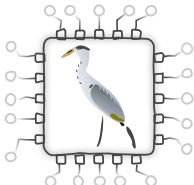
Once bittern, twice shy

Revisiting hardware architectures for lazy functional languages with Heron

Craig Ramsay & Rob Stewart

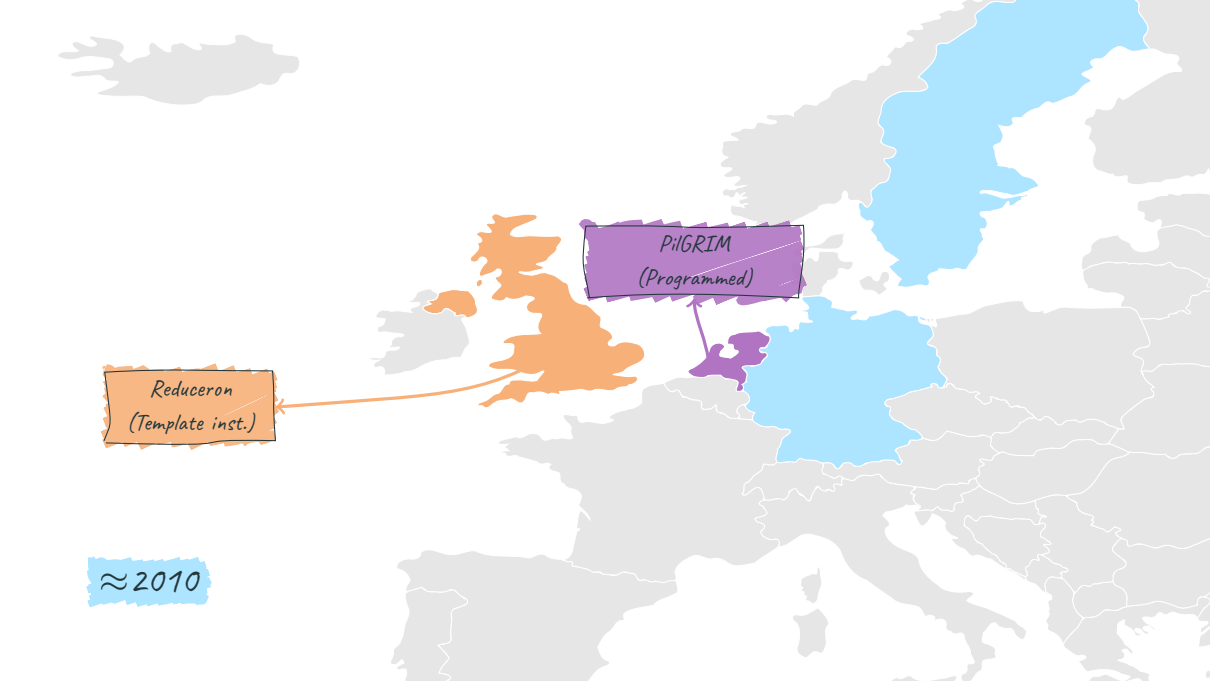
September 2024

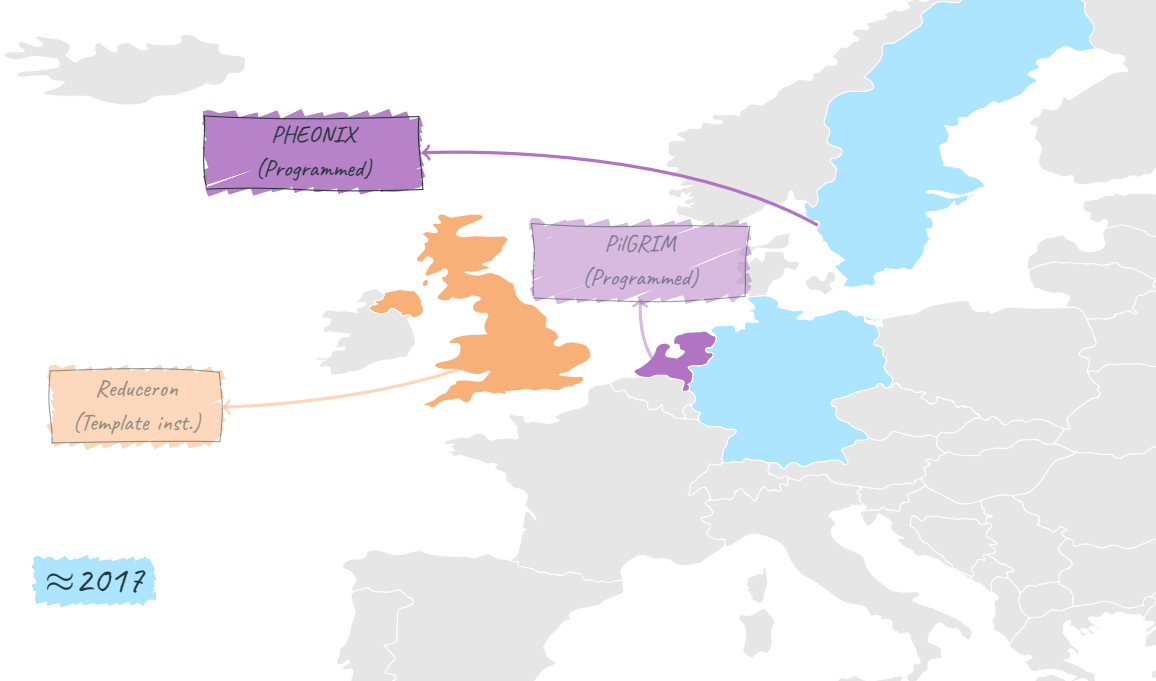
Heriot-Watt University

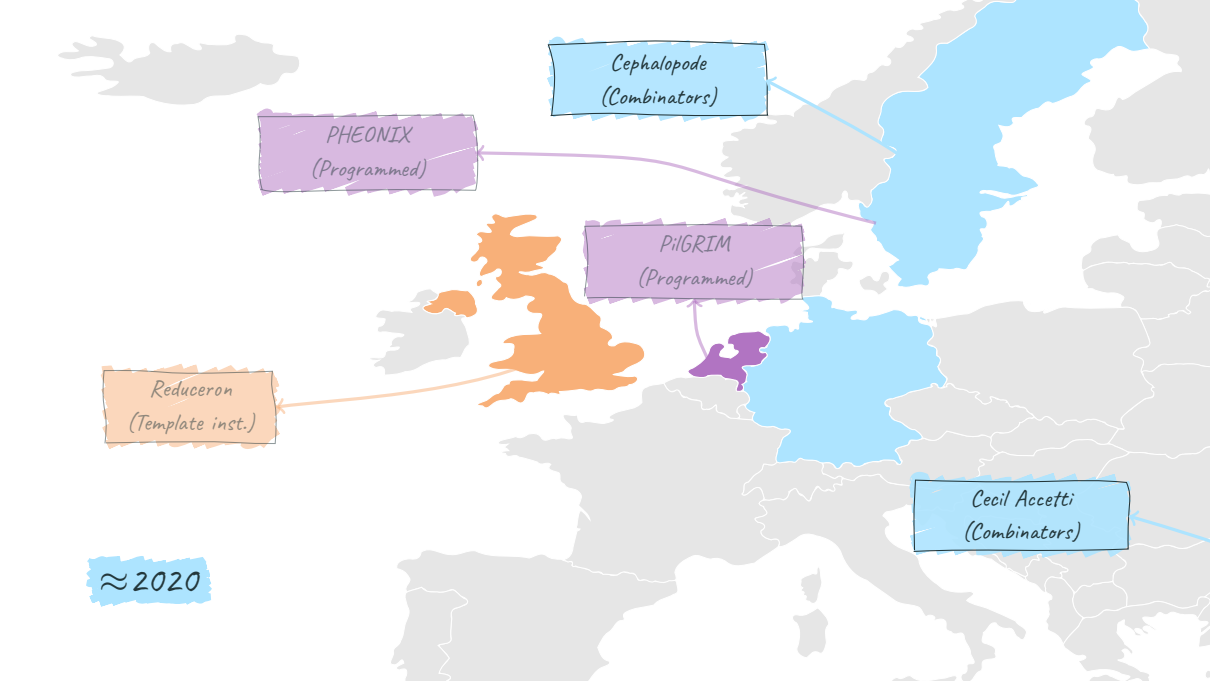


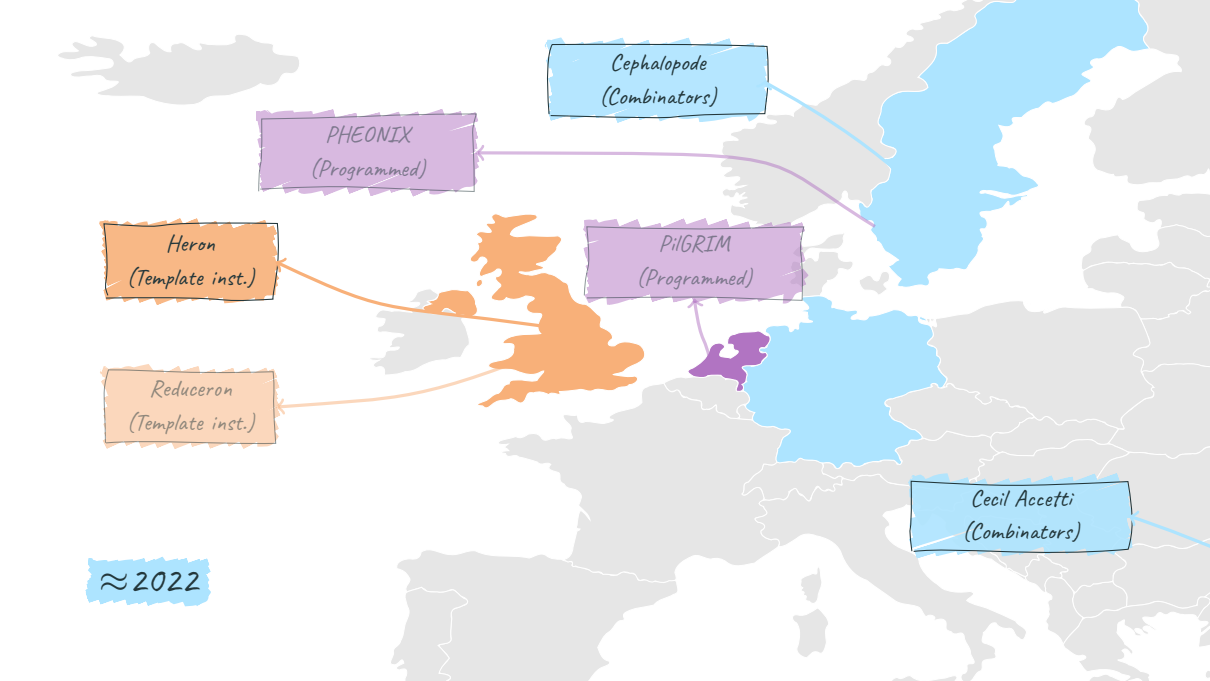

https://haflang.github.io/history

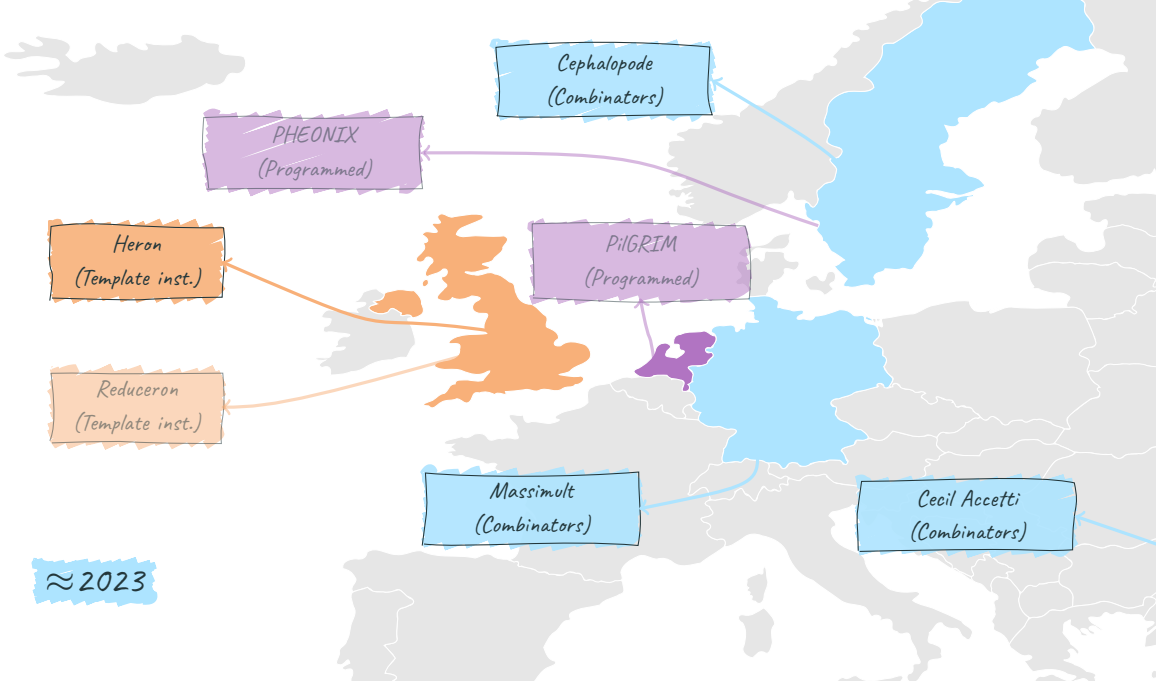
Craig 29.5.99 Self portrait.


PilGRIM
(Programmed)
Reduceron
(Template inst.)
≈2010


PHEONIX
(Programmed)
PilGRIM
(Programmed)
Reduceron
(Template inst.)
≈2017


Cephalopode
(Combinators)
PHEONIX
(Programmed)
PilGRIM
(Programmed)
Reduceron
(Template inst.)
Cecil Accetti
(Combinators)
≈2020


Cephalopode
(Combinators)
PHEONIX
(Programmed)
Heron
(Template inst.)
PilGRIM
(Programmed)
Reduceron
(Template inst.)
Cecil Accetti
(Combinators)
≈2022

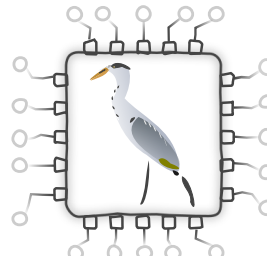

Cephalopode
(Combinators)
PHEONIX
(Programmed)
Heron
(Template inst.)
PilGRIM
(Programmed)
Reduceron
(Template inst.)
Massimult
(Combinators)
Cecil Accetti
(Combinators)
≈2023

HAFLANG Project

Heron

noun [C]

/"herɔn/



A **graph reduction** processor.

Performs template instantiation in one clock cycle via multiple, wide, multi-ported memories.

[0] Ramsay and Stewart, "Heron: Modern Graph Reduction Hardware".

Translating functions to templates

1) Source

```
sum acc xs = case xs of
  []     -> acc
  (y:ys) -> sum (acc + y) ys
```

Translating functions to templates

1) Source

```
sum acc xs = case xs of
  []     -> acc
  (y:ys) -> sum (acc + y) ys
```

2) Lifted case alts

```
sum acc xs = case xs of
  Nil       -> altNil       acc
  Cons y ys -> altCons y ys acc
```

```
altNil       acc = acc
altCons y ys acc = sum (acc + y) ys
```

Translating functions to templates

2) Lifted case alts

```
sum acc xs = case xs of
