The Reduceron

Matthew Naylor and Colin Runciman
University of York

Talk given at Birmingham University, April 2009

Observation 1

Efficient compilation of high-level functional programs on conventional computers is a big challenge.

Sophisticated techniques needed to exploit architectural features designed for low-level imperative execution.

"I wonder how popular Haskell needs to become for Intel to optimize their processors for my runtime, rather than the other way around."

Simon Marlow, 2009

Observation 2

The target (architecture) changes!

"In light of evidence that Haskell programs compiled by GHC exhibit large numbers of mispredicted branches on modern processors, we re-examine the tagless aspect of the STG-machine."

Marlow et al., 2007

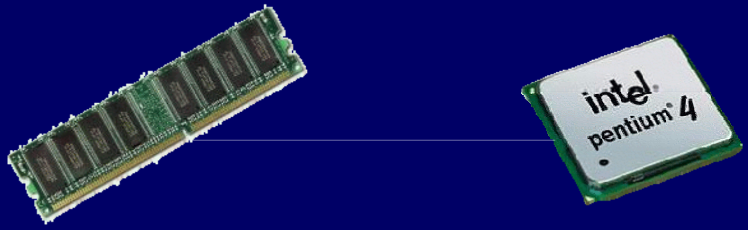
"We also discovered that one optimisation in the STG-machine, vectored-returns, is no longer worthwhile."

Marlow et al., 2007

Observation 3

Conventional computers have inherent limitations when it comes to running functional programs.

Memory Bottleneck



“Much of the implicit parallelism available in Haskell programs is at too fine a granularity for it to be exploitable on stock hardware.”

Harris, 2008

Question

Why not build a machine especially
to run functional programs?

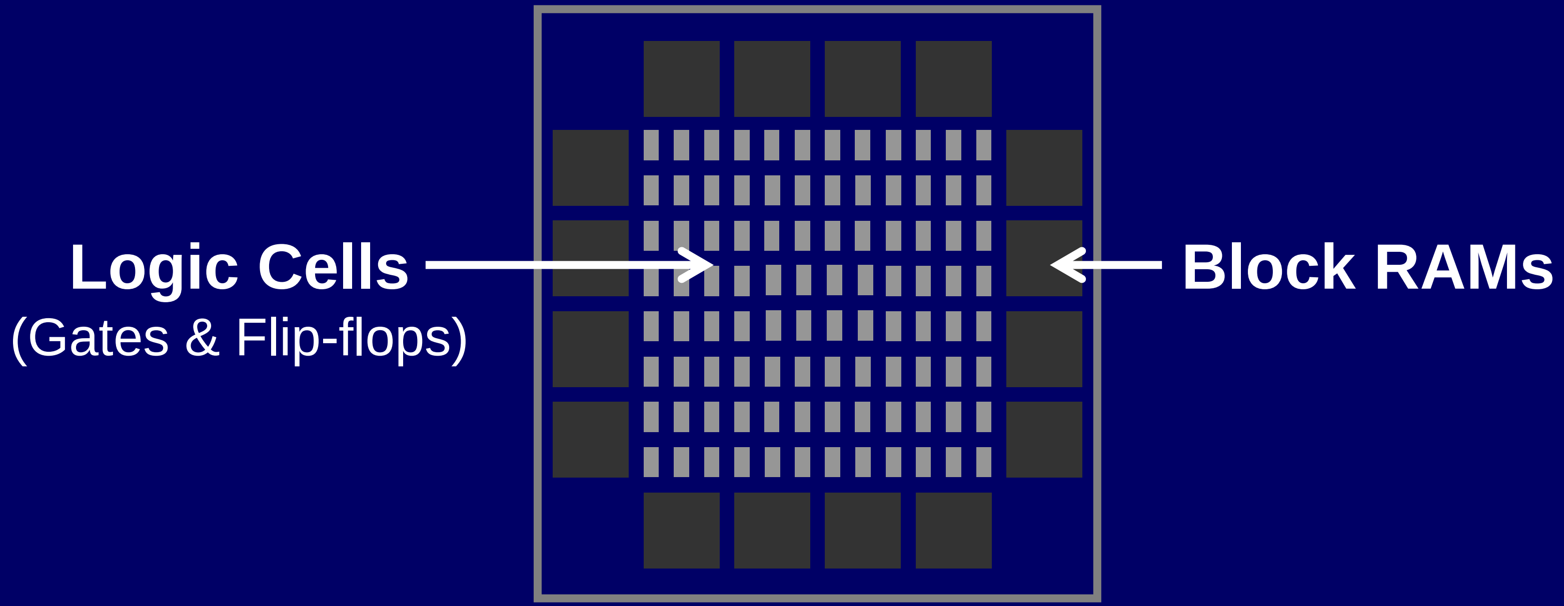
"Current RISC technology will probably have increased in speed enough by the time a graph-reduction chip could be designed and fabricated to make the exercise pointless."

Koopman, 1990

But now the situation is changing…

FPGAs (think Lego)

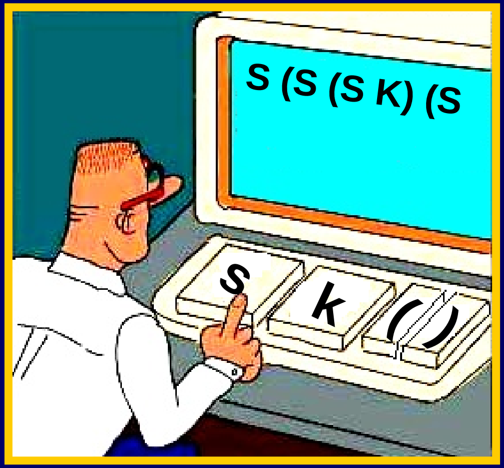
Contain a large, fixed set of components that can be connected together in any desired way.



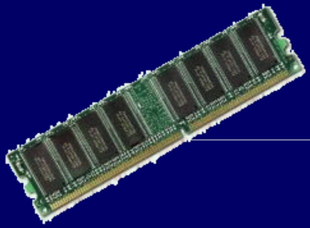
Widely available, and quick to program.

The Reduceron

A computer designed to run lazy functional programs,



not restricted by conventional architectural constraints,



implemented on an FPGA, using a functional language.

Project status

- Initially developed as *part* of my PhD
  - submitted September 2008.
- Now a 15-month EPSRC project
  - started October 2008;
  - one Professor, one RA, one under-graduate.
- Broad aim: fast reduction!
  - Explore what is possible, in a few directions.

Lazy evaluation

Quick recap

Reduction strategies

Suppose that **`double`** is defined by

```
double x   =   x + x
```

and we wish to reduce

```
double (1 + 1)
```

then there are **two** main strategies.

1. Innermost

Suppose that **`double`** is defined by

```
double x   =   x + x
```

then, using the **innermost strategy**,

```
      double (1 + 1)
=>    double 2
=>    2 + 2
=>    4
```

2. Outermost

Suppose that **`double`** is defined by

```
double x   =   x + x
```

then, using the **outermost strategy**,

```
      double (1 + 1)
=>    (1 + 1) + (1 + 1)
=>    2 + (1 + 1)
=>    2 + 2
=>    4
```

2. Outermost

Suppose that **`double`** is defined by

```
double x    =    x + x
```

then, using the **outermost strategy**,

```
      double (1 + 1)
=>    (1 + 1) + (1 + 1)
=>    2 + (1 + 1)
=>    2 + 2
=>    4
```

Reduced twice!

Evaluation strategies

How many times is a function argument evaluated?

Innermost:	**Exactly once**
Outermost:	**Zero or more**

Evaluation strategies

How many times is a function argument evaluated?

