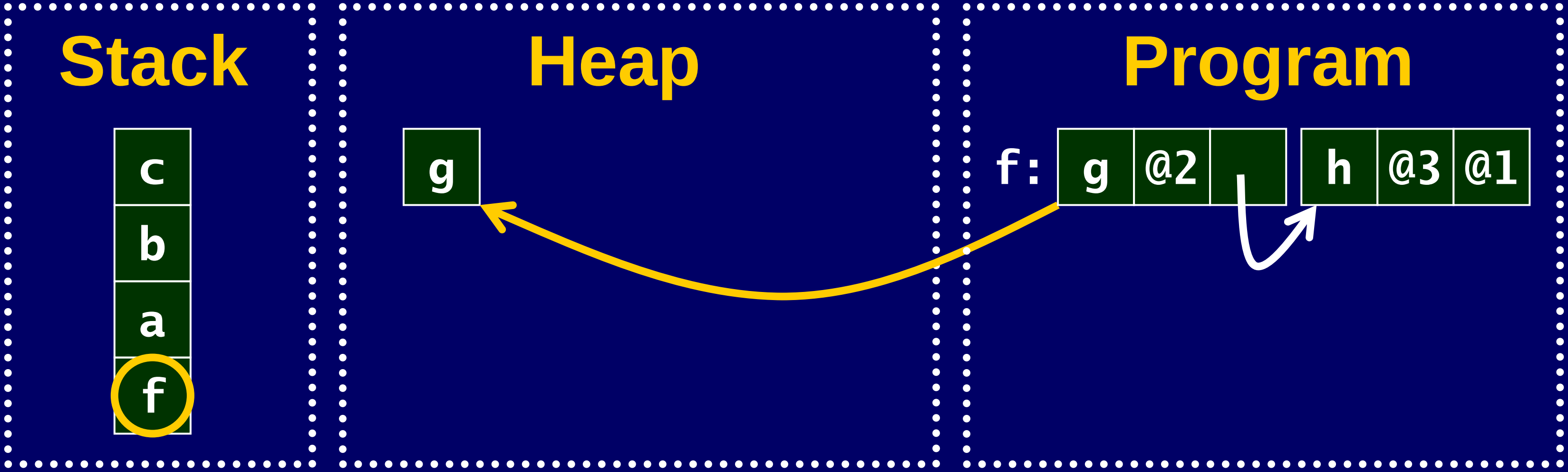
f: g @2 | h @3 @1

Graph Reduction

Operation:

```
f <- Stack[0]
g <- Code[f]
g -> Heap
```

Count: 3

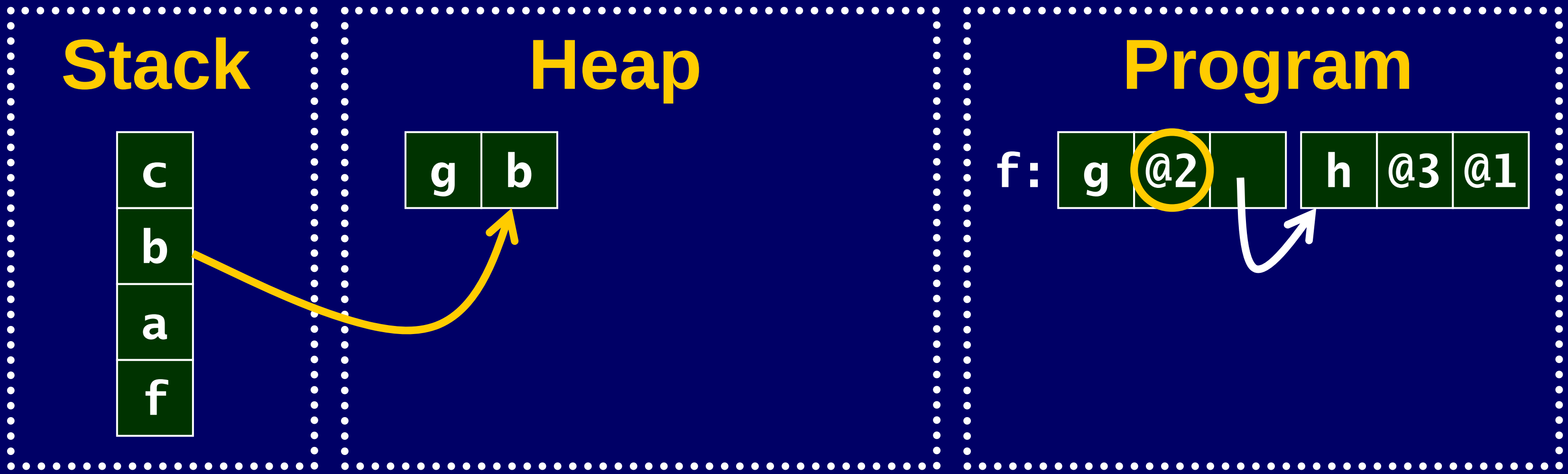


Graph Reduction

Operation:

```
arg <- Code[f+1]
b   <- Stack[arg]
b   -> Heap
```

Count: 6



Stack

c
b
a
f

Heap

g b

Program

f: g @2

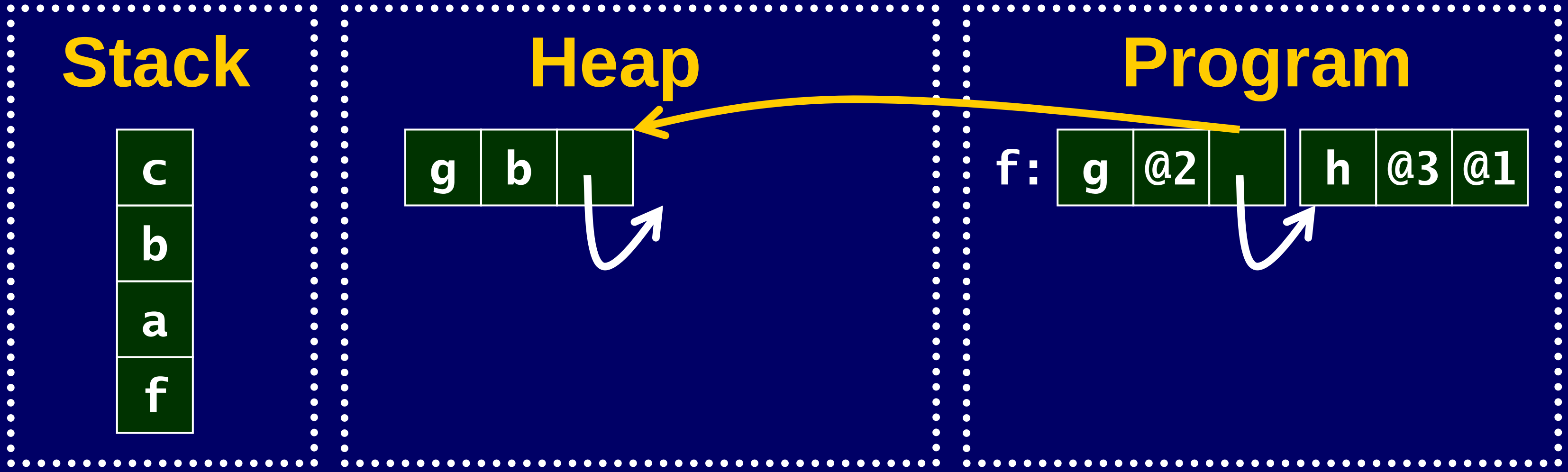
h @3 @1

Graph Reduction

Operation:

```
ptr  <- Code[f+2]
ptr’ -> Heap