  Nil       -> altNil       acc
  Cons y ys -> altCons y ys acc
```

```
altNil       acc = acc
altCons y ys acc = sum (acc + y) ys
```

Translating functions to templates

3) Using case tables

```
sum acc xs = xs <altCon;altNil> acc
altNil       acc = acc
altCon y ys acc = sum (acc + y) ys
```

2) Lifted case alts

```
sum acc xs = case xs of
  Nil       -> altNil       acc
  Cons y ys -> altCons y ys acc
```

```
altNil       acc = acc
altCons y ys acc = sum (acc + y) ys
```

Translating functions to templates

3) Using case tables

```
sum acc xs = xs <altCon;altNil> acc
altNil       acc = acc
altCon y ys acc = sum (acc + y) ys
```

Translating functions to templates

3) Using case tables

```
sum acc xs = xs <altCon;altNil> acc
altNil       acc = acc
altCon y ys acc = sum (acc + y) ys
```

4) Flatten to ANF

```
sum acc xs = xs <altCon;altNil> acc
altNil       acc = acc
altCon y ys acc = let z = acc + y
                  in sum z ys
```

Template syntax

$e ::=$	**Atoms**	
	$CON\ a\ n$	(Constructor tag)
$\mid$	$INT\ n$	(Primitive integers)
$\mid$	$PRI\ a\ \otimes$	(Primitive operation)
$\mid$	$FUN\ s\ a\ n$	(Function pointer)
$\mid$	$ARG\ s\ n$	(Argument pointer)
$\mid$	$VAR\ s\ n$	(Application pointer)
$\mid$	$REG\ n$	(Primitive register pointer)

$u, v ::=$	**Applications**	
	$APP\ \bar{e}$	(Normal application)
$\mid$	$CASE\ c\ \bar{e}$	(App with case table)
$\mid$	$PRIM\ n\ \bar{e}$	(PRS candidate)
$c ::= TAB\ n$	**Case table pointer**	
$t ::= let\ \bar{u}\ in\ v$	**Template**	

where $n :: Int$, $a :: Arity$, $s :: IsShared$

Template syntax

$e ::=$	**Atoms**	
	$CON\ a\ n$	(Constructor tag)
$\mid$	$INT\ n$	(Primitive integers)
$\mid$	$PRI\ a\ \otimes$	(Primitive operation)
$\mid$	$FUN\ s\ a\ n$	(Function pointer)
$\mid$	$ARG\ s\ n$	(Argument pointer)
$\mid$	$VAR\ s\ n$	(Application pointer)
$\mid$	$REG\ n$	(Primitive register pointer)