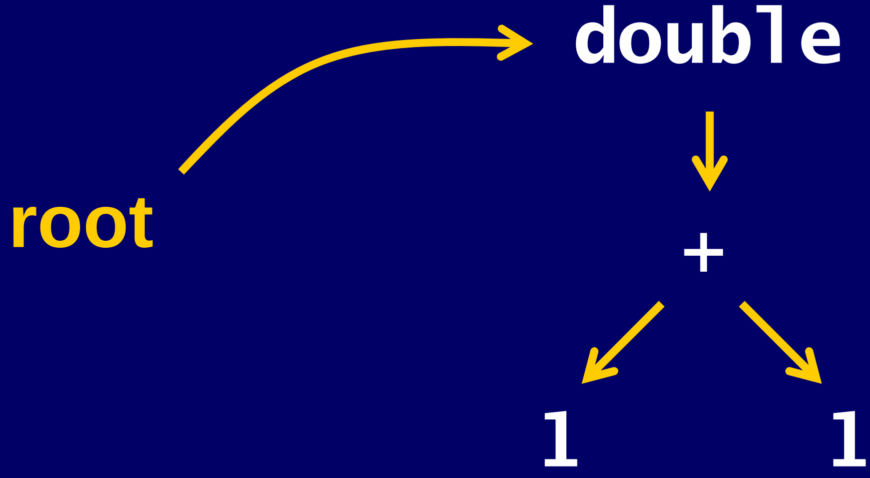
Innermost:	**Exactly once**
Outermost:	**Zero or more**
Lazy:	**Zero or once**

Expressions as graphs

Suppose that **`double`** is defined by

```
double x    =    x + x
```

and we wish to reduce **`double (1 + 1)`**.

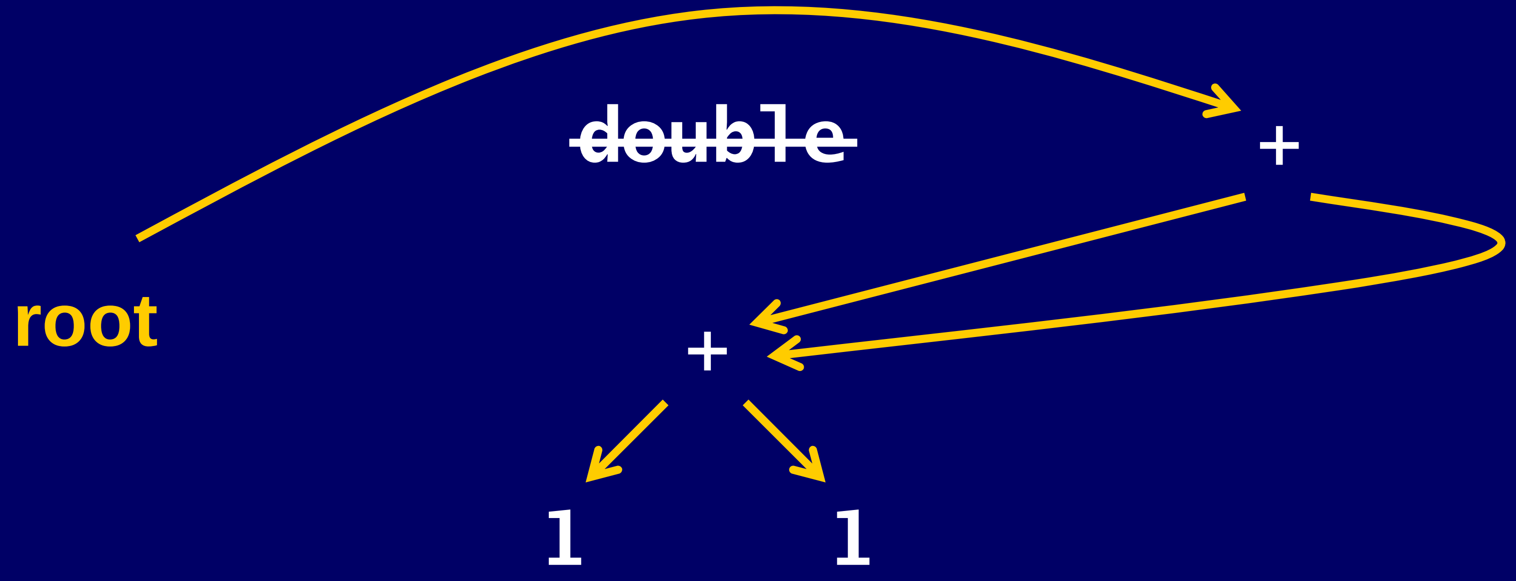


Graph reduction

Suppose that **`double`** is defined by

```
double x   =   x + x
```

then, by **lazy evaluation**,

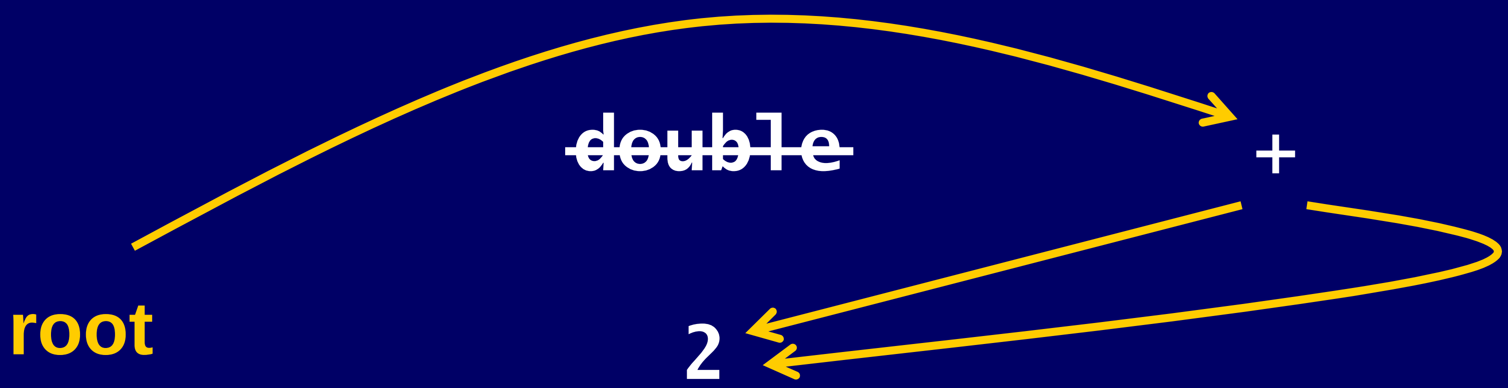


Graph reduction

Suppose that **`double`** is defined by

```
double x   =   x + x
```

then, by **lazy evaluation**,



Graph reduction

Suppose that **`double`** is defined by

```
double x    =    x + x
```

then, by **lazy evaluation**,

~~`double`~~ → **4**

root

Graph reduction machines

Widening the von Neumann Bottleneck

Graph reduction machines

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

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

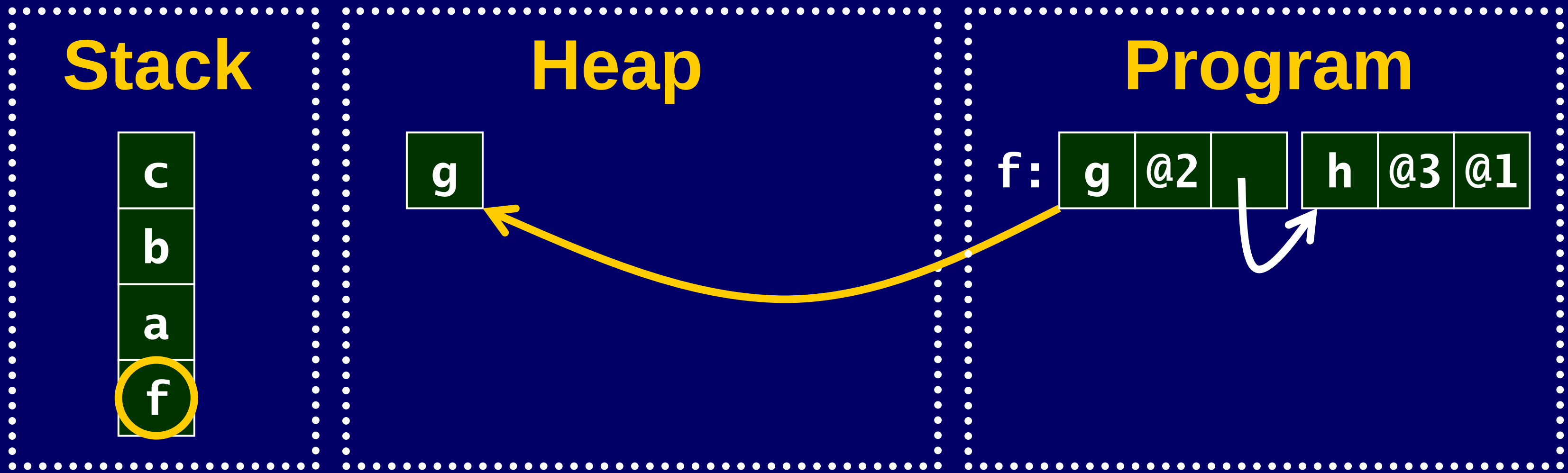
where **g** and **h** are functions and the following machine-state arises during reduction.