```

Count: 8

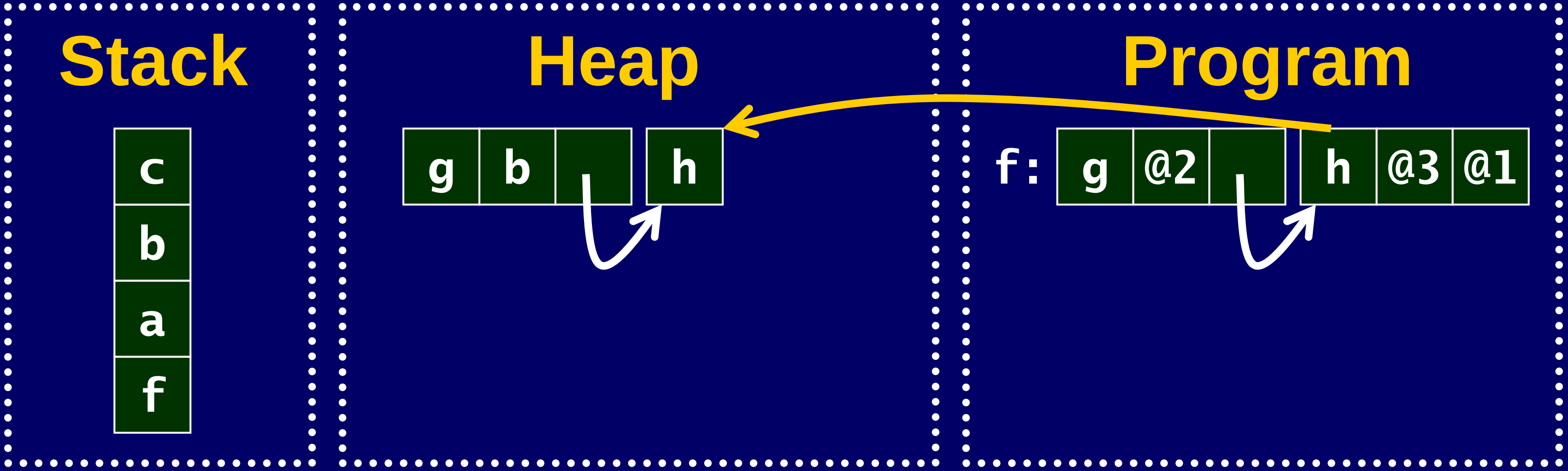


Graph Reduction

Operation:

```
h <- Code[f+3]
h -> Heap
```

Count: 10

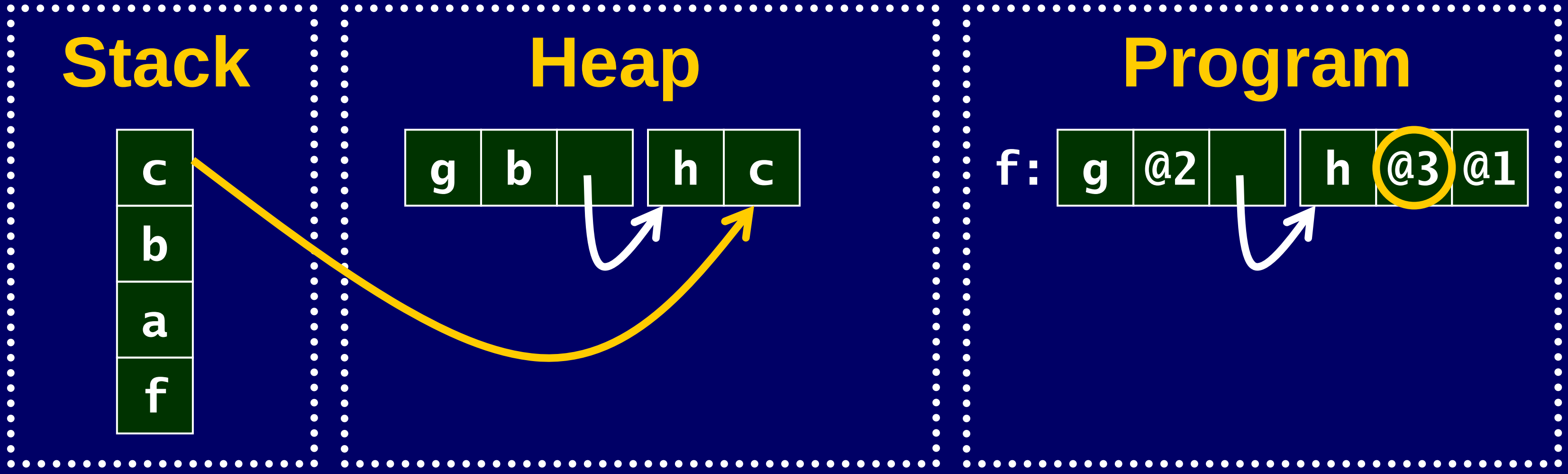


Graph Reduction

Operation:

```
arg <- Code[f+4]
c   <- Stack[arg]
c   -> Heap
```

Count: 13

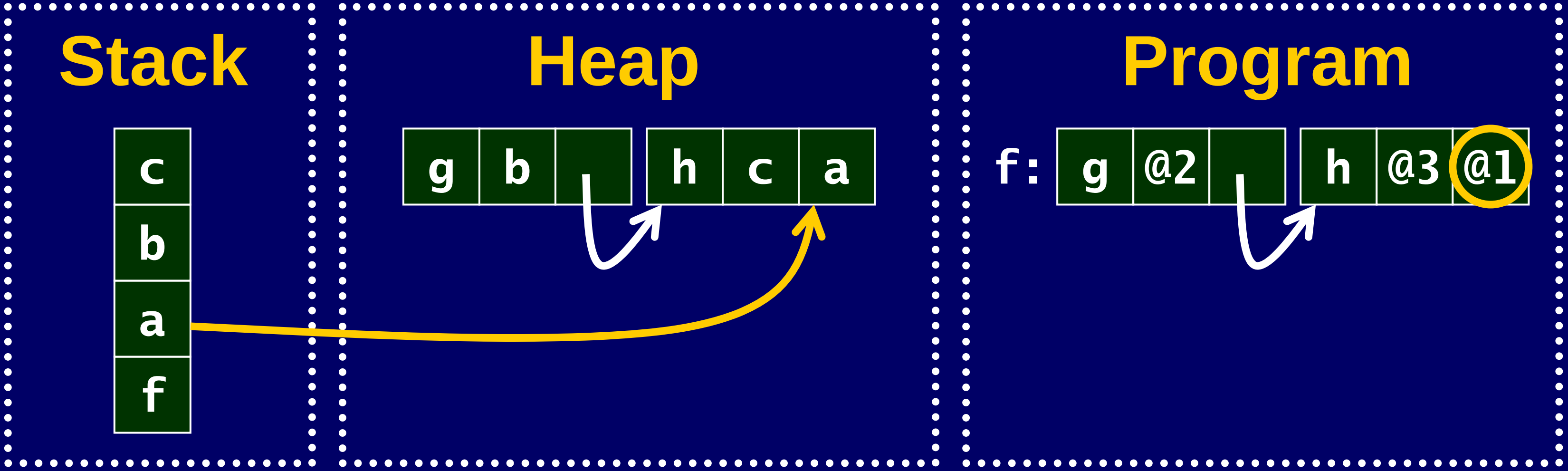


Graph Reduction

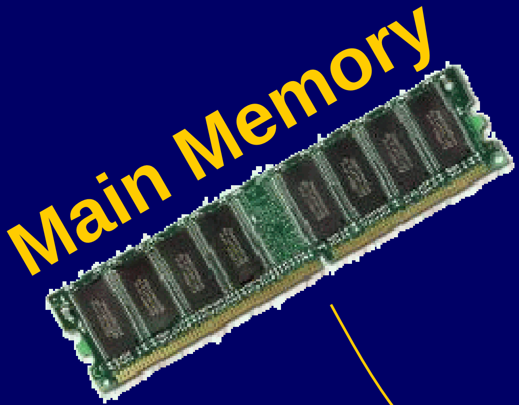
Operation:

```
arg <- Code[f+5]
a   <- Stack[arg]
a   -> Heap
```