$u, v ::=$	**Applications**	
	$APP\ \overline{e}$	(Normal application)
$\mid$	$CASE\ c\ \overline{e}$	(App with case table)
$\mid$	$PRIM\ n\ \overline{e}$	(PRS candidate)
$c ::= TAB\ n$	**Case table pointer**	
$t ::= let\ \overline{u}\ in\ v$	**Template**	

where $n :: Int$, $a :: Arity$, $s :: IsShared$

Template syntax

$e ::=$		**Atoms**
	$CON\ a\ n$	(Constructor tag)
	$\mid INT\ n$	(Primitive integers)
	$\mid PRI\ a\ \otimes$	(Primitive operation)
	$\mid FUN\ s\ a\ n$	(Function pointer)
	$\mid ARG\ s\ n$	(Argument pointer)
	$\mid VAR\ s\ n$	(Application pointer)
	$\mid REG\ n$	(Primitive register pointer)

$u, v ::=$		**Applications**
	$APP\ \overline{e}$	(Normal application)
	$\mid CASE\ c\ \overline{e}$	(App with case table)
	$\mid PRIM\ n\ \overline{e}$	(PRS candidate)
$c ::= TAB\ n$		**Case table pointer**
$t ::= let\ \overline{u}\ in\ v$		**Template**

where $n :: Int$, $a :: Arity$, $s :: IsShared$

Template syntax

e ::=	**Atoms**	
	CON a n	(Constructor tag)
|	INT n	(Primitive integers)
|	PRI a ⊗	(Primitive operation)
|	FUN s a n	(Function pointer)
|	ARG s n	(Argument pointer)
|	VAR s n	(Application pointer)
|	REG n	(Primitive register pointer)

u, v ::=	**Applications**	
	APP $\bar{e}$	(Normal application)
|	CASE c $\bar{e}$	(App with case table)
|	PRIM n $\bar{e}$	(PRS candidate)
c ::=	TAB n	**Case table pointer**
t ::=	let $\bar{u}$ in v	**Template**

where n :: Int, a :: Arity, s :: IsShared

Template bounds

```
let   APP [   ARG True 0,      PRI 2 +,      INT 1 ]
      APP [   FUN True 2 0,    VAR False 0,  ARG False 1 ]

in    APP [   CON 2 0,        ARG True 0,   VAR False 1 ]
```

Template bounds

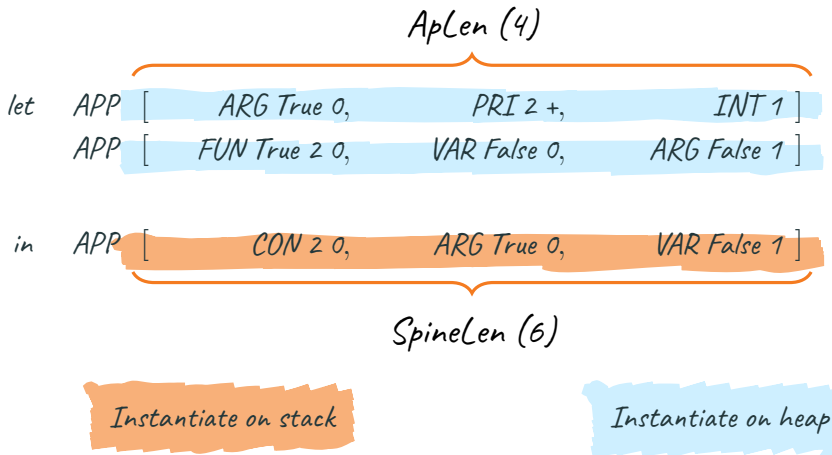
```
let  APP [  ARG True 0,   PRI 2 +,   INT 1 ]
     APP [  FUN True 2 0, VAR False 0, ARG False 1 ]

in   APP [  CON 2 0,      ARG True 0, VAR False 1 ]
```

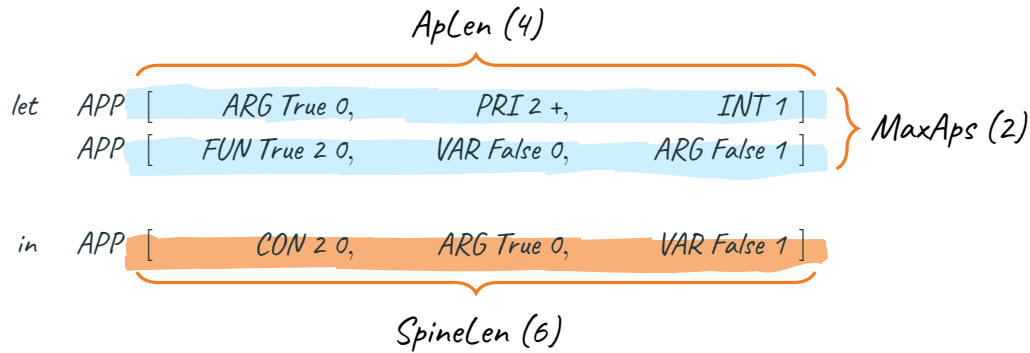
SpineLen (6)