Graph reduction machines

Operation:

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

Count: 3



Graph reduction machines

Operation:

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

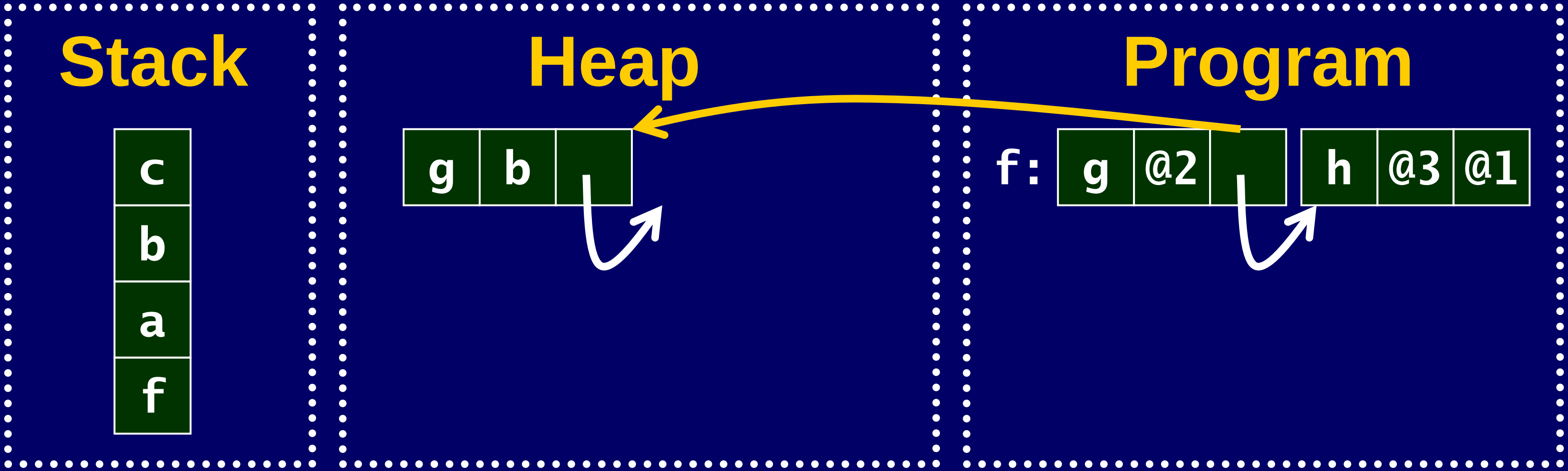
Count: 6

Graph reduction machines

Operation:

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

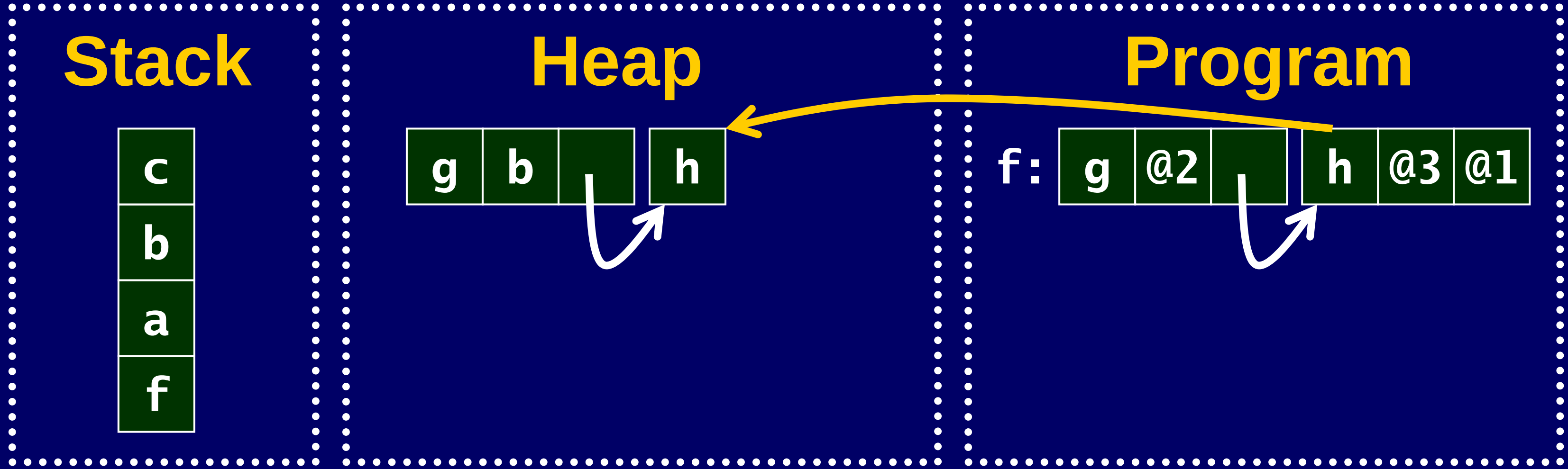
Count: 8



Graph reduction machines

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

Count: 10

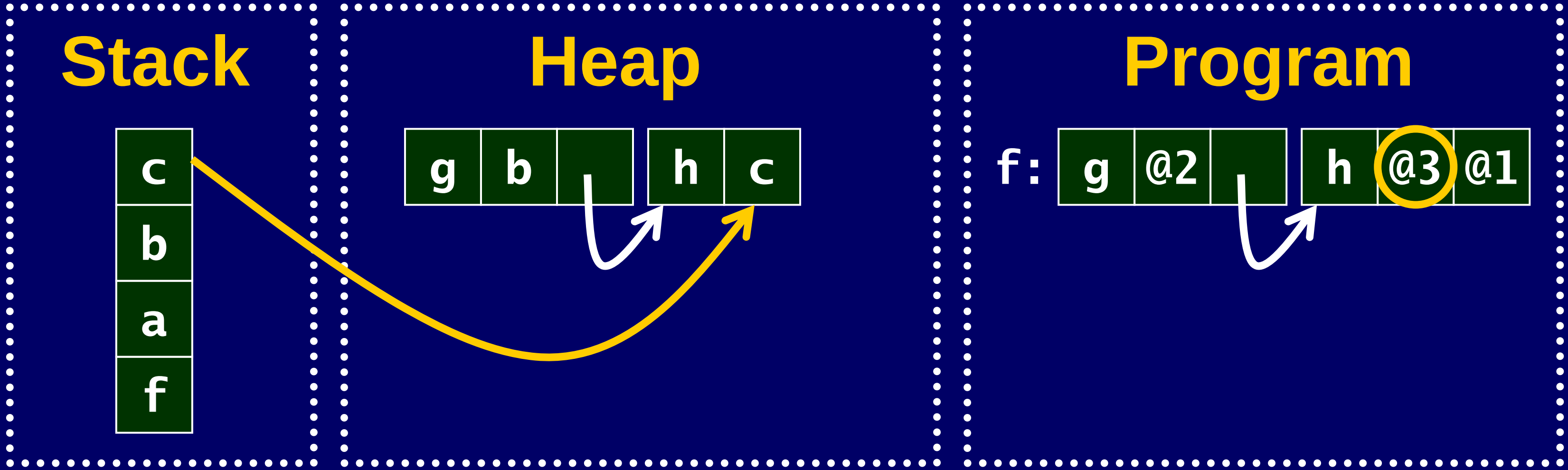


Graph reduction machines

Operation:

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

Count: 13

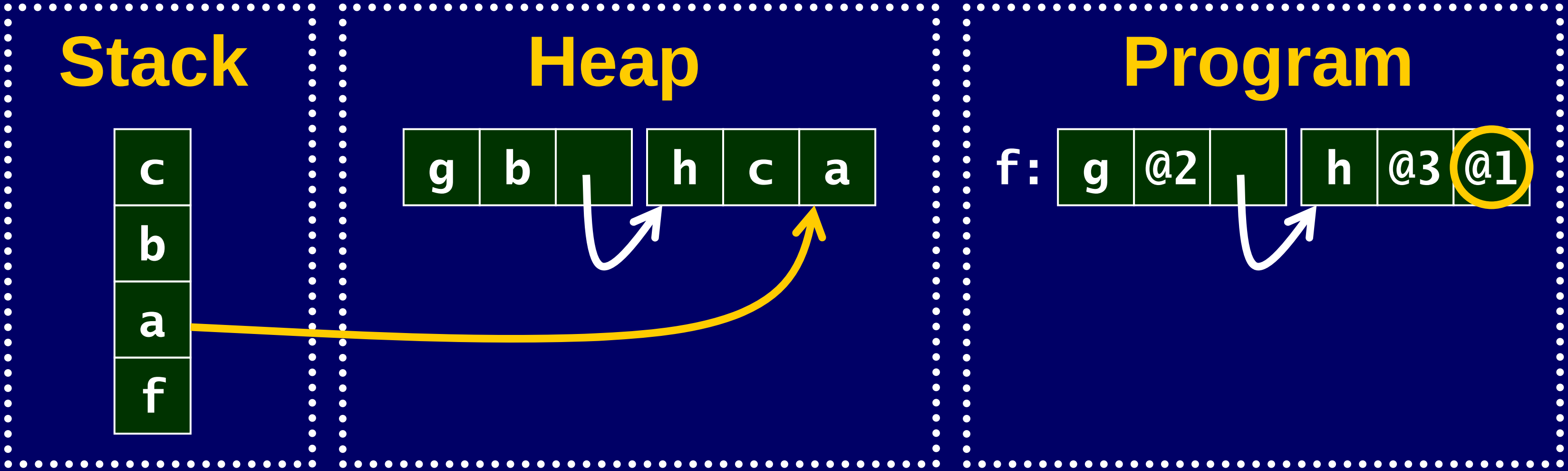


Graph reduction machines

Operation:

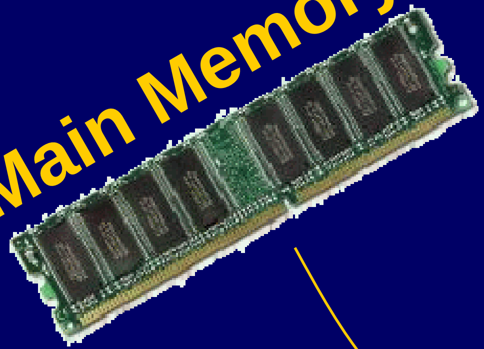
```
arg <- Code[f+5]
a   <- Stack[arg]
a   -> Heap
```