Count: 16



The von Neumann Bottleneck

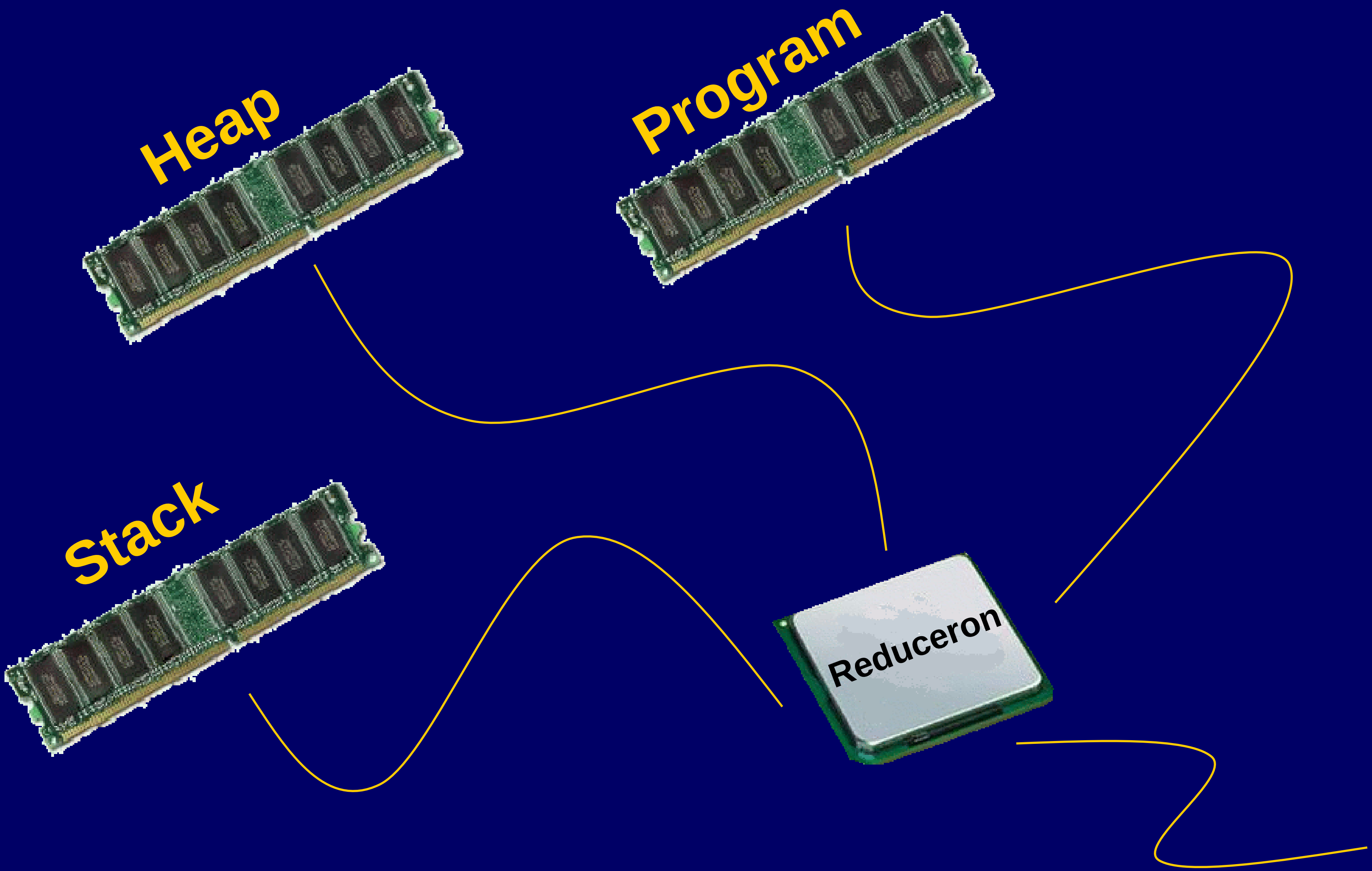


Main Memory

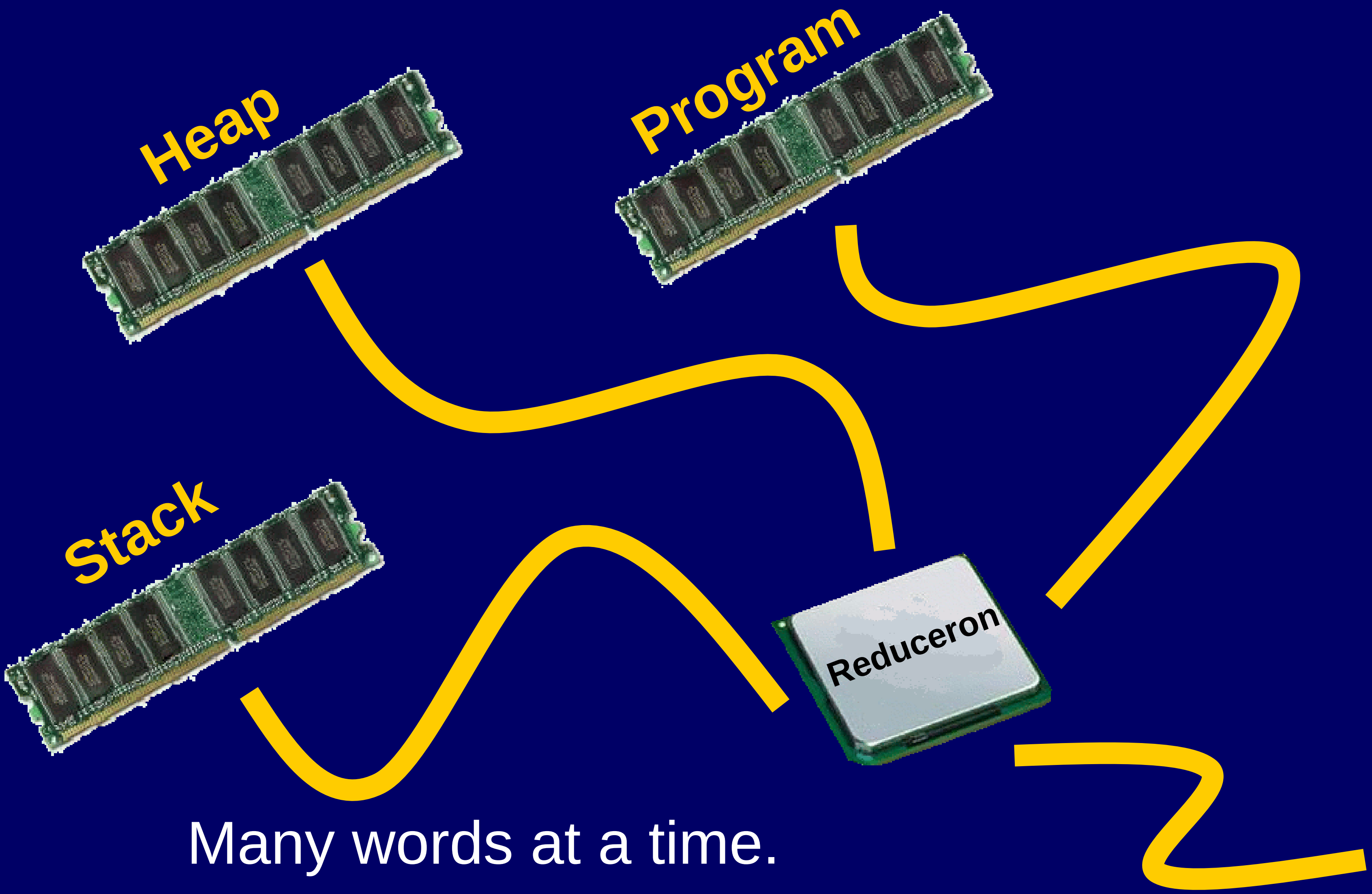
One word at a time.

Each of the **16** memory transactions is done sequentially.