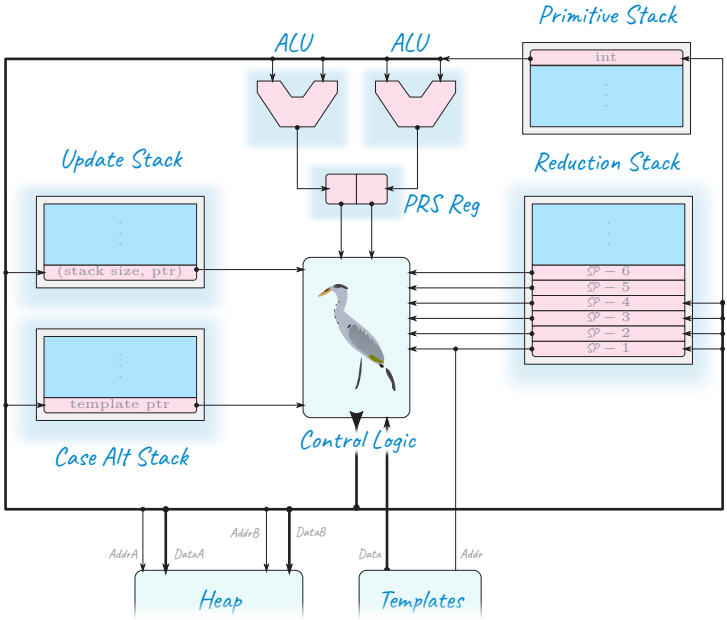
Instantiate on stack

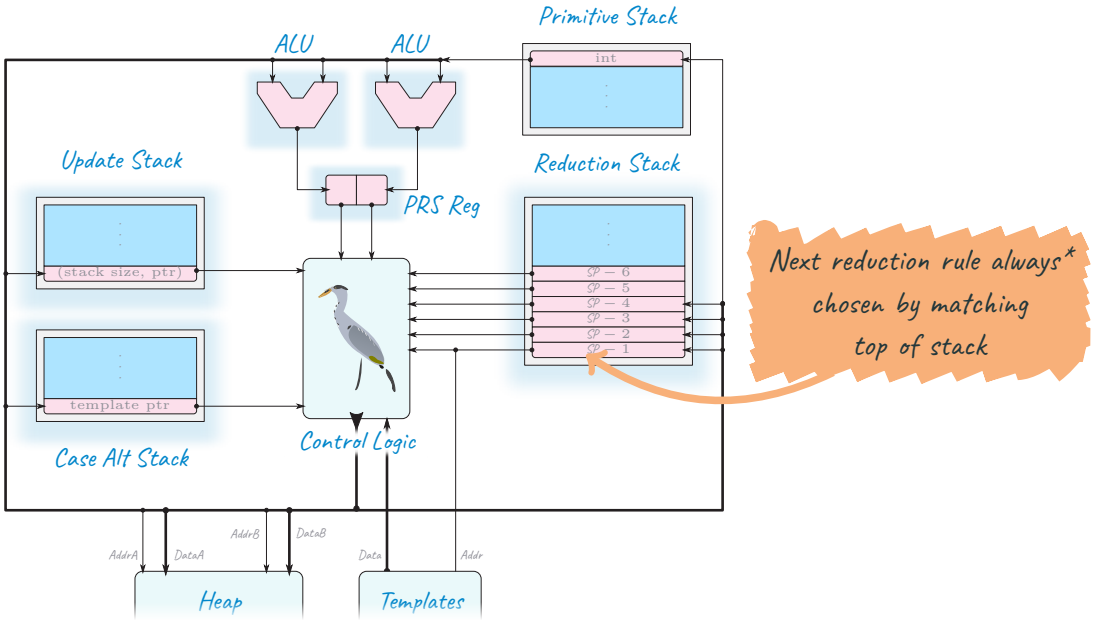
Template bounds

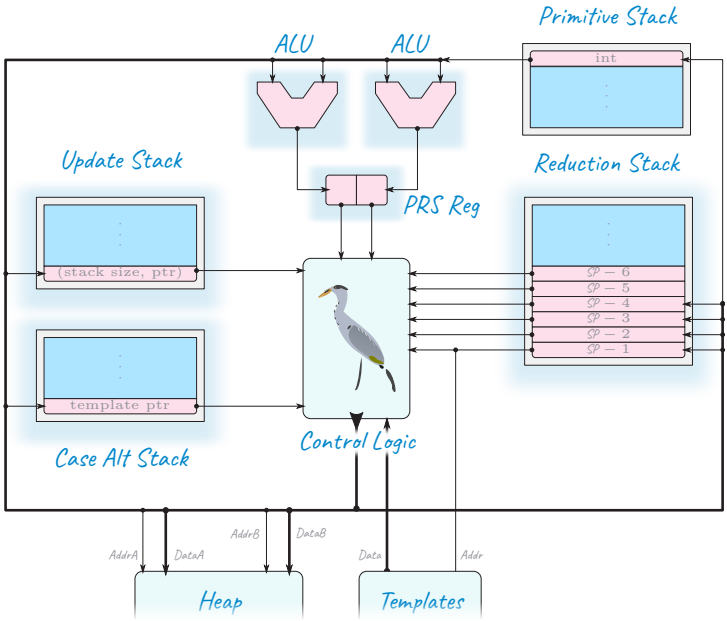


Template bounds



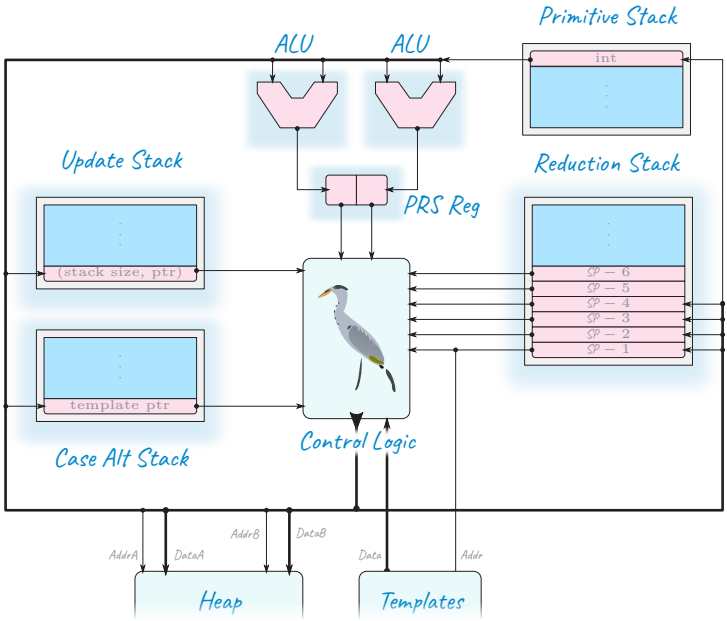

Primitive Stack
ALU
ALU
int
Update Stack
Reduction Stack
PRS Reg
(stack size, ptr)
SP − 6
SP − 5
SP − 4
SP − 3
SP − 2
SP − 1
template ptr
Case Alt Stack
Control Logic
AddrA
DataA
AddrB
DataB
Data
Addr
Heap
Templates


Primitive Stack
ALU
ALU
int
Update Stack
Reduction Stack
PRS Reg
(stack size, ptr)
SP − 6
SP − 5
SP − 4
SP − 3
SP − 2
SP − 1
template ptr
Case Alt Stack
Control Logic
Next reduction rule always* chosen by matching top of stack
AddrA
DataA
AddrB
DataB
Data
Addr
Heap
Templates



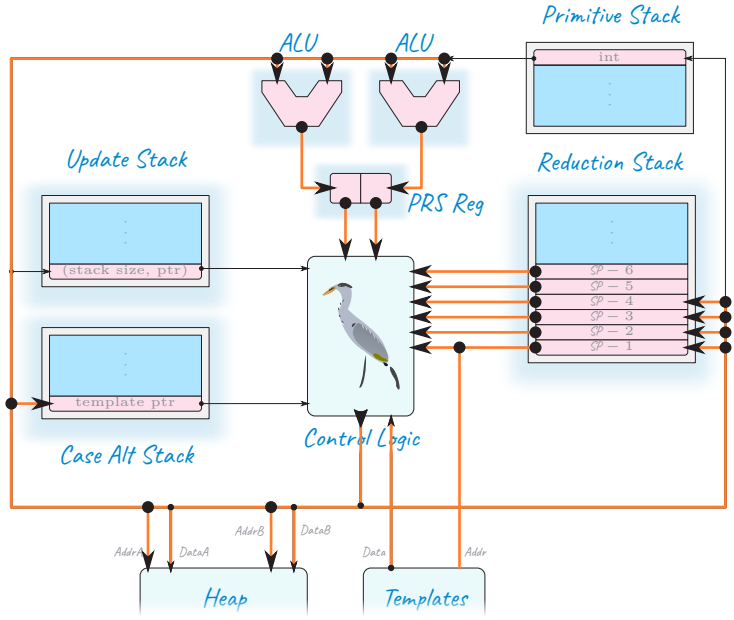
Atoms

= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n



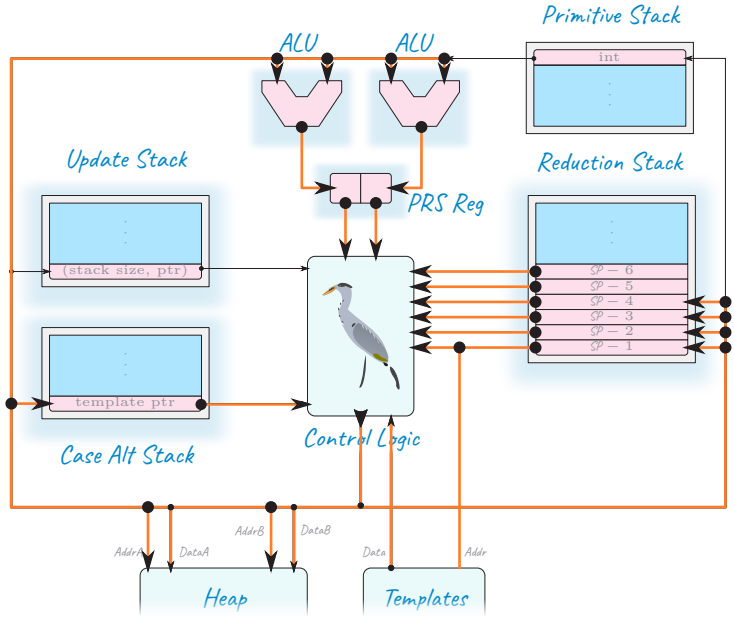
Atoms

```
= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n
```



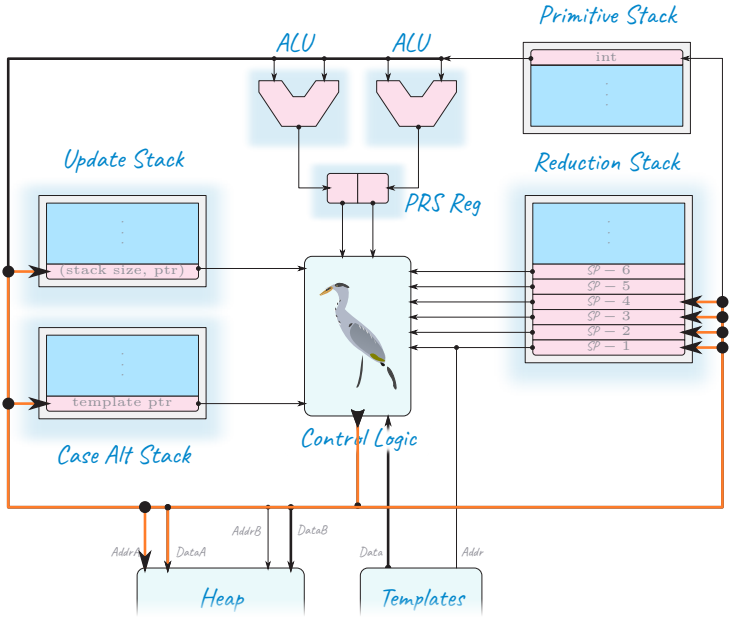
Atoms

```
= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n
```