Count: 16



The von Neumann Bottleneck

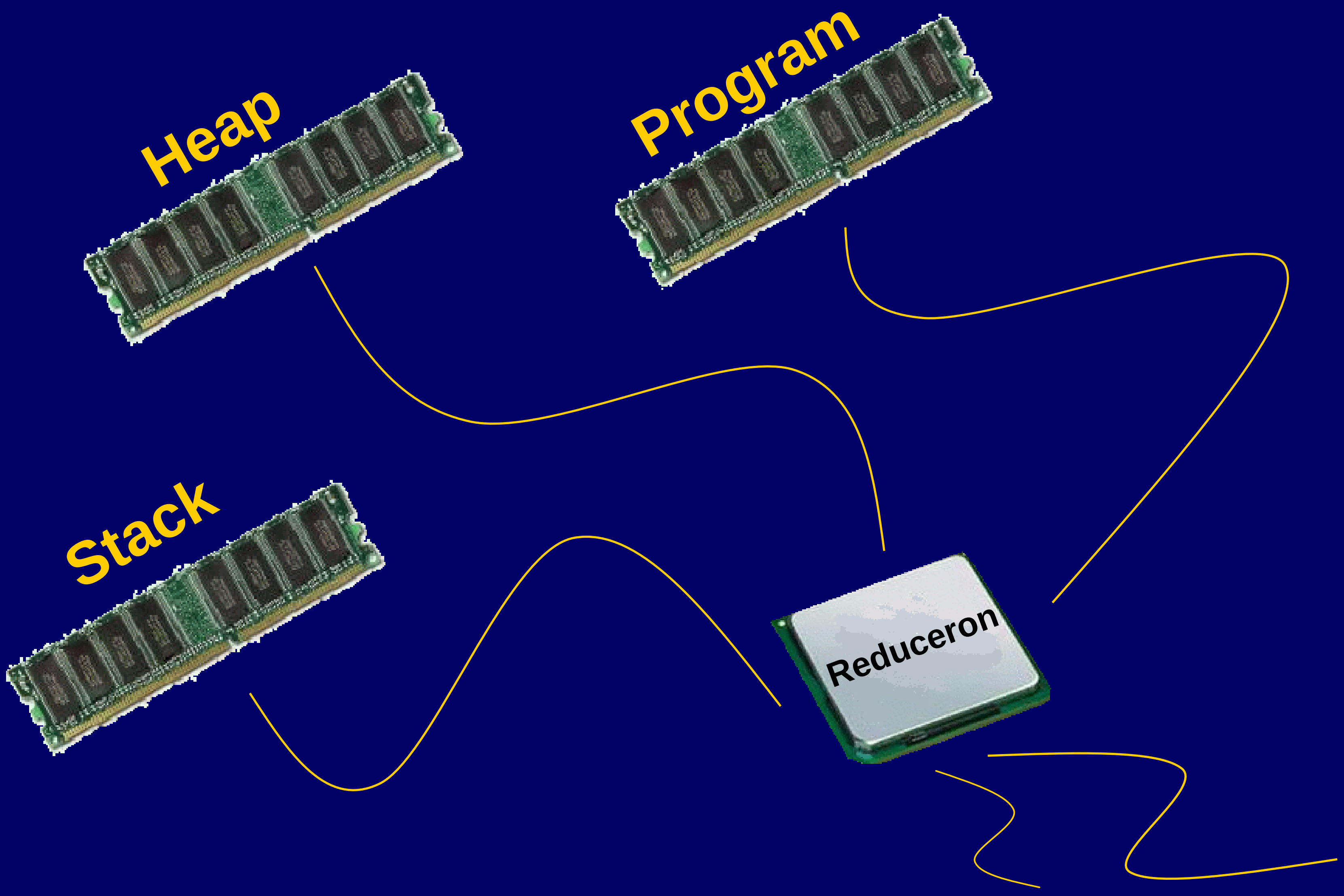
Main Memory



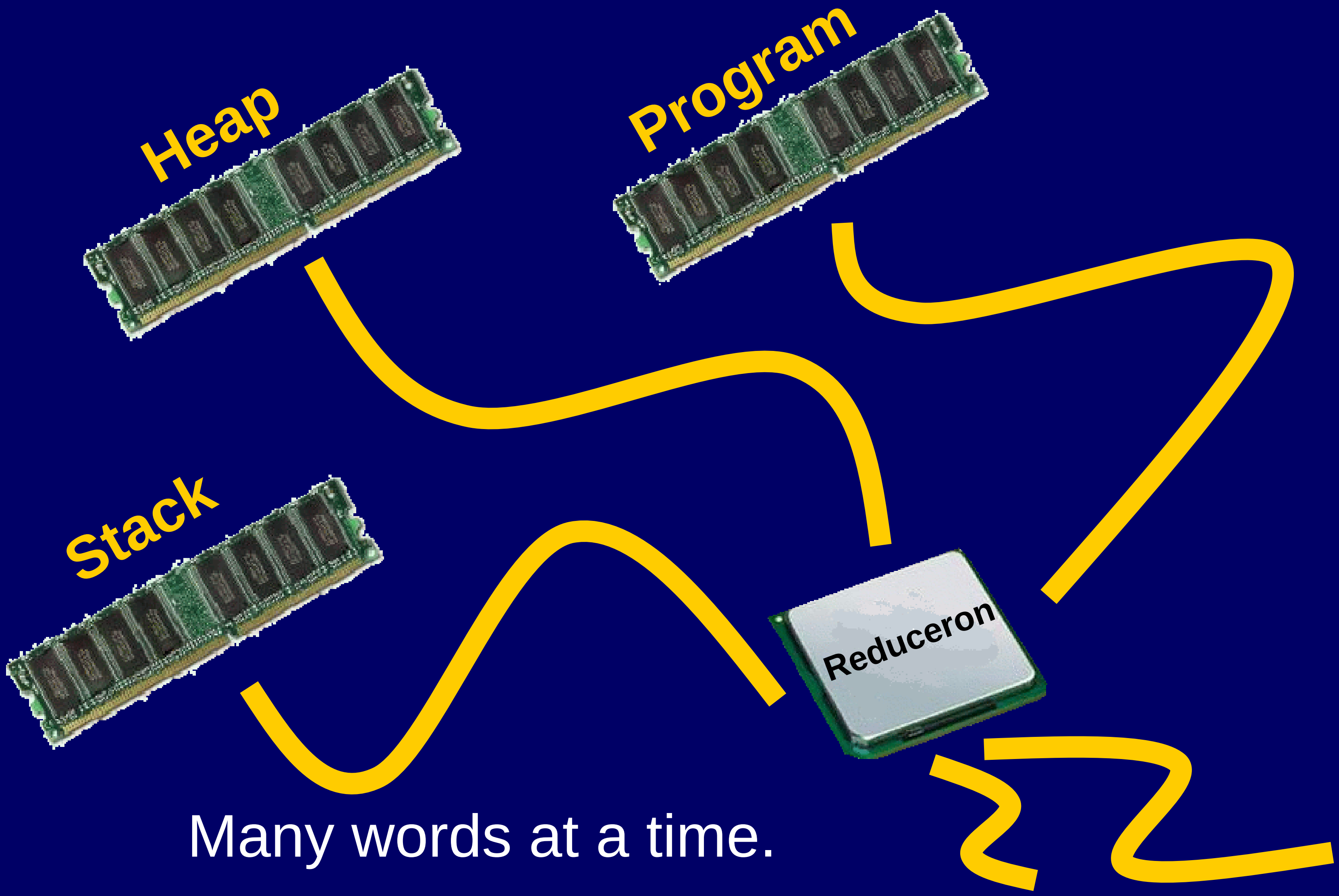
One word at a time.

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

Widening the Bottleneck



Widening the Bottleneck, again



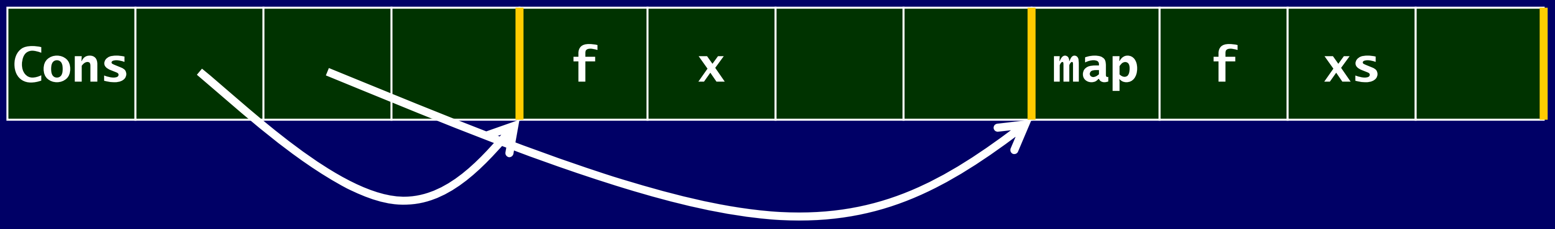
Many words at a time.

Heap layout

The expression

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

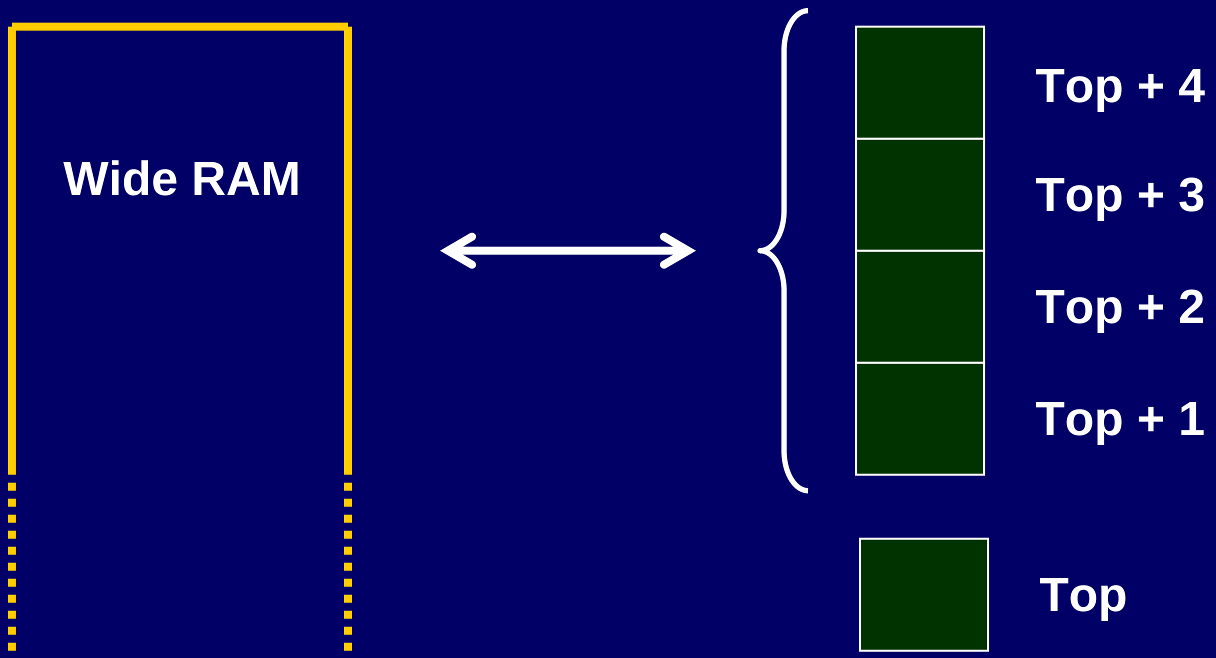
is represented in memory as



Two 4-word applications can be read or written per clock cycle.

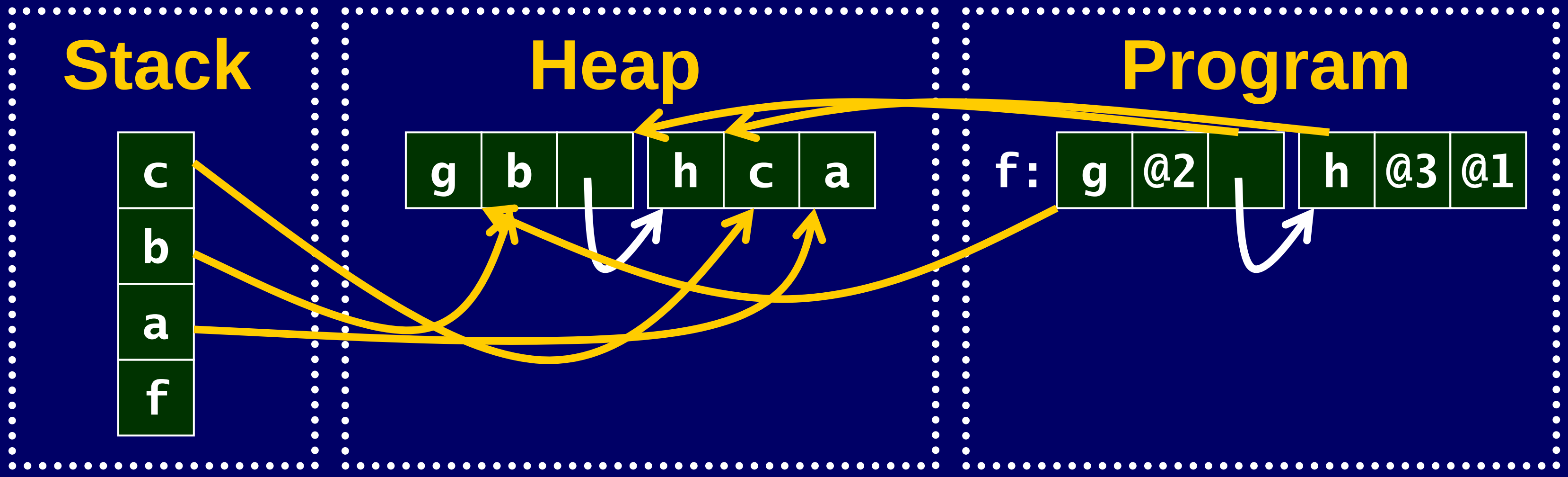
Stack layout

Top elements can be stored in registers.