Widening the Bottleneck

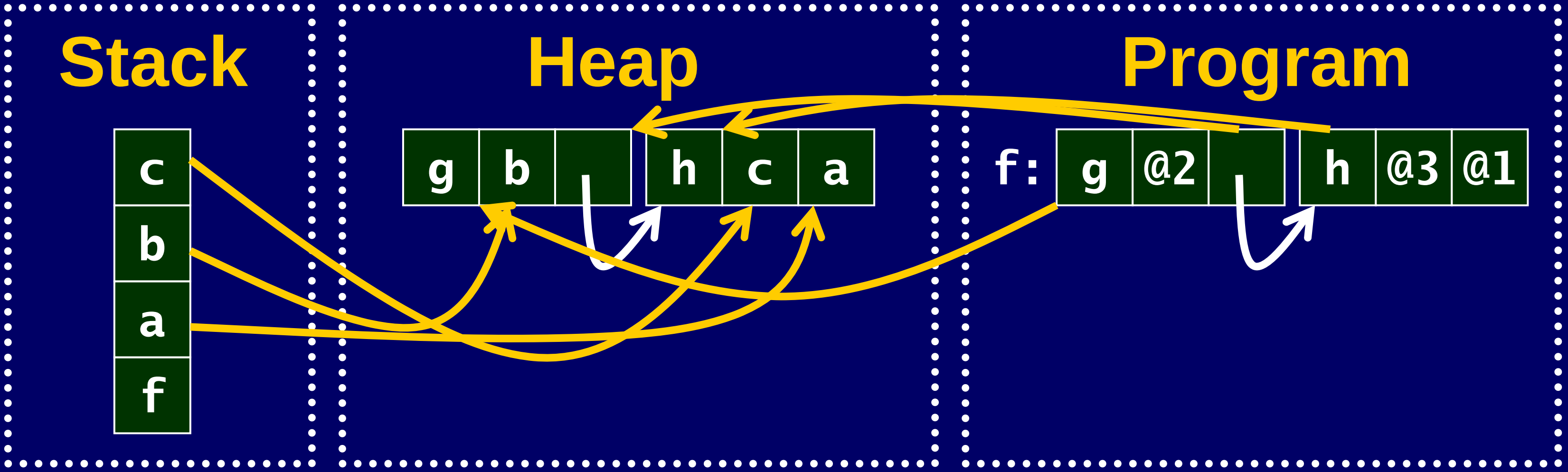


Widening the Bottleneck, again



Many words at a time.

Applying a function "in one go"



Reduceron, September 2008

Operation	Clock cycles
Apply	$3 + \lfloor \mathbf{n}/8 \rfloor$
Unwind	2
Swap	2
Primitive Apply	3

Includes updating

Where **n** = number of ***nodes*** in function body.

Reduceron, September 2008

Wide Reduceron
(uses wide, parallel memories)

5x faster than

Narrow Reduceron
(single connection to memory)

Wide Reduceron
at 92MHz on Virtex-II FPGA

5x slower than

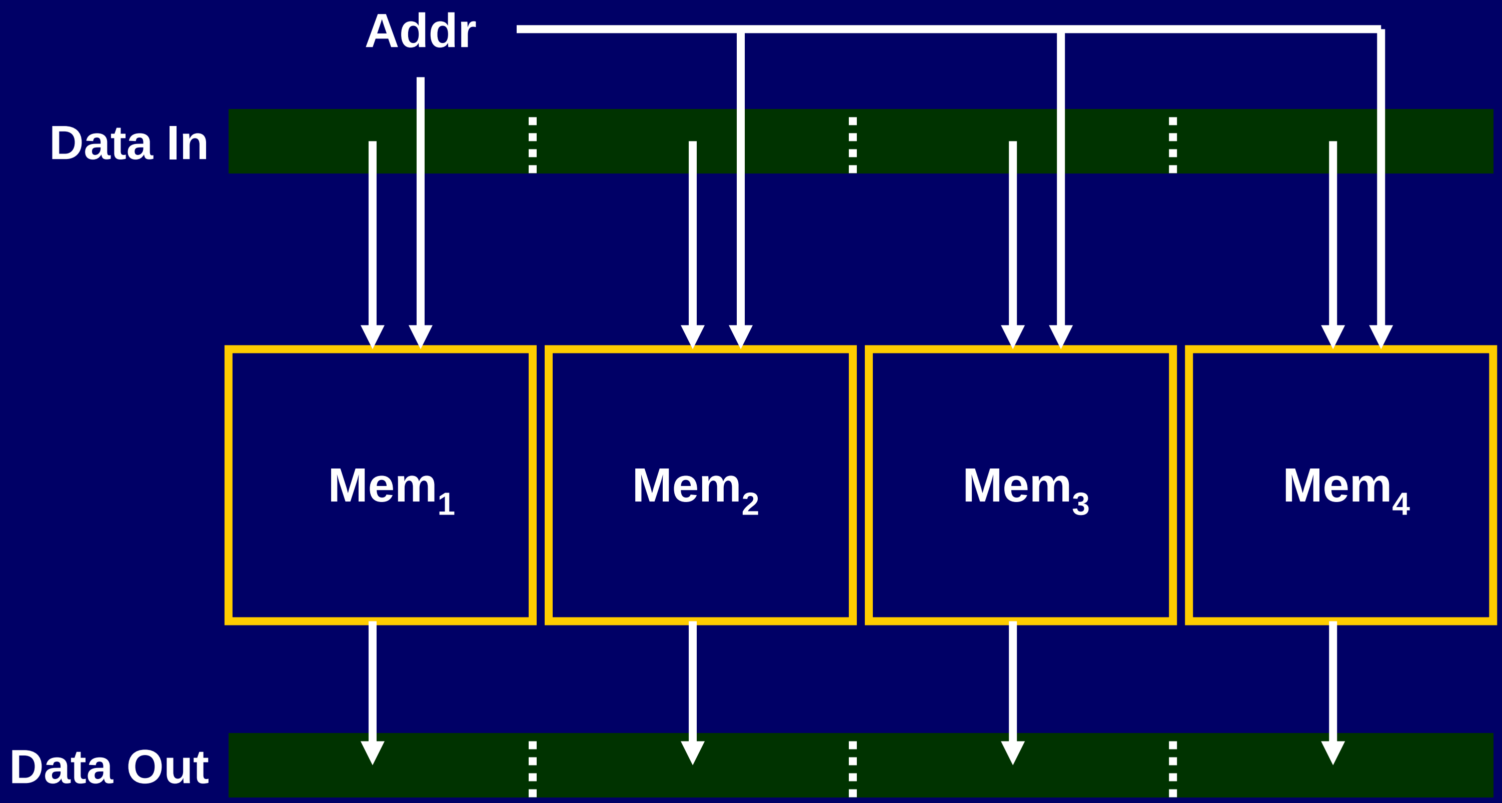
GHC –O2
(advanced optimising compiler)
at 2800MHz on Pentium-4 PC

(On “symbolic programs”.)