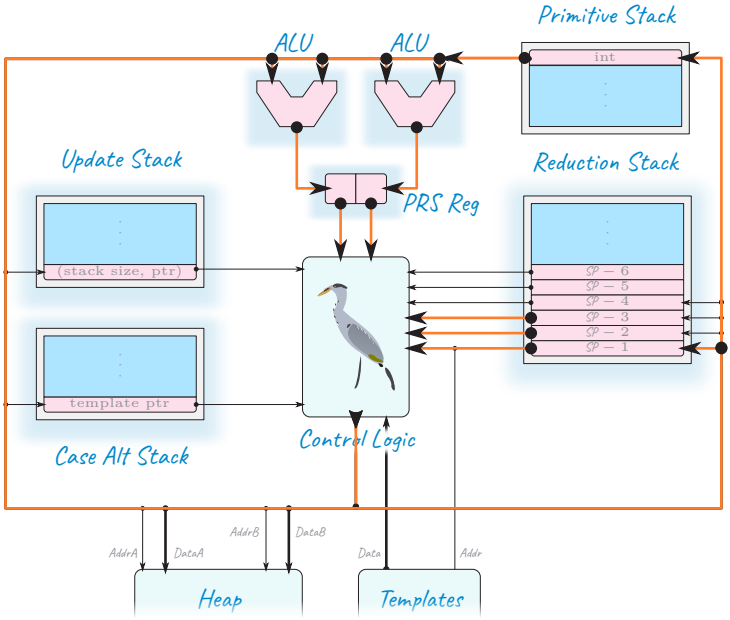
Atoms

= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n



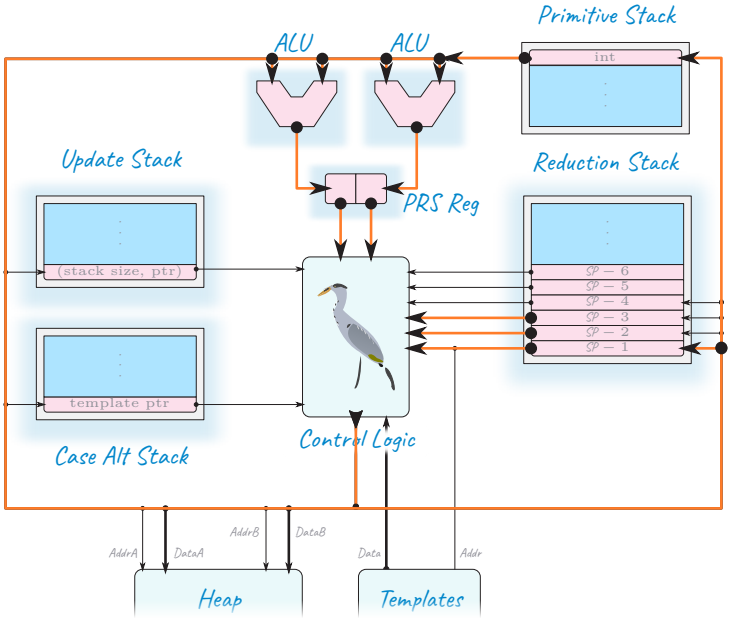
Atoms

```
= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n
```



Atoms

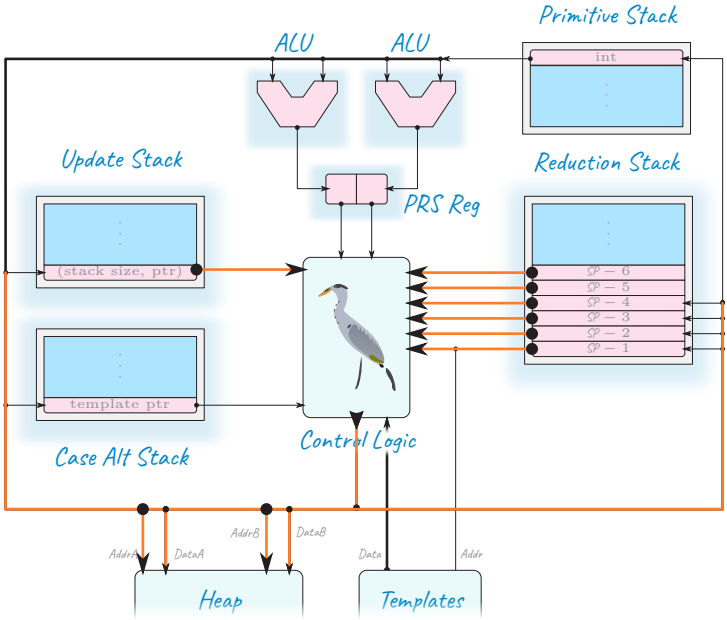
= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n



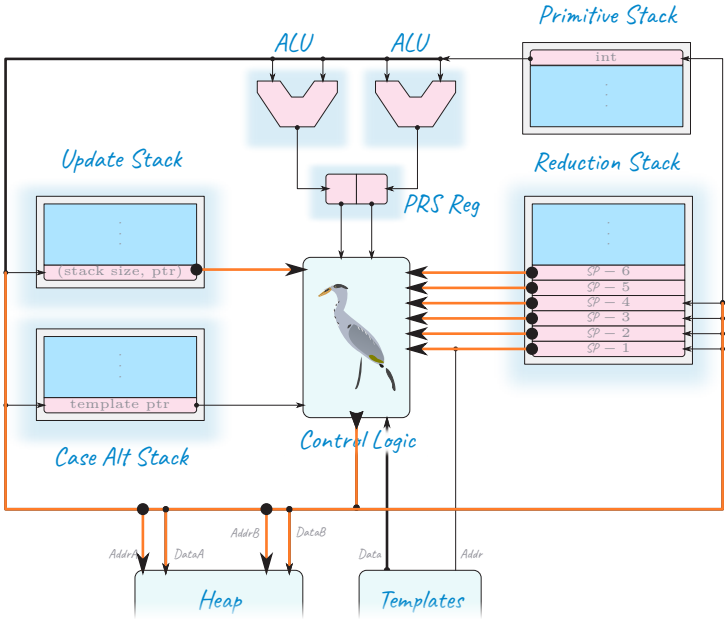
Postfix prims for long spines

(f x y) + (g z)

⇒ f x y g z +



...But what about heap updates?



Avoid most updates via run-time sharing analysis!

Atoms

```
= FUN s a n
| CON a n
| VAR s n
| INT n
| PRI a ⊗
| ARG s n
| REG n
```

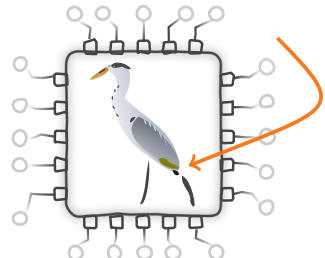
Cloaca

noun [C]

