Verification of an ML compiler

Lecture 2:

Data representation and garbage collection

Marktoberdorf Summer School MOD 2017

Magnus O. Myreen, Chalmers University of Technology

Last lecture

Sketched verification of an imperative toy compiler.

Main lemma for proving a compiler phase correct:

source intermediate language (IL)

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒

∃t2. evaluate (compile code) t1 = (res,t2) ∧
     state_rel s2 t2
```

target IL

compiler

Last lecture (cont.)

Sketched verification of a toy imperative compiler.

Source language:

```
t = Dec string t
  | Exp e
  | Break
  | Seq t t
  | If e t t
  | For e e t

e = Var string
  | Num int
  | Add e e
  | Assign string e
```

Target language:

```
inst
  = Add reg reg reg
  | Int reg int
  | Jmp offset
  | JmpIf reg offset
```

Last lecture (cont.)

Sketched verification of a toy imperative compiler.

We can make it less toy.

Source language:

```
t = Dec string t
  | Exp e
  | Break
  | Seq t t
  | If e t t
  | For e e t

e = Var string
  | Num word32
  | Add e e
  | Assign string e
  | MemOp …
```

Target language:

```
inst
 = Add reg reg reg
 | Int reg word32
 | Jmp offset
 | JmpIf reg offset
 | Load reg reg
 | Store reg reg
```

We can use machine types

… and have a memory.

These changes are easy, but make the source lower level.

a short demo

Question: How is ML different?

compared with our improved toy compiler (or a C compiler)

Answer: *Abstraction*

ML builds an illusion:

- ➡ functions are first-class values
- ➡ there are no hard size limits

… *and also:* *Safety*

type-checked

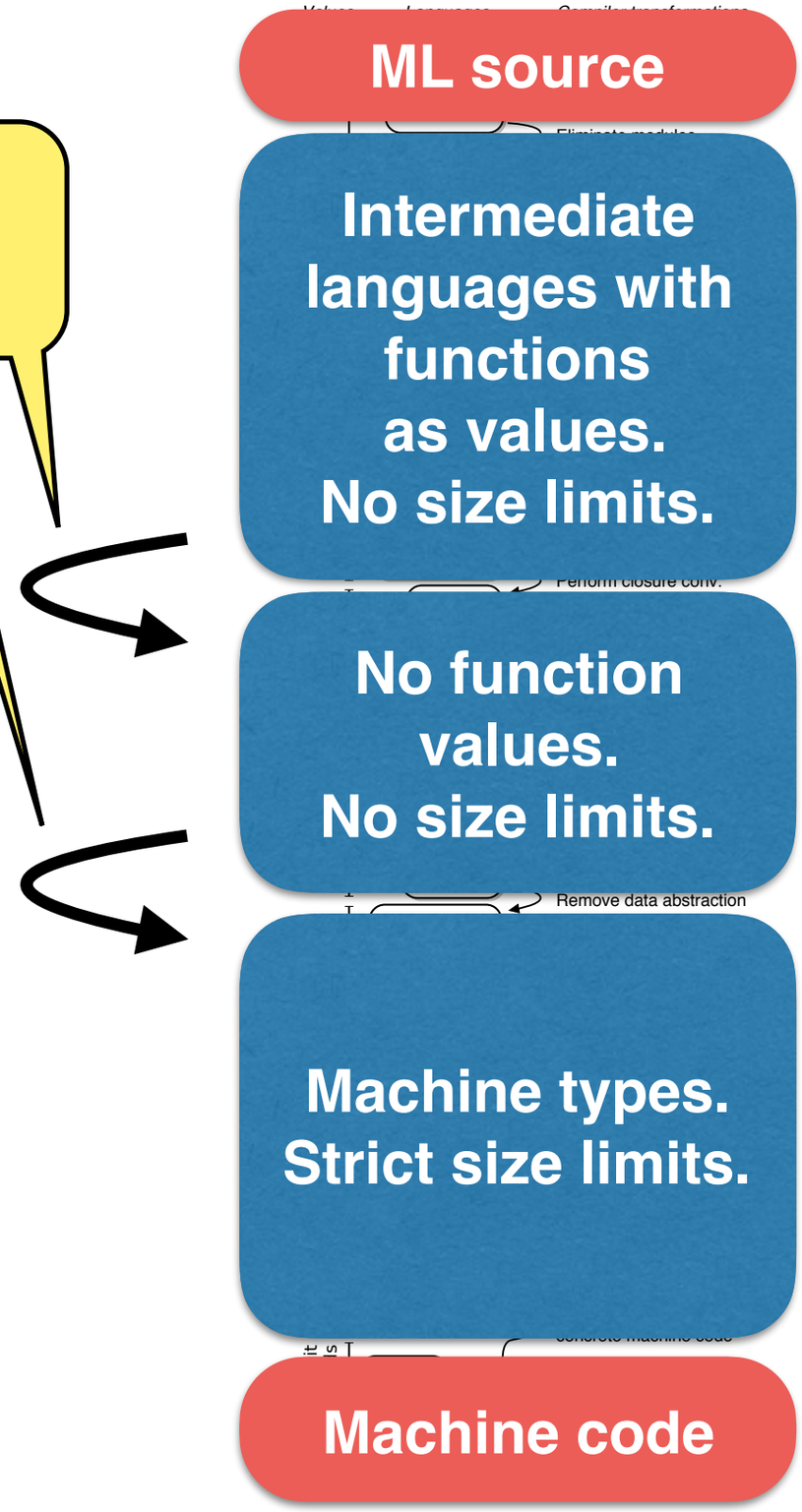
All ML programs have some well-defined semantics (contrast with C)

Implementing the ML abstractions

The two most interesting transitions

function values are implemented
(topic of next lecture)

data abstraction is implemented
(topic of this lecture)



Implementing the ML abstractions

Value type before:

```
v =
    Number int
  | Word64 word64
  | Block num (v list)
  | CodePtr num
  | RefPtr num
```

mathematical integers

tuples / vectors of any length

No function values.
No size limits.

Machine types.
Strict size limits.

Value types after:

machine words (32- or 64-bit)

state contains a memory modelled as finite map

Implementing the ML abstractions

Value type before:

```
v =
    Number int
  | Word64 word64
  | Block num (v list)
  | CodePtr num
  | RefPtr num
```

mathematical integers

tuples / vectors of any length

Value types after:

machine words (32- or 64-bit)

state contains a memory modelled as finite map

This refinement is complicated.

Question: can it be realised?

Implementing the ML abstractions

Value type before:

```
v =
    Number int
  | Word64 word64
  | Block num (v list)
  | CodePtr num
  | RefPtr num
```

mathematical integers

tuples / vectors of any length

Value types after:

machine words (32- or 64-bit)

state contains a memory
modelled as finite map

This refinement is complicated.
It cannot be perfect because the target has fixed size limits.

The tools from Lecture 1 adapted

In Lecture 1, we proved theorems of the form:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2. evaluate (compile code) t1 = (res,t2) ∧
state_rel s2 t2
```

The tools from Lecture 1 adapted

Modified:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2 res2. evaluate (compile code) t1 = (res2,t2) ∧
(state_rel s2 t2 ∧ res2 = res ∨
res2 = RHitHardLimit ∧ t2.IO isPrefix s2.IO)
```

we allow execution with an out-of-memory error

the target list of I/O events must be a prefix of the source I/O events

The tools from Lecture 1 adapted

Modified:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2 res2. evaluate (compile code) t1 = (res2,t2) ∧
          (state_rel s2 t2 ∧ res2 = res ∨
          res2 = RHitHardLimit ∧ t2.IO isPrefix s2.IO)
```

The top-level correctness theorem becomes weaker:

```
machine_semantics (compile prog)
  ∈ extend_with_resource_limit (semantics prog)
```

produce a set of all prefixes …

… of these behaviours

The tools from Lecture 1 adapted

In our simple setting ***without*** I/O:

```
extend_with_resource_limit Crash = { Crash }

extend_with_resource_limit Terminate = { Terminate }

extend_with_resource_limit Diverge = { Diverge, Terminate }
```

can abruptly terminate

Question: Is this too weak?

The top-level correctness theorem becomes weaker:

```
machine_semantics (compile prog)
  ∈ extend_with_resource_limit (semantics prog)
```

produce a set of all prefixes …

… of these behaviours

The tools from Lecture 1 adapted

Modified:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2 res2. evaluate (compile code) t1 = (res2,t2) ∧
          (state_rel s2 t2 ∧ res2 = res ∨
           res2 = RHitHardLimit ∧ t2.IO isPrefix s2.IO)
```

The top-level correctness theorem becomes weaker:

```
machine_semantics (compile prog)
  ∈ extend_with_resource_limit (semantics prog)
```

State relation defines abstraction

CakeML's abs. complicated

Value type before:

```
v =
   Number int
 | Word64 word64
 | Block num (v list)
 | CodePtr num
 | RefPtr num
```

configurable

Value types after:

machine words (32- or 64-bit)

state contains a memory
modelled as finite map

Toy abstraction for this lecture

Value type before:

simplified Lisp
s-expression