Improvement 1

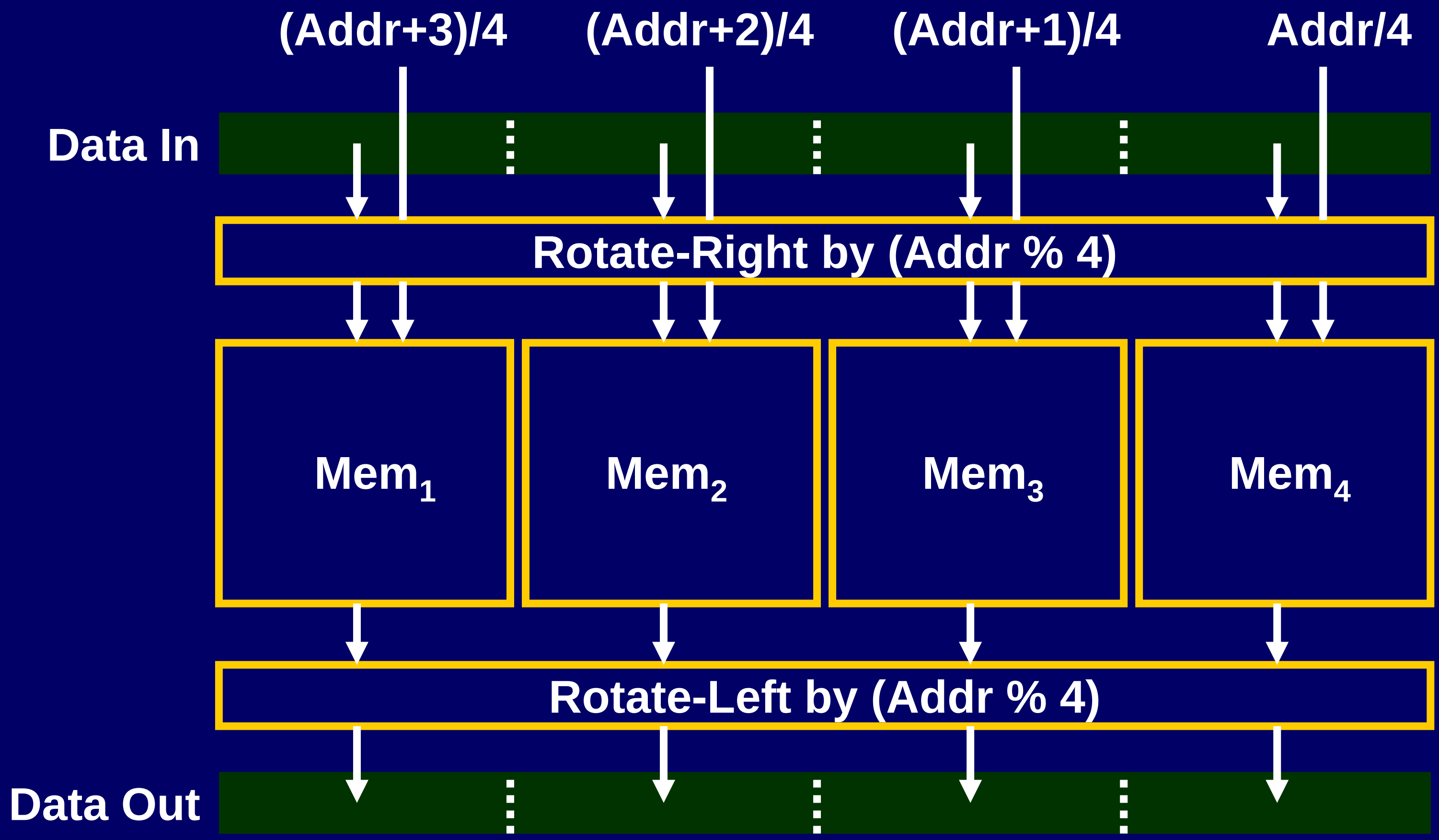
Heap and stack layout

Making a wide memory: Method 1



Cannot address individual words, ***only blocks of 4***.

Making a wide memory: Method 2



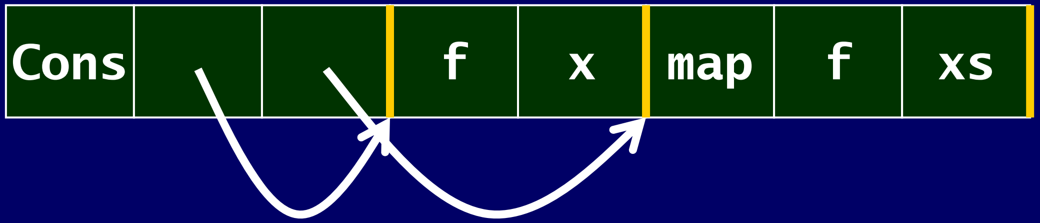
Can address ***any 4*** consecutive words,
but extra logic is needed which may need buffered.

Old Heap Layout

Used Method 2, so the expression

`Cons (f x) (map f xs)`

was represented in memory as



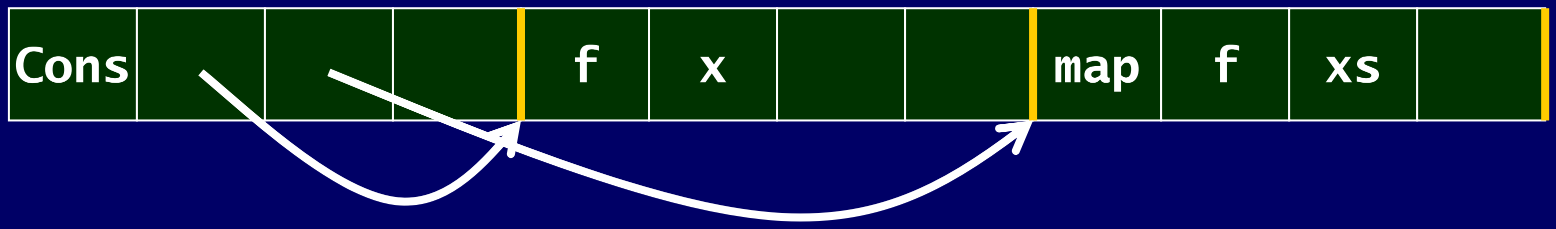
Great utilisation, but buffer on memory bus resulted in ***2-cycle reads***.

New Heap Layout

Uses Method 1, so the expression

```
Cons (f x) (map f xs)
```

is represented in memory as



`Cons`				`f`	`x`			`map`	`f`	`xs`	

Poor utilisation, but allows ***1-cycle reads***.

Also permits updating without indirections.

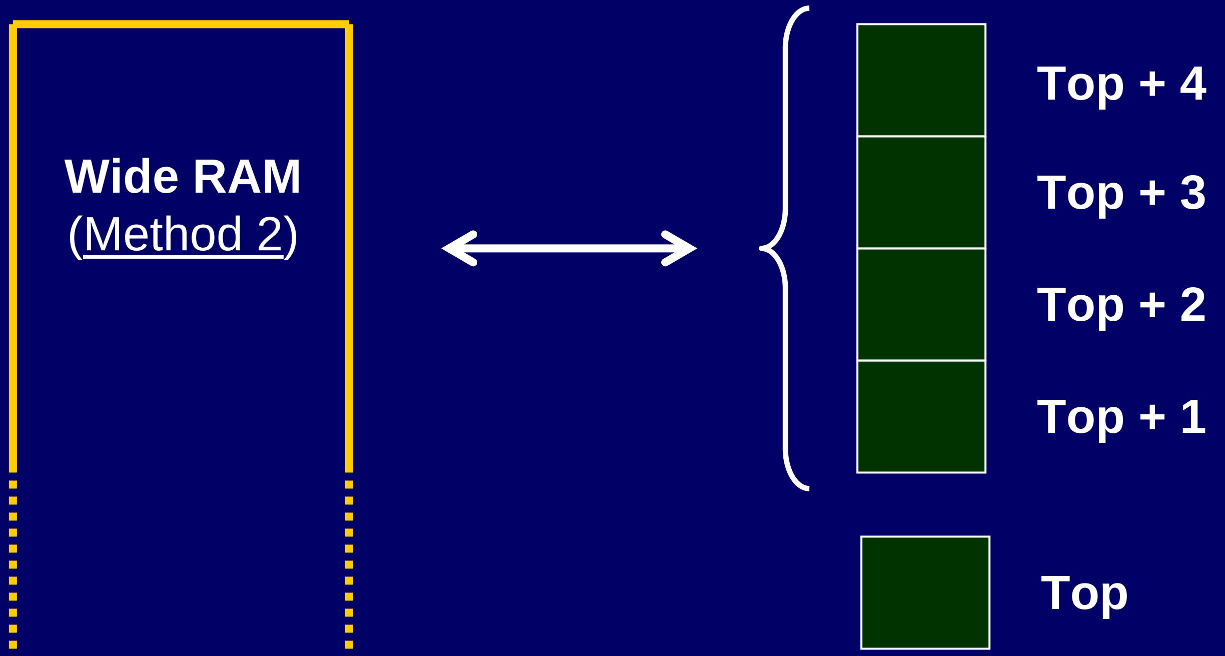
Old Stack Layout

Stack cannot have gaps, so ***must*** use Method 2!