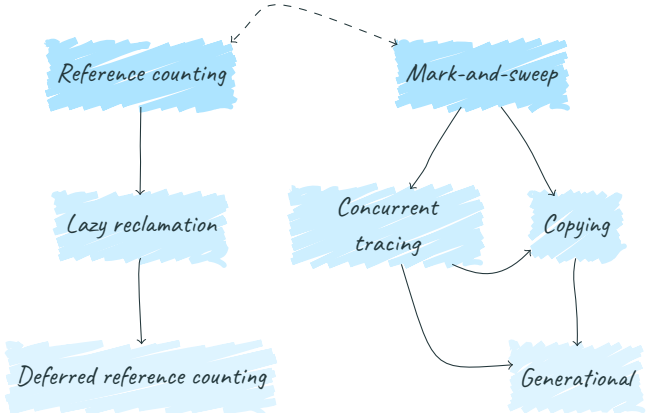
/kloh-ah-kuh/

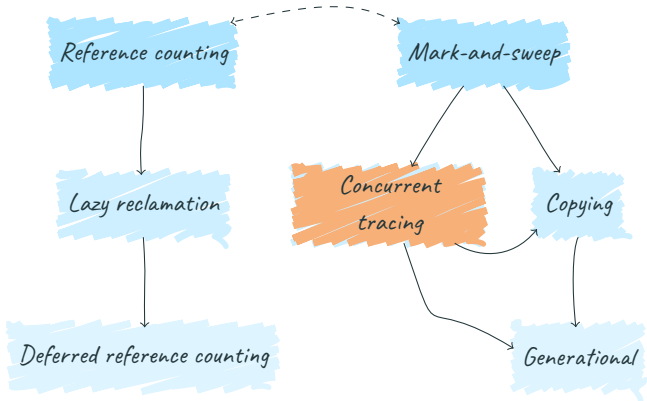
A concurrent hardware

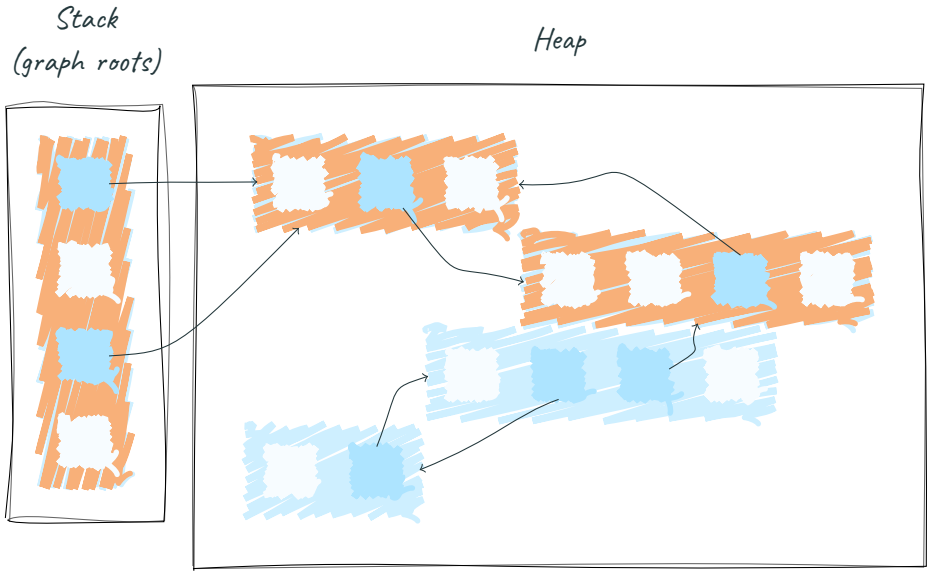
garbage collector for Heron

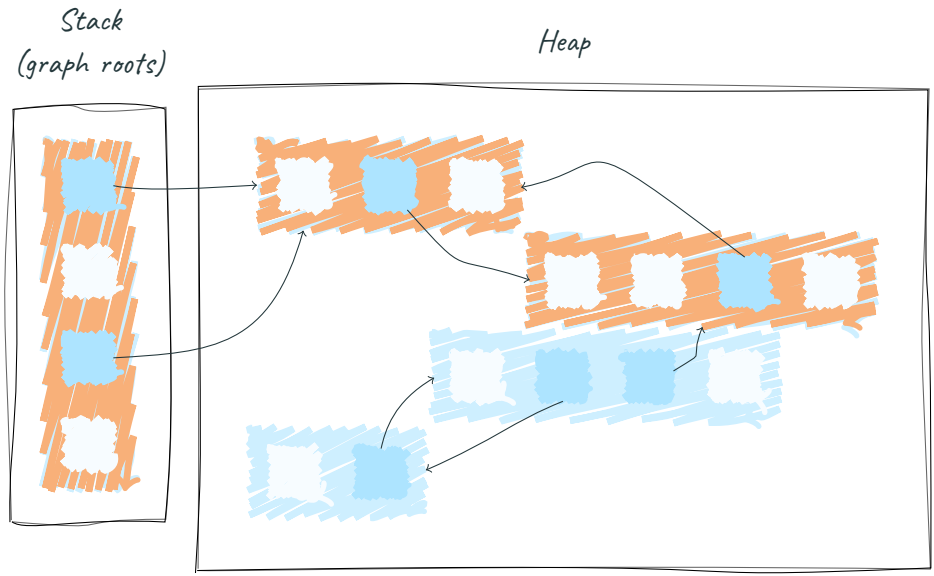


[0] Ramsay and Stewart, "Cloaca: A Concurrent Hardware Garbage Collector for Non-strict Functional Languages".

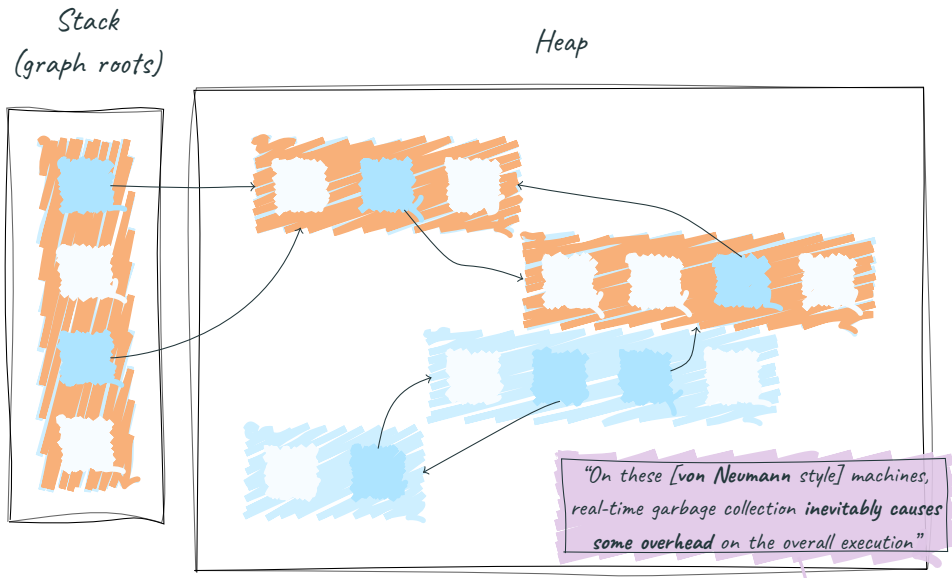

Reference counting
Mark-and-sweep
Lazy reclamation
Concurrent tracing
Copying
Deferred reference counting
Generational


Reference counting
Mark-and-sweep
Lazy reclamation
Concurrent tracing
Copying
Deferred reference counting
Generational


Stack
(graph roots)
Heap



[0] Yuasa, "Real-time garbage collection on general-purpose machines".


Stack
(graph roots)
Heap
"On these [von Neumann style] machines, real-time garbage collection inevitably causes some overhead on the overall execution"
[0] Yuasa, "Real-time garbage collection on general-purpose machines".

Challenges for concurrent software implementation?

Expect 21% median slowdown for nofib!

1) Allocation depends on GC state

Stop-the-world GC

```
Function alloc (app):
    heap[hp] ⟵ app
    hp++
```

vs

Concurrent GC

```
Function alloc (app):
    if allocBarrier(gcPhase, hp)
    then
        tag hp as Marked
    else
        tag hp as Unmarked
    heap[hp] ⟵ app
    hp++
```

2) Non-moving GC needs complex allocation

Stop-the-world GC

```
Function alloc (app):
    heap[hp] ⟵ app
    hp++
```

vs

Concurrent GC

```
Function alloc (app):
    a ⟵ pop from freelist
    if allocBarrier(gcPhase, a)
    then
        tag a as Marked
    else
        tag a as Unmarked
    heap[a] ⟵ app
```

3) Prevent graph updates from destroying edges

Stop-the-world GC

vs

Concurrent GC

```
Function update (nf, a):
    heap[a] ⟵ nf
```

```
Function update (nf, a):
    if updateBarrier(gcPhase) then
        x ⟵ heap[a]
        forall y in x's child pointers do
            remember y for marking
    heap[a] ⟵ nf
```

Additional hardware-enabled optimisations

Heron's existing dynamic update avoidance system...

```
data Atom
  = ...
  | Var Shared Int
  | Arg Shared Int
  ...
```

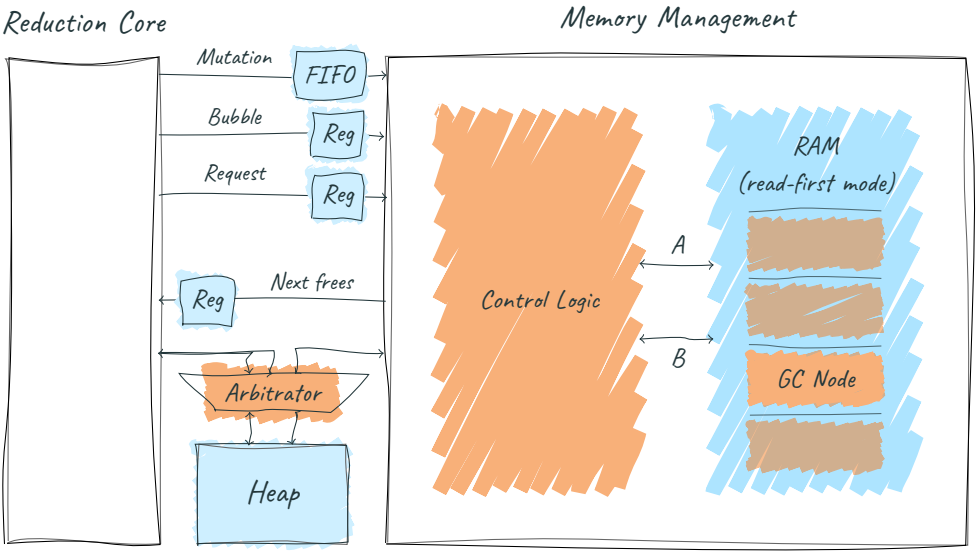
```
Function unwind (a, shared):
  ...
  ...
  ...
  if shared and not NF then
    push a onto update stack
```

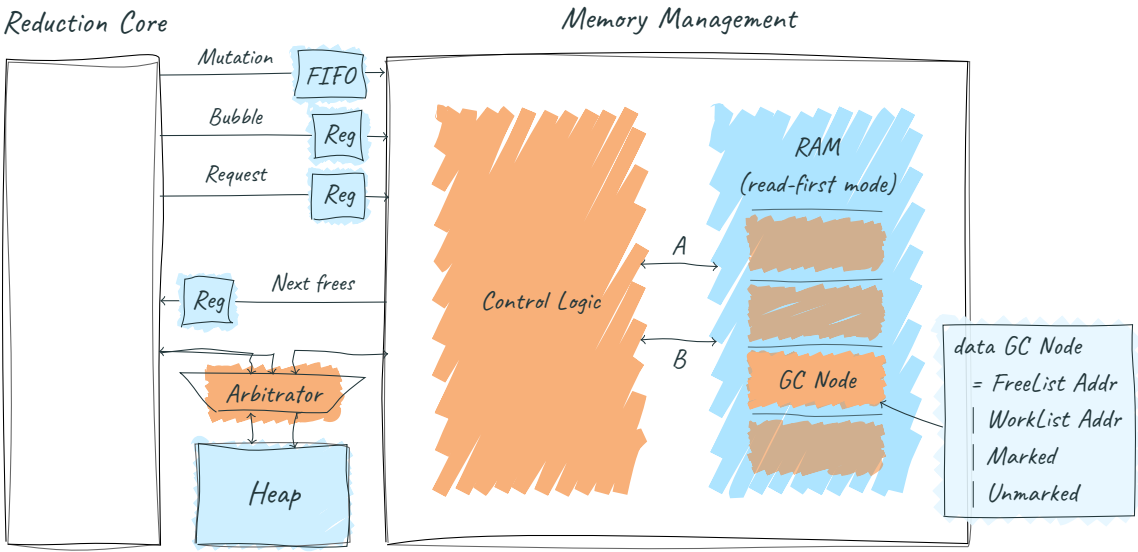
Heron's existing dynamic update avoidance system...