```
sexp =
   Num word30
 | Sym word30
 | Dot sexp sexp
```

fixed &
simple

Value types after:

32-bit machine words

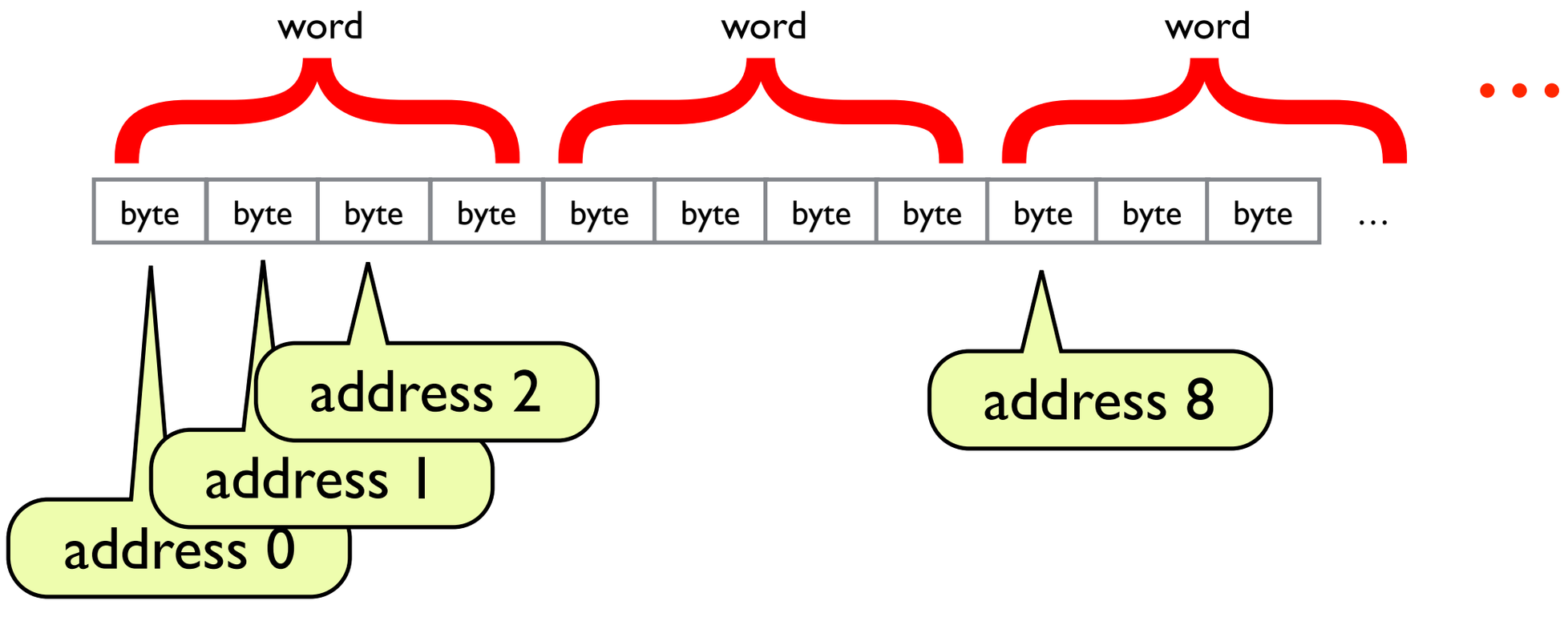
state contains a memory
modelled as finite map

Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)
- 32-bit word = four bytes
- aligned words have address divisible by 4 (called 32-bit aligned)
- word-aligned pointers end in bits '00'



Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

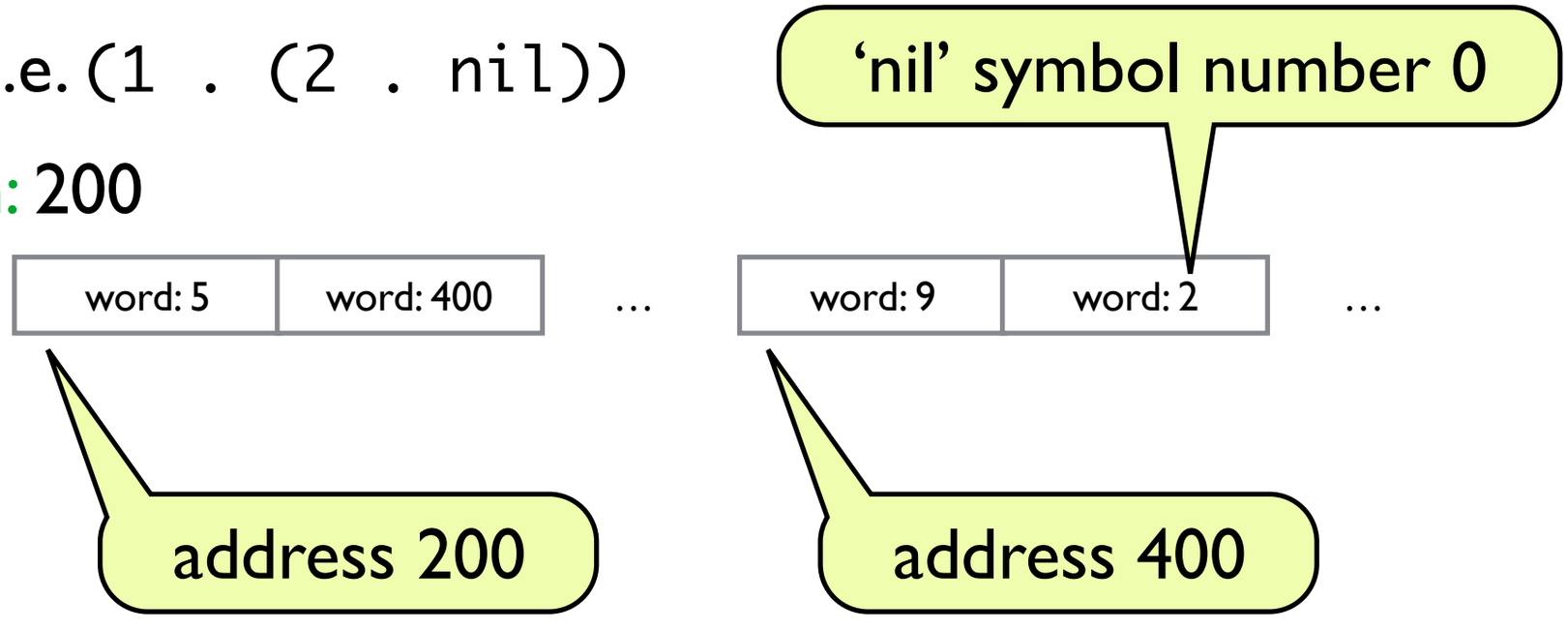
Representation of s-expressions:

- cons-cells (Dot) are repr. as pointer to an aligned pair of words
  i.e. every cons-pointer ends in bits '00'
- numeric value v represented as word 4 × v + 1 (ends in bits '01')
- symbol value s represented as word 4 × s + 2 (ends in bits '10')

Example: `(1 2)` i.e. `(1 . (2 . nil))`

Representation: 200

With memory:



Value relation