Now 2 cycles are needed to read from stack…

New Stack Layout

But top elements can be stored in registers.



Top elements can be read in 0 cycles.

(This is the critical path in my current design – suggestions welcome!)

New clock counts

Using new heap/stack layout (& spineless reduction).

Operation	**Clock cycles**
Apply	$\lceil \mathbf{n}/2 \rceil$
Unwind	1
Update	1
Swap	1
Primitive Apply	1

Where **n** = number of ***applications*** in function body.

Clock frequency not affected.

Improvement 2

Dealing with case expressions

Case expressions

In general

$$\begin{array}{l}
\texttt{case e of} \\
\quad C_1\ v_1 \ldots v_{\#c_1}\ \texttt{->}\ e_1 \\
\qquad\qquad\qquad \vdots \\
\quad C_n\ v_1 \ldots v_{\#c_n}\ \texttt{->}\ e_n
\end{array}$$

Example

$$\begin{array}{l}
\texttt{app xs ys =} \\
\quad \texttt{case xs of} \\
\qquad \texttt{Cons}\ v_1\ v_2\ \texttt{->} \\
\qquad\quad \texttt{Cons}\ v_1\ \texttt{(app}\ v_2\ \texttt{ys)} \\
\qquad \texttt{Nil -> ys}
\end{array}$$

Case expressions

In general

$$\begin{array}{l}
\texttt{case e of} \\
\quad \texttt{C}_1\ \texttt{v}_1 \ldots \texttt{v}_{\#c_1}\ \texttt{->}\ \texttt{e}_1 \\
\qquad\qquad\qquad \vdots \\
\quad \texttt{C}_n\ \texttt{v}_1 \ldots \texttt{v}_{\#c_n}\ \texttt{->}\ \texttt{e}_n
\end{array}$$

Example

$$\begin{array}{l}
\texttt{app xs ys =} \\
\quad \texttt{case xs of} \\
\qquad \texttt{Cons}\ \texttt{v}_1\ \texttt{v}_2\ \texttt{->} \\
\qquad\quad \texttt{Cons}\ \texttt{v}_1\ \texttt{(app}\ \texttt{v}_2\ \texttt{ys)} \\
\qquad \texttt{Nil -> ys}
\end{array}$$

Free variable with respect to **case** expression.

Case elimination, part 1

(The Scott/Parigot/Jansen/… encoding.)

$$\begin{array}{l} \mathbf{case}\ e\ \mathbf{of} \\ \quad C_1\ v_1\ldots v_{\#c_1}\ \text{->}\ e_1 \\ \qquad\qquad\qquad \vdots \\ \quad C_n\ v_1\ldots v_{\#c_n}\ \text{->}\ e_n \end{array} \quad\Longrightarrow\quad \begin{array}{l} e\ (alt_1\ \nu(e_1)) \\ \quad \vdots \\ \quad (alt_n\ \nu(e_n)) \end{array}$$

where

$$\begin{array}{l} alt_1\ \nu(e_1)\ v_1\ldots v_{\#c_1} = e_1 \\ \qquad\qquad\qquad\qquad \vdots \\ alt_n\ \nu(e_n)\ v_1\ldots v_{\#c_n} = e_n \end{array}$$

$\nu(e)$ denotes the ***free variables*** in e.

Case elimination, part 1, example

```
app xs ys =
  case xs of
   Cons $v_0$ $v_1$ -> Cons $v_0$ (app $v_1$ ys)
   Nil -> ys
```

⇓

```
app xs ys = xs (alt$_1$ ys) ys
alt$_1$ ys $v_0$ $v_1$ = Cons $v_0$ (app $v_1$ ys)
```

Case elimination, part 2

(The Scott/Parigot/Jansen/… encoding.)

For each constructor C_i, introduce function

$$C_i\ \ v_1 \ldots v_{\#C_i}\ \ k_1 \ldots k_n\ =\ k_i\ \ v_1 \ldots v_{\#C_i}$$

For example, the list constructors:

```
Nil n c = n
Cons x xs n c = c x xs
```

Case elimination, bigger example

```
data Exp = X
         | Y
         | Neg Exp
         | Add Exp Exp
         | Sub Exp Exp

eval x y e =
  case e of
    X -> x
    Y -> y
    Neg n -> 0 - eval x y n
    Add n m -> eval x y n + eval x y m
    Sub n m -> eval x y n - eval x y m
```

Case elimination, bigger example

```
X x y neg add sub = x
Y x y neg add sub = y
Neg n m x neg add sub = neg n
Add n m x y neg add sub = add n m
Sub n m x y neg add sub = sub n m
eval x y e = e x y (negAlt x y)
                   (addAlt x y)
                   (subAlt x y)

negAlt x y n = 0 - eval x y n
addAlt x y n m = eval x y n + eval x y m
subAlt x y n m = eval x y n - eval x y m
```

Large arities.

Large bodies, with repetition.

Abstraction

$e\ x\ y\ (alt_3\ x\ y)\ (alt_4\ x\ y)\ (alt_5\ x\ y)$

$= \{ alt_1\ x\ y = x,\ alt_2\ x\ y = y \}$

$e\ (alt_1\ x\ y)\ (alt_2\ x\ y)$
$\quad (alt_3\ x\ y)\ (alt_4\ x\ y)\ (alt_5\ x\ y)$

= { abstraction }

No repetition

$e\ \boxed{alt_1\ alt_2\ alt_3\ alt_4\ alt_5}\ x\ y$

Row of constants

Case elimination, revisited

For each case alternative, introduce function

$$\mathtt{alt}_{\mathtt{i}}\ \mathtt{v}_1\ldots\mathtt{v}_{\#\mathtt{C}_{\mathtt{i}}}\ \mathcal{V}(\mathtt{e}_1\ldots\mathtt{e}_n)\ =\ \mathtt{e}_{\mathtt{i}}$$

Transform each **`case`** expression to

$$\mathtt{e}\ \mathtt{alt}_1\ \mathcal{V}(\mathtt{e}_1\ldots\mathtt{e}_n)$$ (Case alts are ***aligned*** – next slide)

Evaluate constructor $\mathtt{C}_{\mathtt{i}}$ to function at address

$$\mathtt{alt}_1\ +\ (\mathtt{i}-1)$$

↑

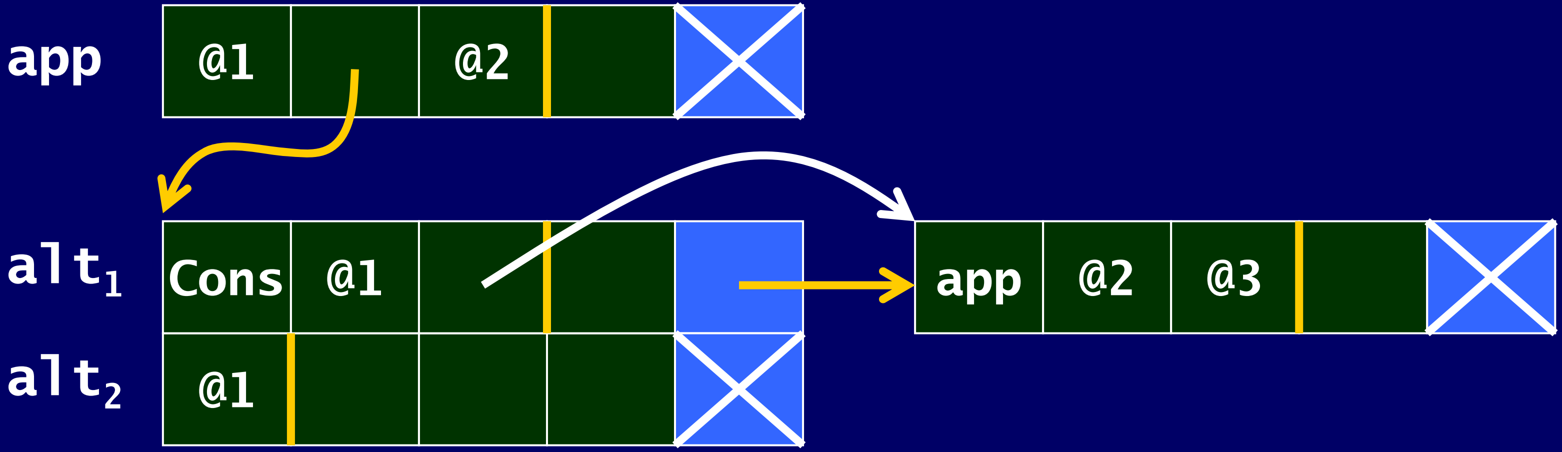
Simple addition - can be computed in 0 cycles?