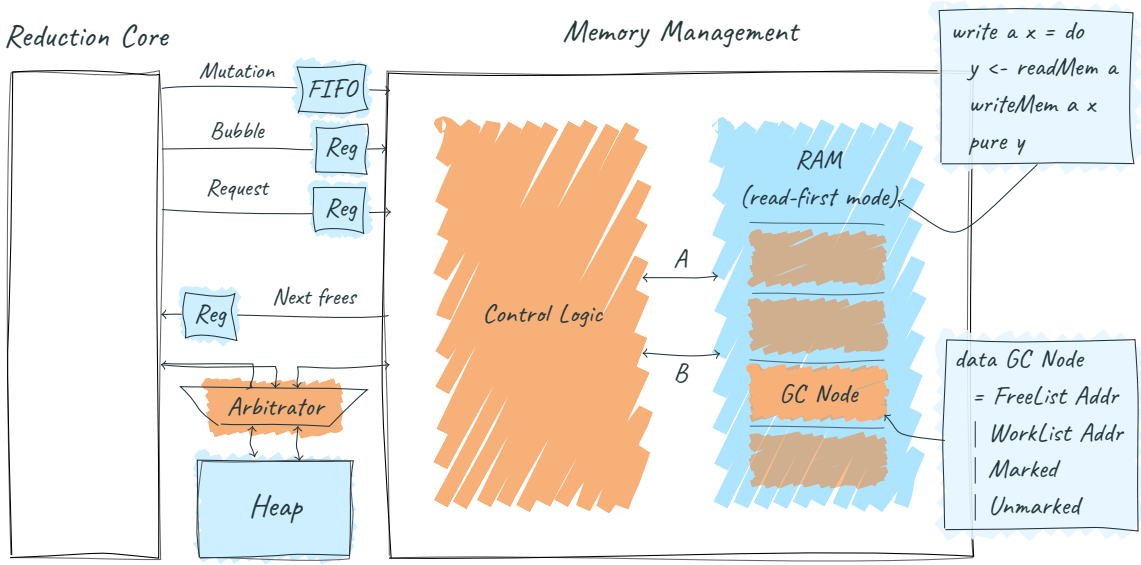
```
data Atom
  = ...
  | Var Shared Int
  | Arg Shared Int
    ...
```

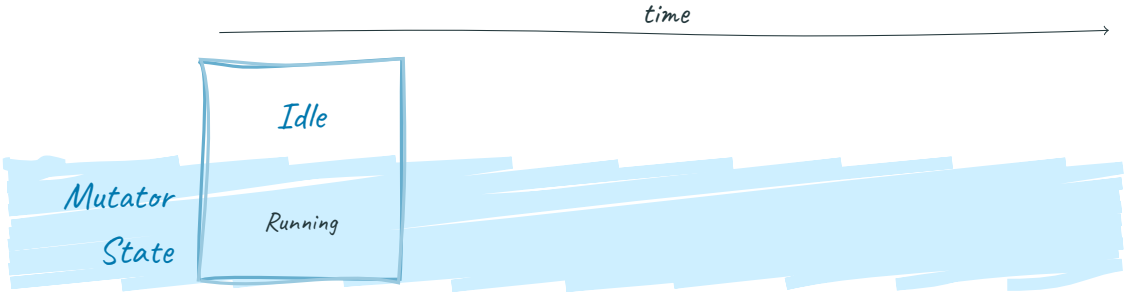
```
Function unwind (a, shared):
    ...
    ...
    ...
    if shared and not NF then
        push a onto update stack
    if not shared then
        dealloc a
```

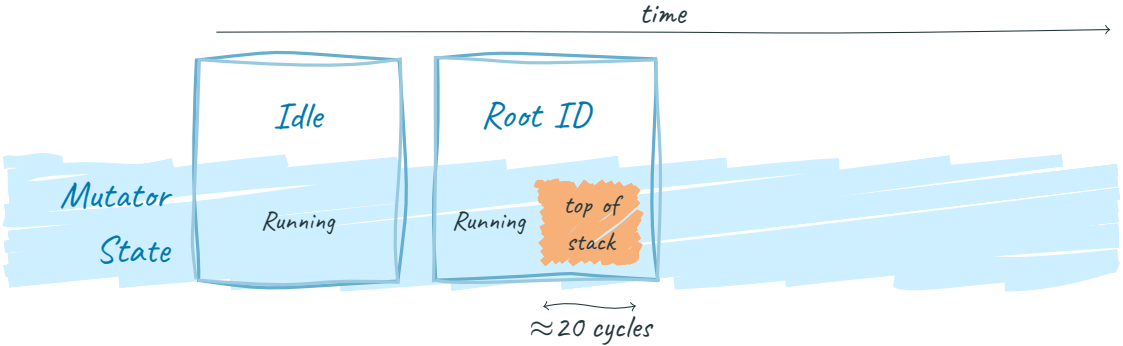
... is just one-bit reference counting with a hat on.

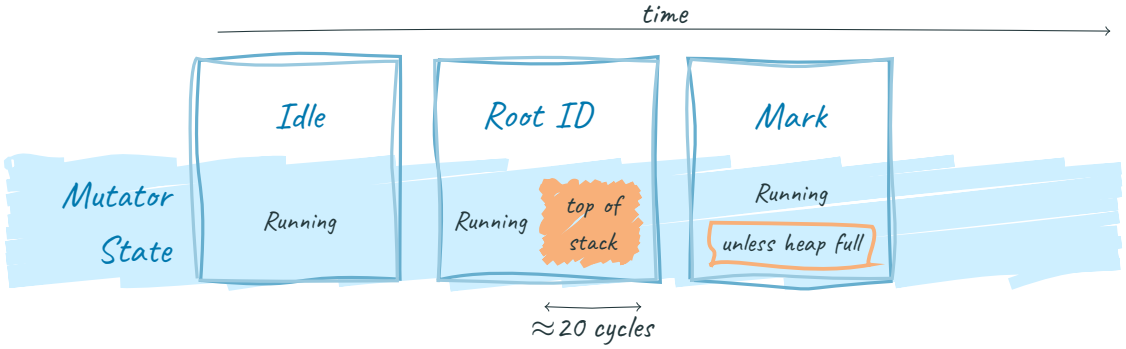

Reduction Core
Memory Management
Mutation
FIFO
Bubble
Reg
Request
Reg
Reg
Next frees
Arbitrator
Heap
Control Logic
A
B
RAM
(read-first mode)
GC Node

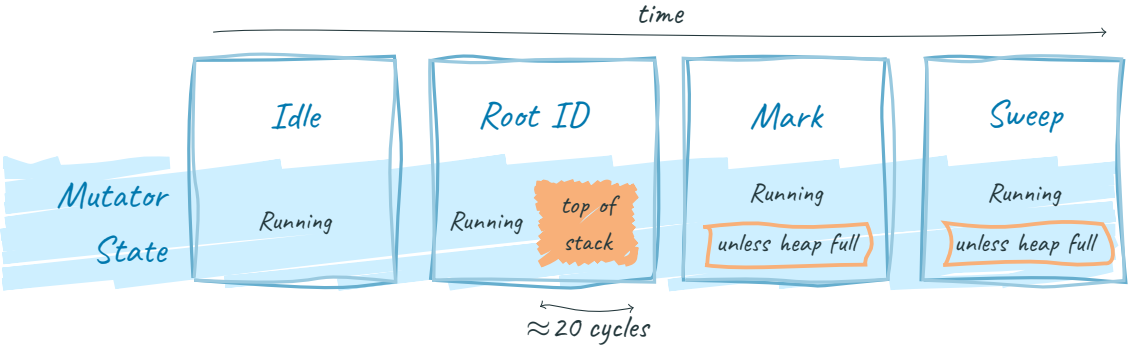

Reduction Core
Memory Management
Mutation
FIFO
Bubble
Reg
Request
Reg
Reg
Next frees
Arbitrator
Heap
Control Logic
A
B
RAM
(read-first mode)
GC Node
data GC Node
= FreeList Addr
WorkList Addr
Marked
Unmarked