Up to 5 words can be pushed & popped per cycle.

Applying a function “in one go”



The function `f` can be applied in a single clock cycle.

A note about memory

- For *simplicity* and *flexibility*, we make sole use of FPGA block RAMs, no off-chip memories.

- The total block RAM capacity of the latest Xilinx FPGAs is only *five megabytes*.

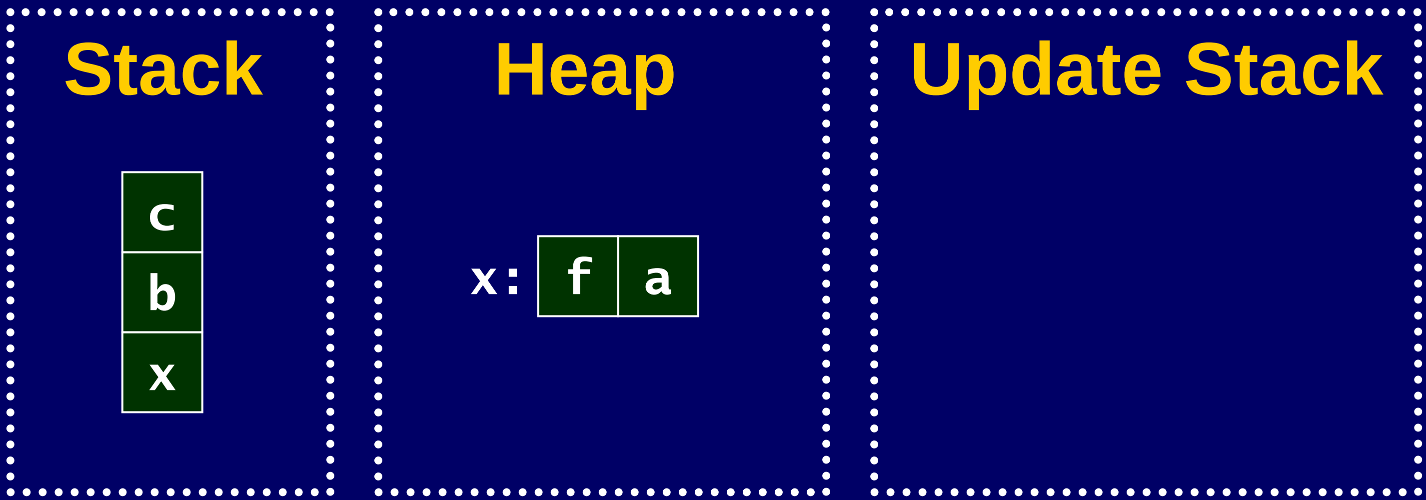
- Some consider this to be a *serious* limitation.
  - But FPGAs continue to get bigger and bigger…

Further widening (1)

The update stack

The update stack

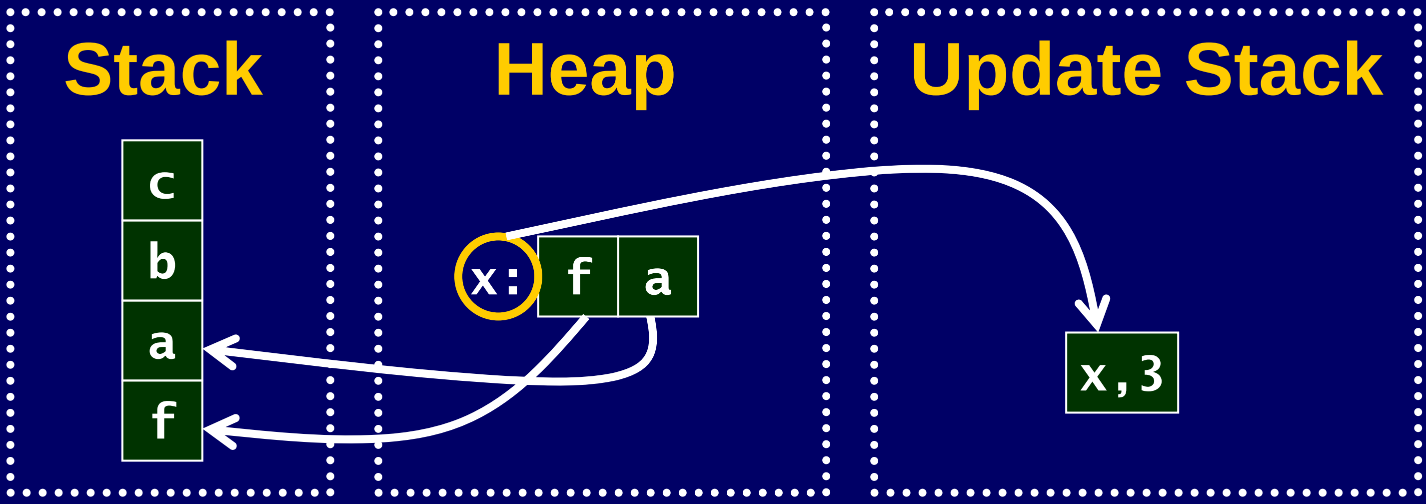
When a pointer to an application **x** on the heap appears on top of the stack,



the application is copied from heap to stack, ***and*** a pointer/stack-size pair is pushed onto the update stack.

The update stack

***Idea**: when evaluation of* **`f a`** *is complete, the value at* **`x`** *can be updated with the result.*



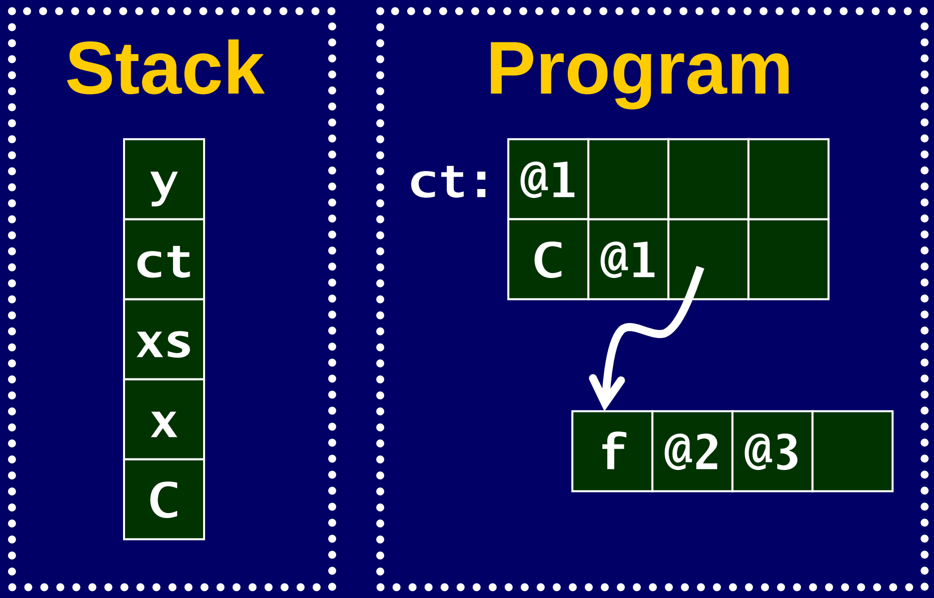
Writing to stack and update-stack is done in parallel.

Further widening (2)

The case stack

The case stack

When a data constructor is on top of the stack,



the corresponding case alternative is chosen from the ***case table***, pointed to by **`ct`**.

The case stack

Problem: pointer to case table must be fetched from the value stack, and it is not always in the same position! This look-up consumes a cycle.

Solution: store case table pointers on a separate stack. The case table pointer of interest is always at the top.

A separate case stack allows zero-cycle matching.