Jumping in 0 cycles, part 1

The code for **app** (list append)

```
app xs ys     = xs alt₁ ys
alt₁ x xs ys = Cons x (app xs ys)
alt₂ ys       = ys
```

is represented in memory as follows.



Jumping in 0 cycles, part 2

Evaluate constructor $\mathbf{C}_i$ to function at address

$$\texttt{alt}_1 + (i-1)$$

↑

Not such a simple addition!

Must fetch $\texttt{alt}_1$ from stack, $\#\mathbf{C}_i$ places from the top

Solution: store $\texttt{alt}_1$ on a separate (parallel) stack.

Clock frequency not affected.

Improvement 3

Dynamic **update avoidance**

Shared applications

Distinguish between

- 🟩 ***unshared*** applications, and
- 🟥 ***possibly-shared*** applications.

Idea: When an unshared application is reduced to normal form, no update is needed.

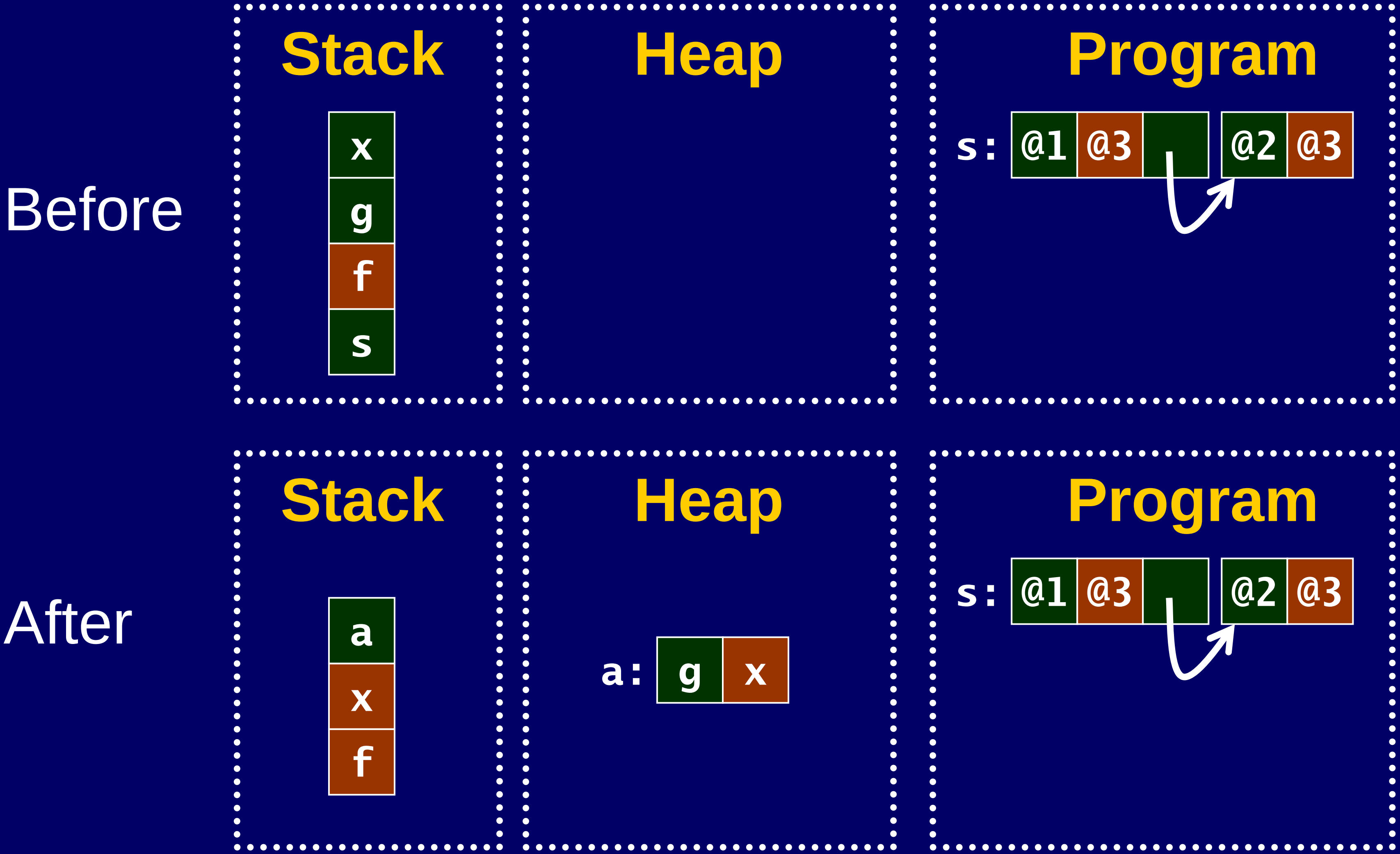
Dynamic v. Static Analysis

"Create all closures as [unshared], and dynamically change their tag to [possibly-shared] if they become shared. We call this operation ***dashing.****"*

"In general we strongly suspect that the cost of ***dashing*** *greatly outweighs the advantages of precision when compared to the [static analysis] method."*