Reduction Core
Memory Management
Mutation
FIFO
Bubble
Reg
Request
Reg
Reg
Next frees
Arbitrator
Heap
Control Logic
A
B
RAM
(read-first mode)
GC Node
write a x = do
y <- readMem a
writeMem a x
pure y
data GC Node
= FreeList Addr
| WorkList Addr
| Marked
| Unmarked


time
Idle
Mutator
State
Running


time
Idle
Root ID
Mutator State
Running
Running
top of stack
≈20 cycles


time
Idle
Root ID
Mark
Mutator State
Running
Running
top of stack
Running
unless heap full
≈20 cycles


time
Idle
Root ID
Mark
Sweep
Mutator State
Running
Running
top of stack
Running
unless heap full
Running
unless heap full
≈20 cycles

Performance

	Peak GHC working set (KB)	GHC Allocations (MB)	LoC
Adjoxo	47	301	72
Braun	46	0	26
Cichelli	52	41	123
Clausify	77	364	69
Countdown	46	54	62
Knuthbendix	105	54	324
Mate	137	430	293
Mss	46	359	13
Ordlist	46	984	28
Permsort	46	2320	10
Queens	55	1038	25
Queens2	47	1084	20
Sumpuz	48	1293	72
Taut	47	236	37
While	46	264	89

	Peak GHC working set (KB)	GHC Allocations (MB)	LoC
Adjoxo	47	301	72
Braun	46	0	26
Cichelli	52	41	123
Clausify	77	364	69
Countdown	46	54	62
Knuthbendix	105	54	324
Mate	137	430	293
Mss	46	359	13
Ordlist	46	984	28
Permsort	46	2320	10
Queens	55	1038	25
Queens2	47	1084	20
Sumpuz	48	1293	72
Taut	47	236	37
While	46	264	89

	Peak GHC working set (KB)	GHC Allocations (MB)	LoC
Adjoxo	47	301	72
Braun	46	0	26
Cichelli	52	41	123
Clausify	77	364	69
Countdown	46	54	62
Knuthbendix	105	54	324
Mate	137	430	293
Mss	46	359	13
Ordlist	46	984	28
Permsort	46	2320	10
Queens	55	1038	25
Queens2	47	1084	20
Sumpuz	48	1293	72
Taut	47	236	37
While	46	264	89

Which Architectures?

Platform	Heron	GHC + Intel i7 1250U	
	Xilinx Alveo U280	Performance	Power-saver

Which Architectures?

	Heron	GHC + Intel i7 1250U	
Platform	Xilinx Alveo U280	Performance	Power-saver
Clock	185 MHz	4.7 GHz	≈2 GHz

Which Architectures?

	Heron	GHC + Intel i7 1250U	
Platform	Xilinx Alveo U280	Performance	Power-saver
Clock	185 MHz	4.7 GHz	≈2 GHz
Power est.	0.8W dynamic + 3.1 W Static[1]	Cores 15 W or Package 16 W	Cores 2W or Package 6 W

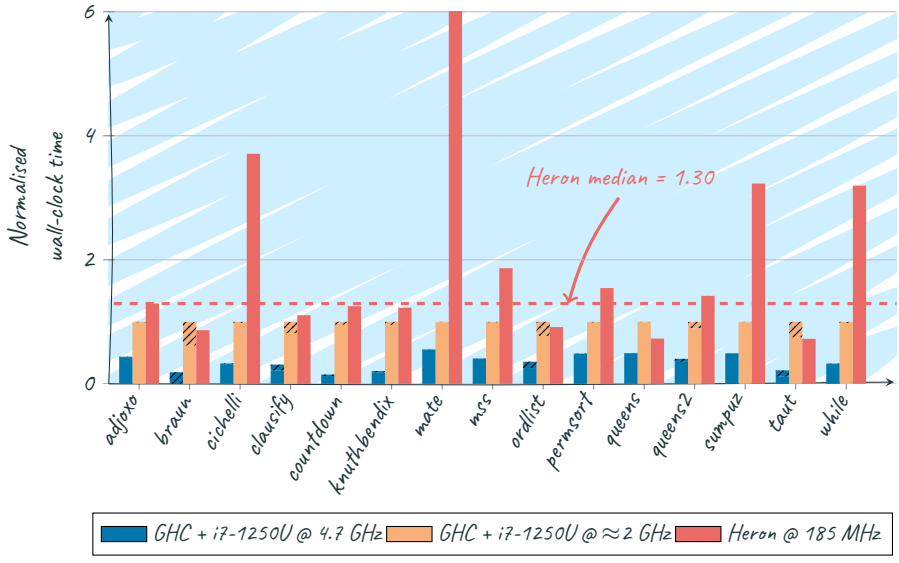
[1] Heron only occupies 1.13% of any resource type though!

Which Architectures?

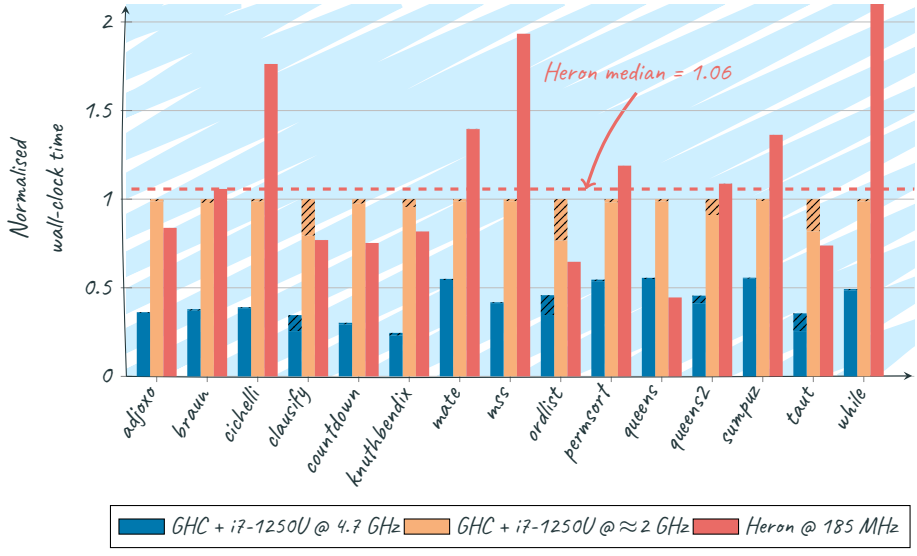
	Heron	GHC + Intel i7 1250U	
Platform	Xilinx Alveo U280	Performance	Power-saver
Clock	185 MHz	4.7 GHz	≈2 GHz
Power est.	0.8W dynamic + 3.1 W Static[1]	Cores 15 W or Package 16 W	Cores 2W or Package 6 W
Fabrication	16 nm (FPGA!)	10 nm	10 nm

[1] Heron only occupies 1.13% of any resource type though!

Wall-clock times vs GHC -O2



Wall-clock times vs GHC -O0



Reflection

Should we widen the local memory bottleneck at all costs?

Should we widen the local memory bottleneck at all costs?

Heron's single-thread performance is competitive with general purpose CPUs!

Should we widen the local memory bottleneck at all costs?

Heron's single-thread performance is competitive with general purpose CPUs!

We may have optimisied single-thread
performance to the point of limiting a parallel system[2]...

[2] Context switching is expensive when thread state is leaks out into many memories

Should we widen the local memory bottleneck at all costs?

Heron's single-thread performance is competitive with general purpose CPUs!

We may have optimisied single-thread
performance to the point of limiting a parallel system[2]...

Cephalopode and Heron at the two extremes here.

[2] Context switching is expensive when thread state is leaks out into many memories

How/should we embrace pipelining?

How/should we embrace pipelining?

Non-pipelined core feels natural for template instantiation (or any scheme without an instruction stream)

How/should we embrace pipelining?

Non-pipelined core feels natural for template instantiation (or any scheme without an instruction stream)

Obviously this limits clock frequency...but max clock frequency has been surprisingly fragile and hard to debug

How/should we embrace pipelining?

Non-pipelined core feels natural for template instantiation (or any scheme without an instruction stream)

Obviously this limits clock frequency...but max clock frequency has been surprisingly fragile and hard to debug

Maybe the answer is hardware multi-threading?

Conclusion

A simple, tiny template instantiation core seems to keep up a modern CPU running at x10 speed!

Conclusion

A simple, tiny template instantiation core seems to keep up a modern CPU running at x10 speed!

Assisted by a fully concurrent GC (modulo ≈20 cycles per pass).

Requires several hardware tricks unavailable to software implementations.

Conclusion

A simple, tiny template instantiation core seems to keep up a modern CPU running at x10 speed!

Assisted by a fully concurrent GC (modulo ≈20 cycles per pass).
Requires several hardware tricks unavailable to software implementations.

Lays a path towards a single-chip multi-core architecture.

Conclusion

A simple, tiny template instantiation core seems to keep up a modern CPU running at x10 speed!

Assisted by a fully concurrent GC (modulo ≈20 cycles per pass).
Requires several hardware tricks unavailable to software implementations.

Lays a path towards a single-chip multi-core architecture.

Great to see there is some shared interest in hardware architectures for FP once again!

Questions?