Reduceron “instruction set”

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.

Dynamic analysis (1)

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 vs. static analysis

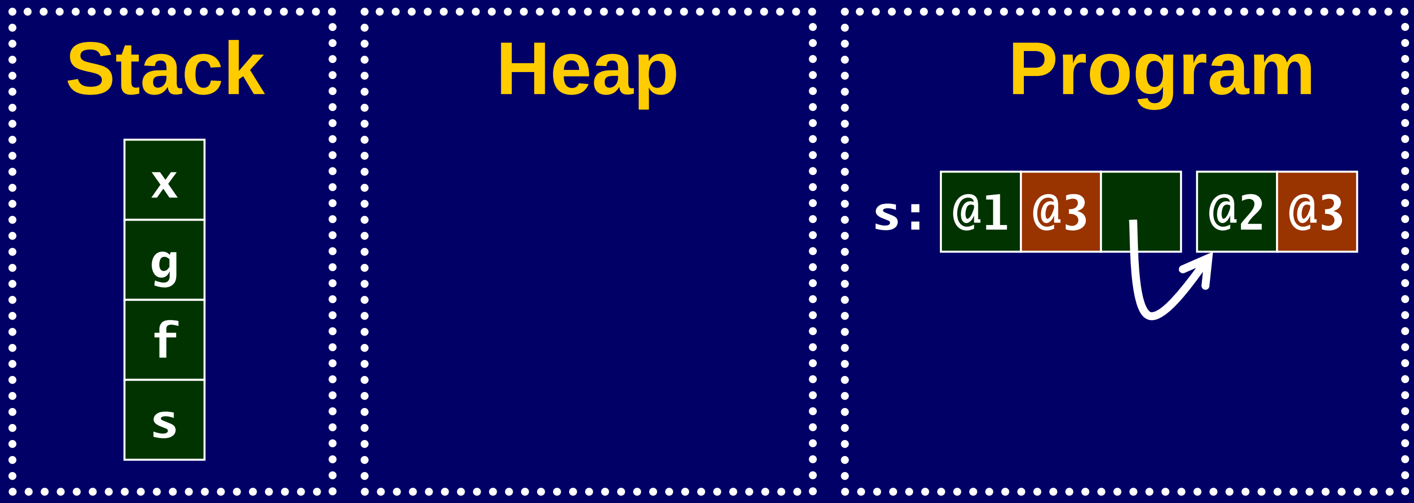
"Create all closures as [unshared], and dynamically change their tag to [possibly-shared] if they become shared. We call this operation ***dashing.****"*

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

Peyton Jones, 1988

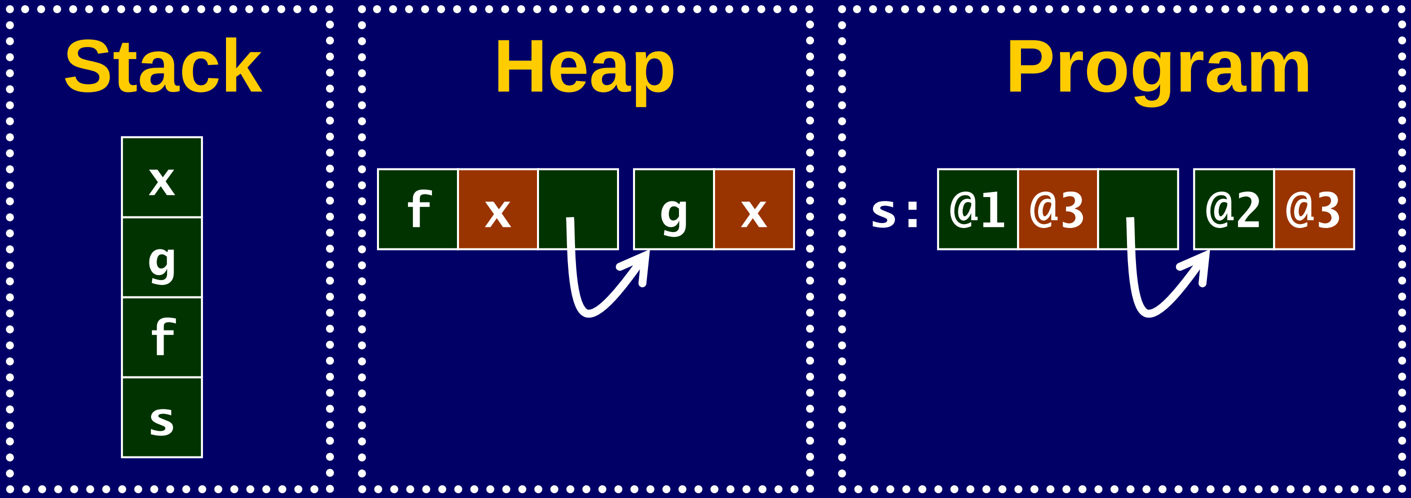
Dashing when applying

When the function ***s***, containing two references to its 3rd parameter, is applied,



Dashing when applying

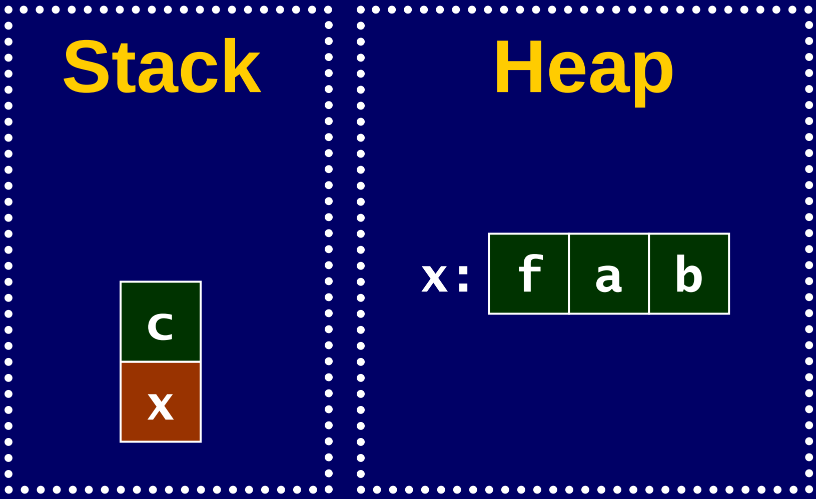
When the function ***s***, containing two references to its 3rd argument, is applied,



the 3rd argument is dashed.

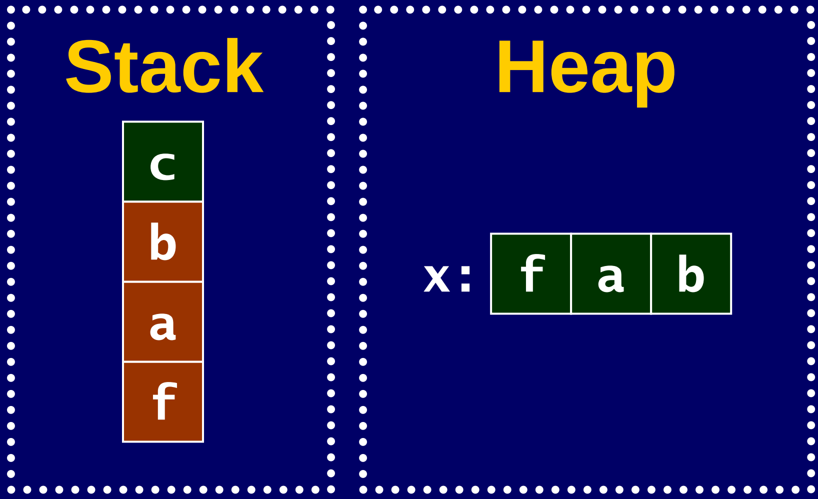
Dashing when unwinding

When a pointer **x** to a shared application appears on top of the stack,



Dashing when unwinding

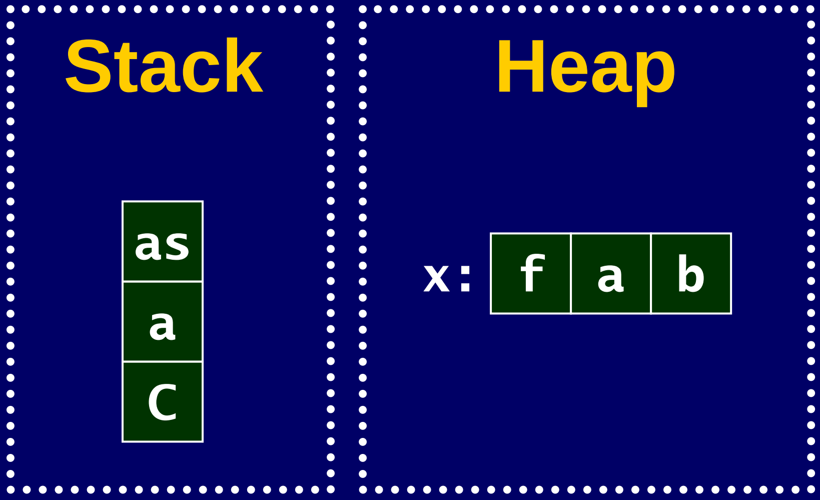
When a pointer **x** to a shared application appears on top of the stack,



the unwound application is dashed.