--- Simon Peyton Jones

Dashing when applying



Before

Stack

x
g
f
s

Heap

Program

s: @1 @3 @2 @3

After

Stack

a
x
f

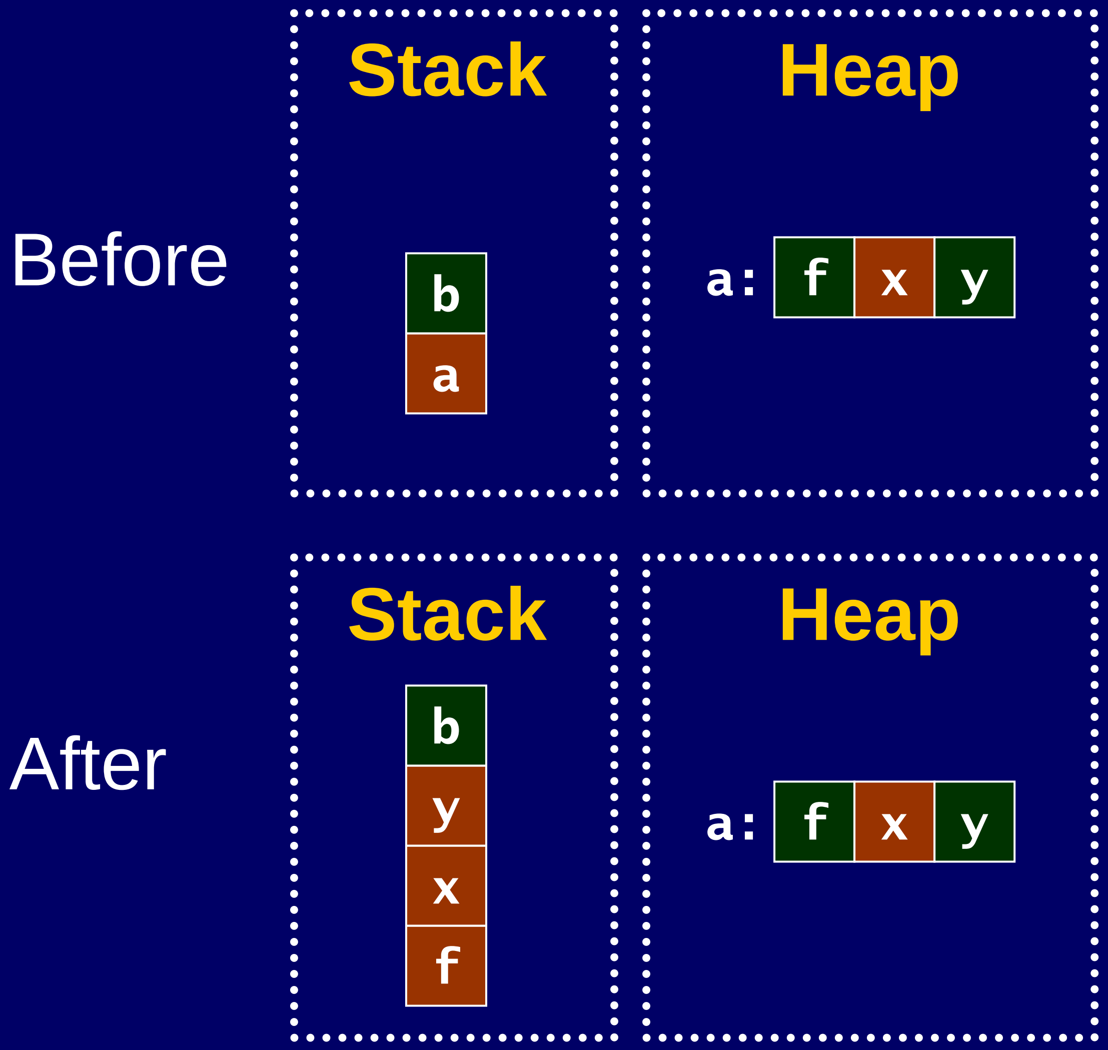
Heap

a: g x

Program

s: @1 @3 @2 @3

Dashing when unwinding



Before

Stack

b

a

Heap

a: f x y

After

Stack

b

y

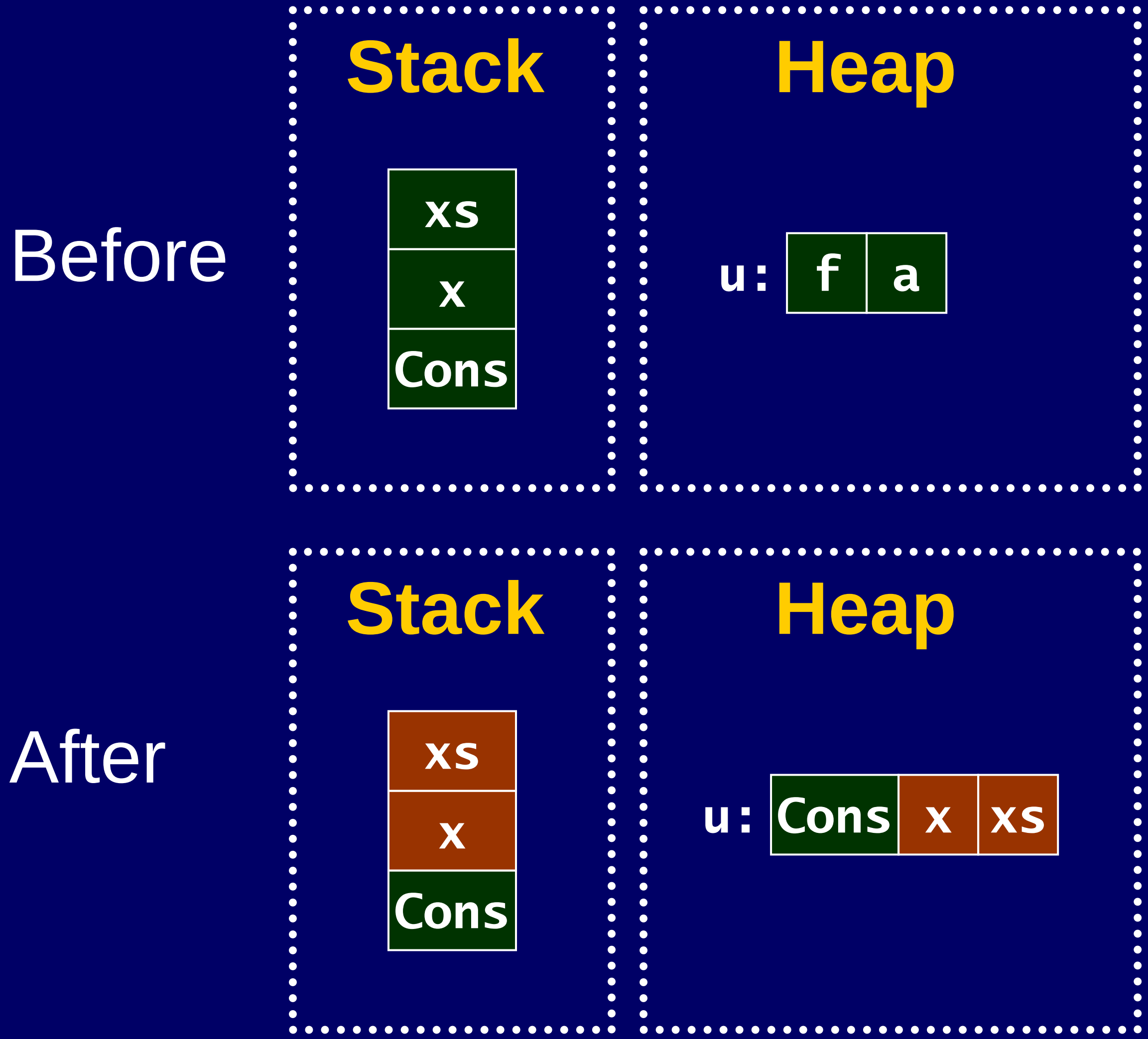
x

f

Heap

a: f x y

Dashing when updating



Before

Stack

xs
x
Cons

Heap

u: f a

After

Stack

xs
x
Cons

Heap

u: Cons x xs

Dynamic v. Static Analysis

In the Reduceron, dynamic update avoidance is rather cheap – it’s just bit-flipping under some simple-to-compute conditions.

Clock frequency not affected.

How the Reduceron is described

Description language

- York Lava
  - Multi-output primitives
  - RAMs
  - Modular - easy to add new primitives and back-ends
  - Behavioural description
  - Statically-sized bit-vectors

My dream…

Compile small-step machine semantics directly to efficient hardware.

For example, here’s the unwind rule from a structural operational semantics.

```
     <APP addr:s, u, h, c>
-->  <(h!addr) ++ s, (length s, addr):u, h, c>
```

But for now...

```
unwind top s u h c doUpdate =
 do top <== n
    vpush len ns s
    upush (mkUpdate (stackSize s)
                    (apAddr $ val top)) u
    doUpdate <== (arity n |>| len)
    cread (funAddr n) c
    hreadB (apAddr n) h
    tick
  where
    app = heapOutB h
    len = appArity app
    (n:ns) = appNodes app
```

Argh –pre-fetching and pre-computing!

Preliminary results and our to-do list

Performance improvement

Program	**Speed-up**
Queens	2.1
Queens$_2$	2.9
PermSort	2.9
MSS	2.7
PropInsert	3.0
Sudoku	4.0
Adjoxo	3.1
While	2.8
Clausify	3.6
Average	**3.0**

To-do list

- Critical path reduction
- Parallel garbage collection (low or high-level?)
- Compile-time optimisation
  - Supercompilation (Neil Mitchell, Jason Reich)
- Speculative evaluation of primitive redexes
- Multi-core Reduceron
- Relax memory restrictions
  - large, off-chip heap
- Efficient hardware from small-step semantics?