```
val_rel (Val n) w (m,b,i) = (w = 4 × n + 1)

val_rel (Sym n) w (m,b,i) = (w = 4 × n + 2)

val_rel (Dot x1 x2) w (m,b,i) =
  word_aligned w ∧ b ≤ w ∧ w+4 < i ∧
  w+0 ∈ domain m ∧ val_rel x1 (m(w+0)) (m,b,i) ∧
  w+4 ∈ domain m ∧ val_rel x2 (m(w+4)) (m,b,i)
```

State relation

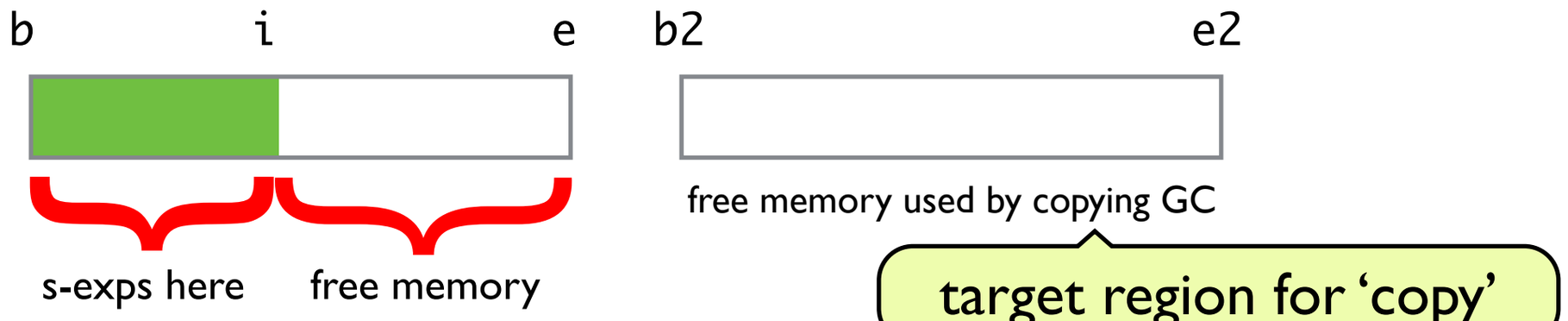
```
state_rel s t =
  s.clock = t.clock ∧
  (∀k val.
     lookup k s.store = Some val ⇒
   ∃w. lookup k t.store = Some w ∧
       val_rel w val (t.memory,t.b,t.i)) ∧
let s1 = { a | t.b ≤ a ∧ a < t.e } in
let s2 = { a | t.b2 ≤ a ∧ a < t.e2 } in
  domain m = s1 ∪ s2 ∧ s1 ∩ s2 = {} ∧
  t.b ≤ t.i ≤ t.e ∧ t.b2 ≤ t.e2 ∧
  every word_aligned [t.b,t.i,t.e,t.b2,t.e2]
```

for each value, there is a corresponding one below

details about memory layout

State relation

Memory is split into two heaps (only one active at a time) setup:




```
let s1 = { a | t.b ≤ a ∧ a < t.e } in
let s2 = { a | t.b2 ≤ a ∧ a < t.e2 } in
  domain m = s1 ∪ s2 ∧ s1 ∩ s2 = {} ∧
  t.b ≤ t.i ≤ t.e ∧ t.b2 ≤ t.e2 ∧
  every word_aligned [t.b,t.i,t.e,t.b2,t.e2]
```

details about memory layout

Lemmas about state relation

Operation in Lisp:

```
cdr (Dot x1 x2) = Some x2
cdr _ = None
```

Lemma for compiler proof:

```
cdr y = Some x ∧ val_rel y w (m,b,i) ⇒

w+4 ∈ domain m ∧ val_rel x (m(w+4)) (m,b,i) ∧
word_aligned (w+4)
```

loading an address corresponds to **cdr**

```
val_rel (Dot x1 x2) w (m,b,i) =
  word_aligned w ∧ b ≤ w ∧ w+4 < i ∧
  w+0 ∈ domain m ∧ val_rel x1 (m(w+0)) (m,b,i) ∧
  w+4 ∈ domain m ∧ val_rel x2 (m(w+4)) (m,b,i)
```

Operation in Lisp:

```
cdr (Dot x1 x2) = Some x2
cdr _ = None
```

Lemma for compiler proof:

```
cdr y = Some x ∧ val_rel y w (m,b,i) ⇒
w+4 ∈ domain m ∧ val_rel x (m(w+4)) (m,b,i) ∧
word_aligned (w+4)
```

loading an address corresponds to **cdr**

Implementation in compiler:

```
compile_exp (Cdr e) =
  MemOp Load (Add (compile_exp e) (Num 4))
```

Lemma for compiler proof:

```
cdr y = Some x ∧ val_rel y w (m,b,i) ⇒

w+4 ∈ domain m ∧ val_rel x (m(w+4)) (m,b,i) ∧
word_aligned (w+4)
```

loading an address corresponds to **cdr**

Lemmas about state relation

Operation in Lisp:

```
plus (Val w1) (Val w2) = Some (Val (w1 + w2))
plus _ _   = None
```

Lemma for compiler proof:

```
plus x y = Some res ∧
val_rel x w1 (m,b,i) ∧
val_rel y w2 (m,b,i) ⇒
val_rel res (w1 + w2 - 1) (m,b,i)
```

why do we subtract one?

Reminder about the definition:

```
val_rel (Val n) w (m,b,e) = (w = 4 × n + 1)
```

Operation in Lisp:

```
plus (Val w1) (Val w2) = Some (Val (w1 + w2))
plus _ _   = None
```

30-bit wrap-around semantics

Lemma for compiler proof:

```
plus x y = Some res ∧
val_rel x w1 (m,b,i) ∧
val_rel y w2 (m,b,i) ⇒
val_rel res (w1 + w2 - 1) (m,b,i)
```

why do we subtract one?

Lemmas about state relation

Operation in Lisp:

```
cons x1 x2 = Some (Dot x1 x2)
```

Question: *what's wrong with this lemma statement?*

```
cons x y = Some res ∧
val_rel x w1 (m,b,i) ∧
val_rel y w2 (m,b,i) ==>
val_rel res i (m[i ↦ w1, i+4 ↦ w2],b,i+8)
```

might wrap around

memory is updated

… might not be safe

Lemmas about state relation

Operation in Lisp:

```
cons x1 x2 = Some (Dot x1 x2)
```

assume there is space

A better one:

assume memory layout

```
cons x y = Some res ∧ state_rel s t ∧ t.i + 8 <= t.e ∧
val_rel x w1 (t.memory,t.b,t.i) ∧
val_rel y w2 (t.memory,t.b,t.i) ==>
val_rel res t.i (t.memory[i ↦ w1, i+4 ↦ w2],t.b,t.i+8) ∧
t.i ∈ domain m ∧ t.i + 4 ∈ domain m
```

memory is updated

Implementation of cons

Pseudo code:

fast and simple check

requires verified GC

if not (t.i + 8 ≤ t.e) then:

call alloc subroutine → run the copying GC

mem[t.i] = w1

mem[t.i+4] = w2

t.i := t.i + 8

if not (t.i + 8 ≤ t.e) then:

halt execution

return

Exercise: Sketch memory layout and cons-code for a mark-and-sweep garbage collector.

Question

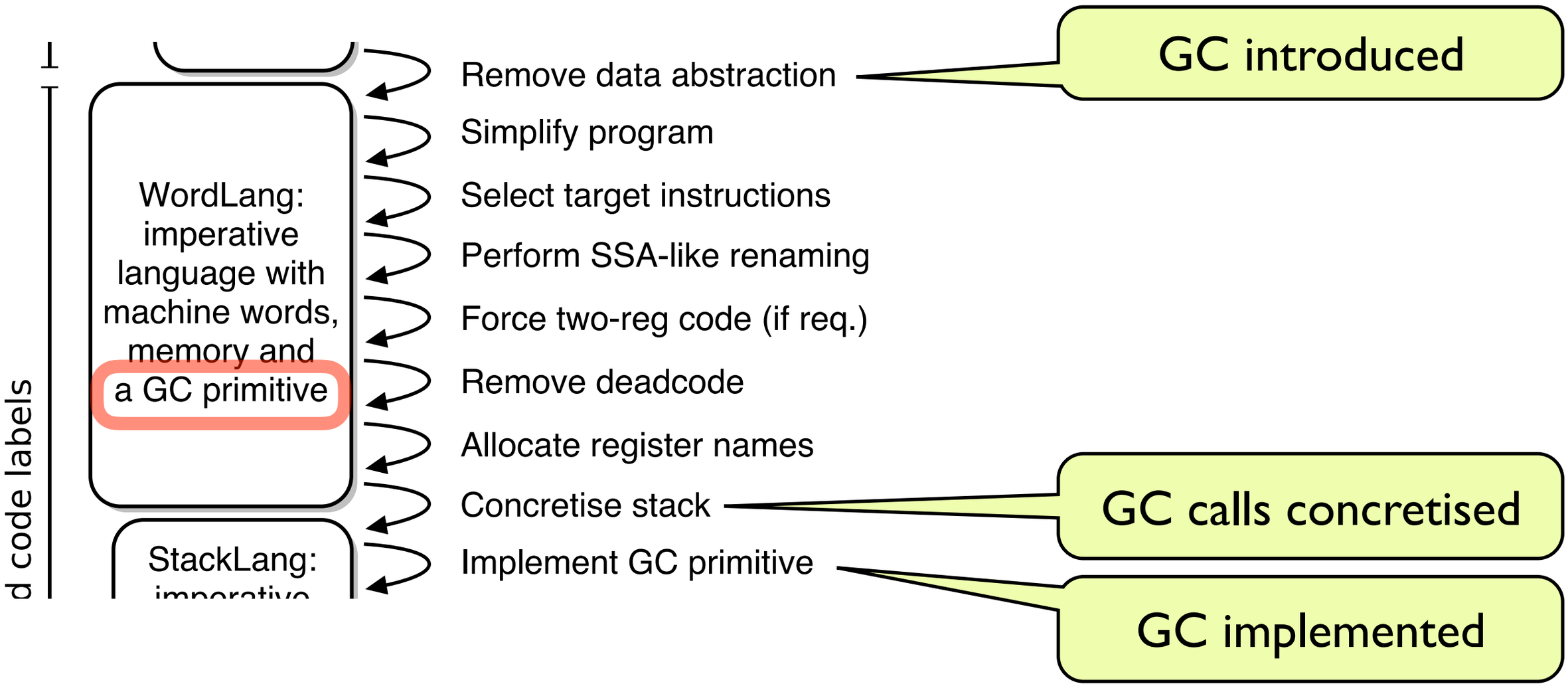
What do we need to prove
about the GC?