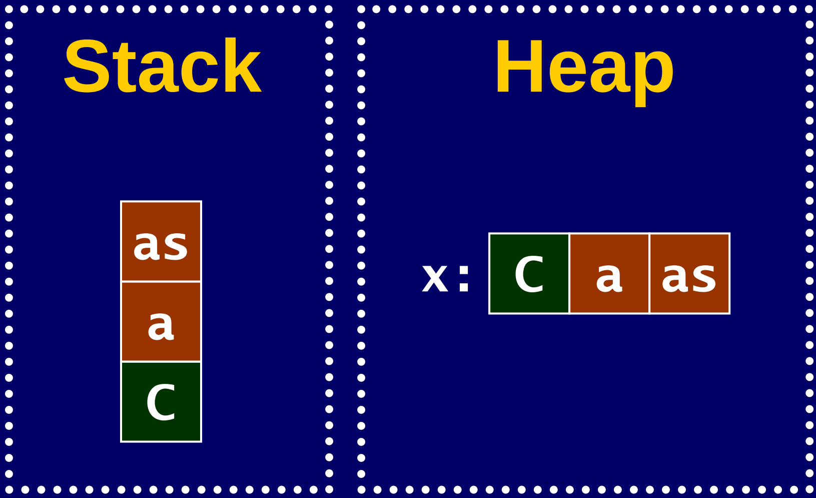
Dashing when updating

When a normal-form has been reached,



Dashing when updating

When a normal-form has been reached,



it is copied onto the heap, overwriting the original application, and its arguments are dashed.

Dynamic vs. static analysis

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

Dynamic analysis (2)

Speculative evaluation of primitive redexes

(Not implemented yet!)

Primitive redexes

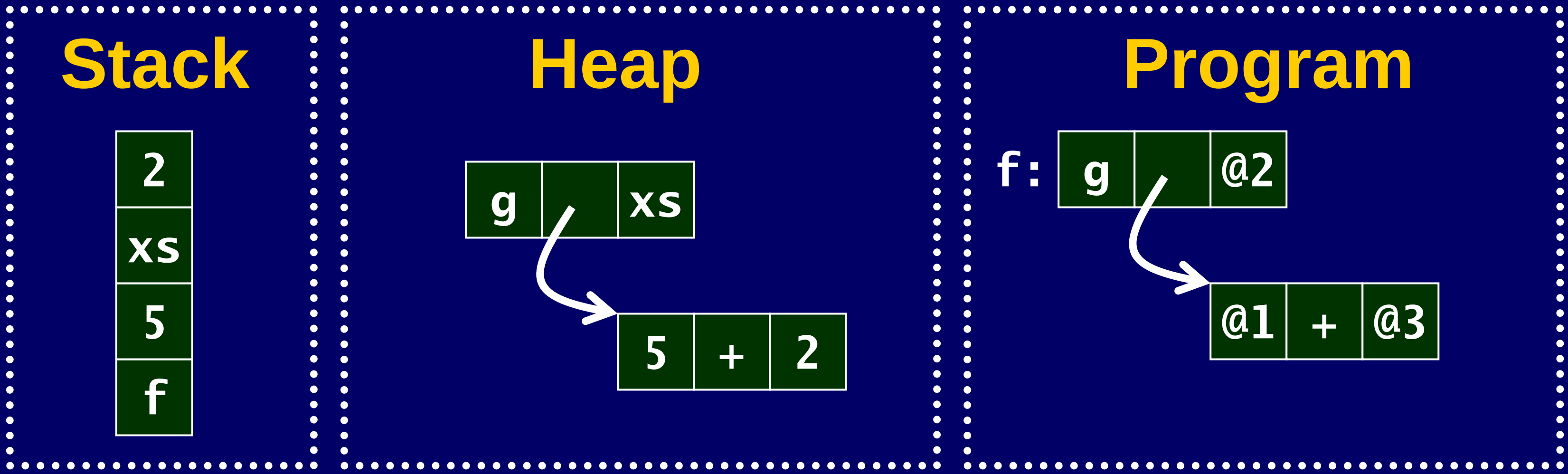
Suppose that function **`f`** is defined by

```
f x xs a   =   g (x+a) xs
```

where **`g`** is a function, and **`+`** is primitive addition.

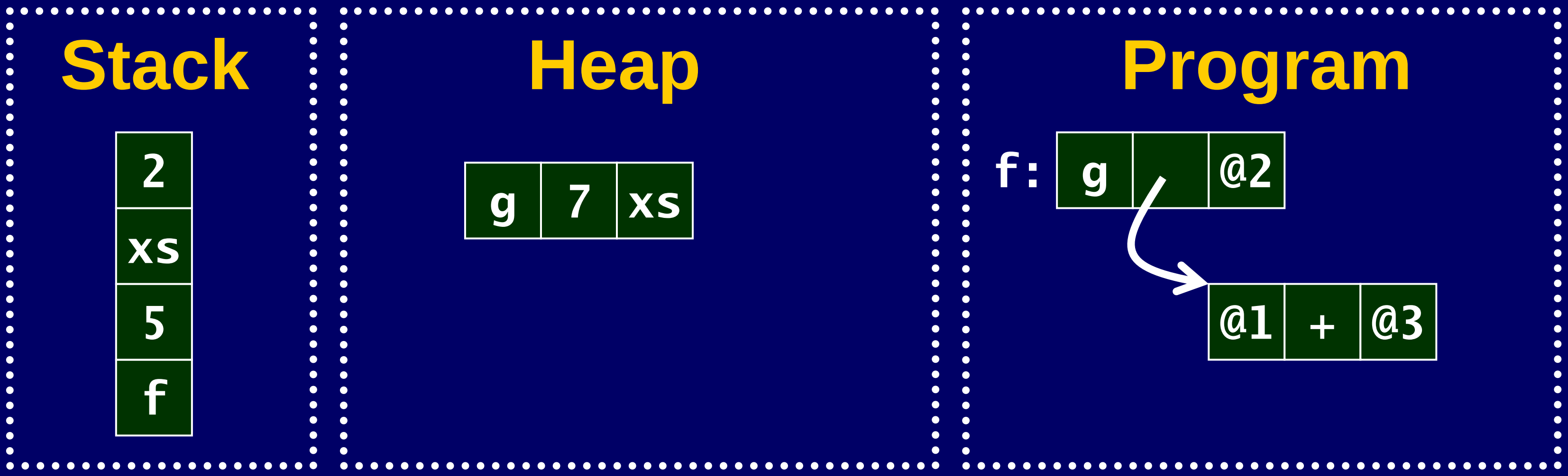
Primitive redexes

Application of **`f`** results in the ***primitive redex*** **`5+2`** being instantiated on the heap.



Speculative evaluation

Idea: *at runtime, look at the arguments to a primitive function. If they are already evaluated, apply the primitive speculatively.*



Results and to-do list

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”)

Improvements, March 2009

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**

So now within a factor of 2 of leading Haskell implementation.

To-do list

- Critical path reduction
- Parallel garbage collection (low vs. high-level)
- Compile-time optimisation
  - Supercompilation (Neil Mitchell, Jason Reich)
- Speculative evaluation of primitive redexes
- Dual-core Reduceron
- Larger, off-chip heap?
- Show off the *description language*!