```
∀s t.  state_rel s t  ⇒  state_rel s (run_gc t)
```

running the GC …

… has no impact
on the abstract state.

Garbage collection in CakeML



Copying GC illustration

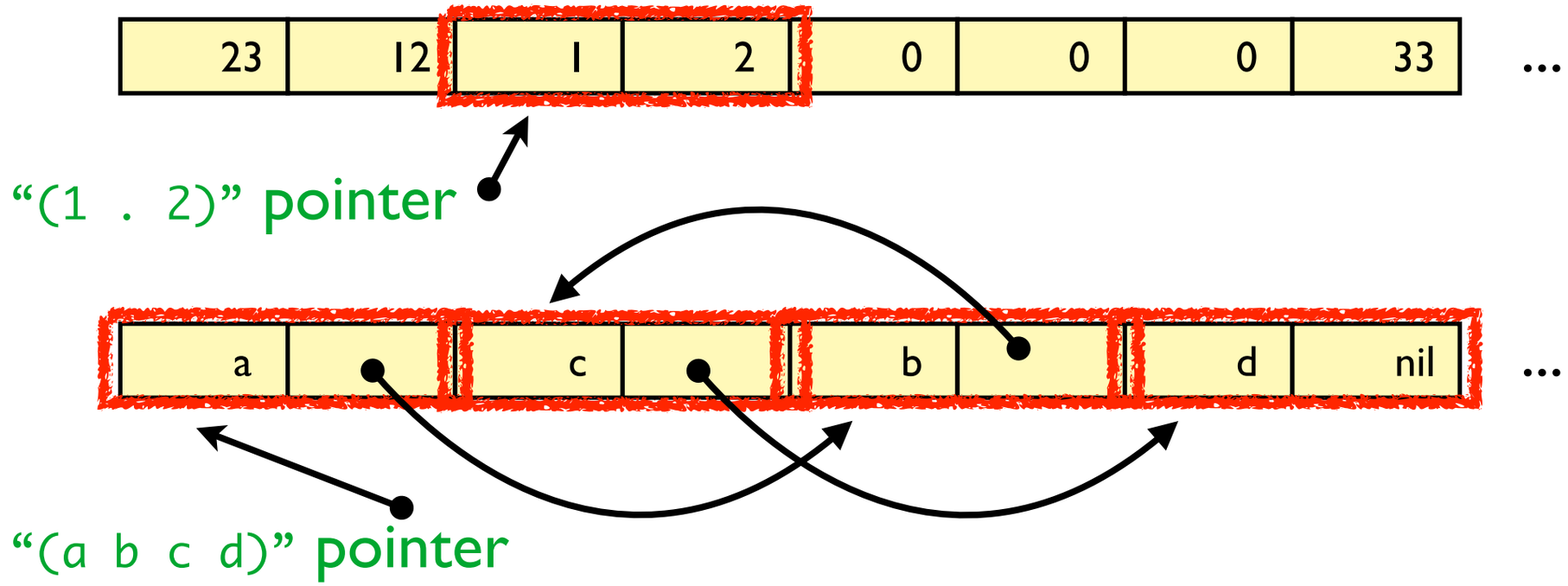
(a sketch on the blackboard)

Question

How do we model and verify the GC?

Memory as a graph

Representation in memory:



Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Step 1: specify what GC is to achieve

Step 2: write abstract implementation
(small-step relation), prove correct w.r.t spec

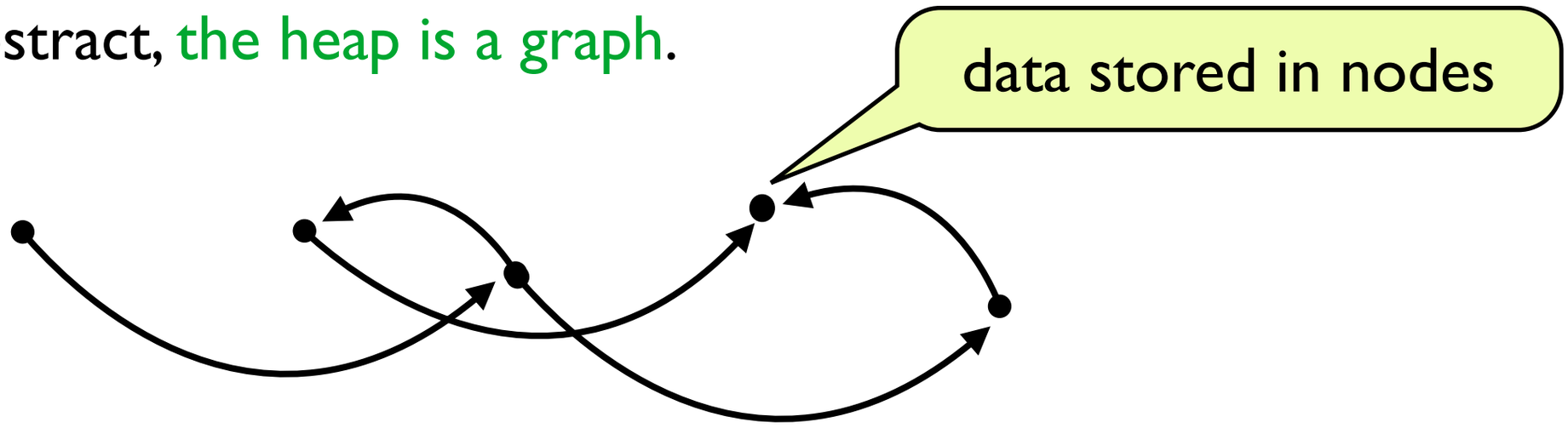
Step 3: introduce a more concrete notion of memory,
prove connection with small-step relation

Step 4: write and verify imperative style code with
concrete types

Specification of copying GC

How to model the 'heap' (i.e. memory) abstractly?

In the abstract, the heap is a graph.



We model the graph as a finite partial map from `num` to `heap_node`.

```
heap_addr ::= LHS num | RHS 'ptr_data
heap_node ::= (heap_addr list, 'data)
```

misaligned ptrs are data

at this level, type variable

State = heap graph + root pointers (active pointers in program).

Reachability

GC must not delete reachable nodes. Reachable:

$$\frac{a \in \text{set } roots}{a \in \text{reach } (h, roots)}$$

$$\frac{a \in \text{set } as \ \wedge \ h(b) = (as, data) \ \wedge \ b \in \text{reach } (h, roots)}{a \in \text{reach } (h, roots)}$$

A full GC ought to only keep reachable nodes:

$$\text{filter } (h, roots) \ = \ (h \lfloor (\text{reach } (h, roots)), roots)$$

restricts domain of h function to reach set

Moving

A moving GC is allowed to rename addresses:

We are allowed to apply a renaming function.

$$\text{domain}\ (\text{rename}\ f\ h) = \text{image}\ f\ (\text{domain}\ h)$$
$$(\text{rename}\ f\ h)(f(x)) = (\text{map}\ f\ as, d) \qquad \text{whenever}\ h(x) = (as, d)$$

Define:

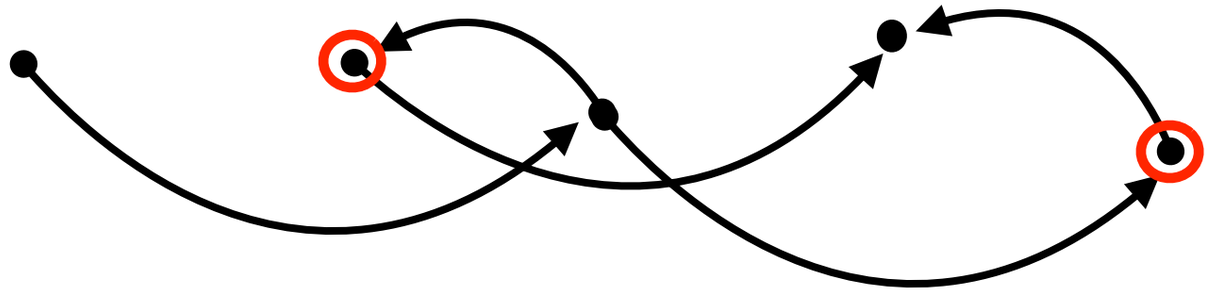
$$\frac{f \circ f = \text{id}}{(h, roots) \xrightarrow{\text{translate}} (\text{rename}\ f\ h, \text{map}\ f\ roots)}$$

The specification of a full moving GC:

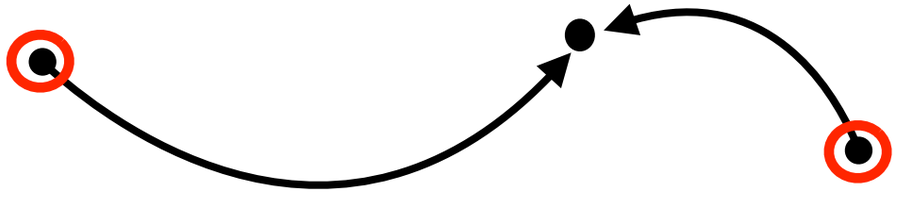
$$x \xrightarrow{\text{gc}} y \quad = \quad (\text{filter}\ x) \xrightarrow{\text{translate}} y$$

Example

Initial heap graph (with roots marked red):



After GC:



Abstract implementation

a needs to be moved

b is unused location

content at *a*

swap

Next: small step semantics for tri-colour moving GC algorithm

$$\frac{a \in z \ \wedge \ b \notin \text{domain } h \ \wedge \ f(a) = a \ \wedge \ f(b) = b \ \wedge \ h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\text{step}} (h[b \mapsto (as, d)] \restriction \{a\}^c, x, y \cup \{b\}, z \cup \text{set } as, f[a \mapsto b][b \mapsto a])}$$

already processed

finalise heap cell

$$\frac{a \in z \ \wedge \ f(a) \neq a}{(h, x, \ldots}$$

$$\frac{b \in y \ \wedge \ h(b) = (as, d) \ \wedge \ \text{set } as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\text{step}} (h[b \mapsto (\text{map } f \ as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

- h — the heap, a finite partial mapping,
- x — address set: completely processed heap elements,
- y — address set: moved elements with pointers to not-yet-moved elements,
- z — address set: elements that are still to be moved,
- f — a function which records where elements have been moved: $\mathbb{N} \rightarrow \mathbb{N}$

Correctness

starts off with roots to-be moved

terminates when nothing left to-do

Correctness theorem:

$$\forall h\ h_2\ roots\ x\ f.$$
$$(h, \{\}, \{\}, \mathsf{set}\ roots, \mathsf{id}) \xrightarrow{\mathsf{step}}{}^{*} (h_2, x, \{\}, \{\}, f)\ \wedge\ \mathsf{ok\_heap}\ (h, roots)\ \Longrightarrow$$
$$(h, roots) \xrightarrow{\mathsf{gc}} (h_2 \lfloor x, \mathsf{map}\ f\ roots)$$

where ok_heap $(h, roots)$ $=$ pointers $h \cup$ set $roots \subseteq$ domain h

pointers h $=$ $\{\ x \mid \exists a\ as\ d.\ x \in \mathsf{set}\ as \wedge h(a) = (as, d)\ \}$

any such terminating execution is a valid GC execution

Proof: we prove that an invariant is maintained

$$\forall x\ s\ t.\ \ \mathsf{inv}\ x\ s\ \wedge\ s \xrightarrow{\mathsf{step}} t\ \Longrightarrow\ \mathsf{inv}\ x\ t$$

and sufficient. Invariant on next slide...

Invariant

The lengthy invariant:

$$
\begin{array}{rl}
 & \mathsf{inv}\ (h_0, roots)\ (h, x, y, z, f) = \\
0 & \quad \mathsf{let}\ old = (\mathsf{domain}\ h \cup \{\ a \mid f(a) \neq a\ \}) - (x \cup y)\ \mathsf{in} \\
1 & \qquad (x \cap y = \{\}) \wedge (f \circ f = \mathsf{id}) \wedge \\
2 & \qquad \mathsf{pointers}\ (h \lfloor x) \subseteq x \cup y \wedge \\
3 & \qquad \mathsf{pointers}\ (h \lfloor x^c) \subseteq old \wedge \\
4 & \qquad \mathsf{pointers}\ (h \lfloor y) \cup \mathsf{set}\ roots \subseteq \mathsf{image}\ f\ (x \cup y) \cup z \subseteq \mathsf{reach}\ (h_0, roots) \wedge \\
5 & \qquad (\forall a.\ a \in z \implies \mathsf{if}\ f(a) = a\ \mathsf{then}\ a \in old\ \mathsf{else}\ f(a) \in x \cup y) \wedge \\
6 & \qquad (\forall a.\ f(a) \neq a \implies \neg(a \in x \cup y \iff f(a) \in x \cup y)) \wedge \\
7 & \qquad (\forall a.\ a \in x \cup y \iff f(a) \neq a \wedge a \in \mathsf{domain}\ h) \wedge \\
8 & \qquad \mathsf{domain}\ h = \mathsf{image}\ f\ (\mathsf{domain}\ h_0) \wedge \\
9 & \qquad (\forall a\ as\ d.\ f(a) \in \mathsf{domain}\ h \wedge h(f(a)) = (as, d) \implies \\
 & \qquad\qquad h_0(a) = \mathsf{if}\ f(a) \in x\ \mathsf{then}\ (\mathsf{map}\ f\ as, d)\ \mathsf{else}\ (as, d))
\end{array}
$$

Most effort is spent finding the invariant.

First-order prover can automate much of the proof.

Implementation with memory

Next refinement introduces an abstract memory.

Memory consists of

Block (as, l, d)	—	block of data, e.g. a cons-cell
Ref a	—	record of where data has moved
Emp	—	empty or 'don't care'

Relation to small-step relation's state:

$$
\begin{array}{ll}
m(a) = \text{Block } (h(a)) & \text{if } a \in \text{domain } h \\
m(a) = \text{Ref } (f(a)) & \text{if } a \notin \text{domain } h \text{ and } f(a) \neq a \\
m(a) = \text{Emp} & \text{if } a \notin \text{domain } h \text{ and } f(a) = a
\end{array}
$$

Implementation with memory

$$
\begin{aligned}
\text{move } (\text{RHS } n, j, m) &= (\text{RHS } n, j, m) \\
\text{move } (\text{LHS } a, j, m) &= \text{case } m(a) \text{ of} \\
&\qquad \text{Ref } i \rightarrow (\text{LHS } i, j, m) \\
&\quad \mid \text{Block } (as, l, d) \rightarrow \\
&\quad \text{let } m = m[a \mapsto \text{Ref } j] \text{ in} \\
&\quad \text{let } m = m[j \mapsto \text{Block } (as, l, d)] \text{ in} \\
&\qquad (\text{LHS } j, j + l + 1, m)
\end{aligned}
$$

Implementation with memory

$$
\begin{array}{l}
\mathsf{move}\ (\mathsf{RHS}\ n, j, m) = (\mathsf{RHS}\ n, j, m) \\
\mathsf{move}\ (\mathsf{LHS}\ a, j, m)\ = \mathsf{case}\ m(a)\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\quad \mathsf{Ref}\ i \rightarrow (\mathsf{LHS}\ i, j, m) \\
\qquad\qquad\qquad\qquad \mid \mathsf{Block}\ (as, l, d) \rightarrow \\
\qquad\qquad\qquad\qquad\quad\ \mathsf{let}\ m = m[a \mapsto \mathsf{Ref}\ j]\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\quad\ \mathsf{let}\ m = m[j \mapsto \mathsf{Block}\ (as, l, d)]\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{LHS}\ j, j + l + 1, m)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{move\_list}\ ([], j, m) = ([], j, m) \\
\mathsf{move\_list}\ (r{::}rs, j, m)\ = \\
\quad \mathsf{let}\ (r, j, m) = \mathsf{move}\ (r, j, m)\ \mathsf{in} \\
\quad \mathsf{let}\ (rs, j, m) = \mathsf{move\_list}\ (rs, j, m)\ \mathsf{in} \\
\qquad (r{::}rs, j, m)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{readBlock}\ (\mathsf{Block}\ x) = x \\
\mathsf{cut}\ (i, j)\ m\ =\ \lambda k.\ \ \mathsf{if}\ i \leq k \wedge k < j\ \mathsf{then}\ m\, k\ \mathsf{else}\ \mathsf{Emp}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{loop}\ (i, j, m)\ = \\
\quad \mathsf{if}\ i = j\ \mathsf{then}\ (i, m)\ \mathsf{else} \\
\qquad \mathsf{let}\ (as, l, d) = \mathsf{readBlock}\ (m\ i)\ \mathsf{in} \\
\qquad \mathsf{let}\ (as, j, m) = \mathsf{move\_list}\ (as, j, m)\ \mathsf{in} \\
\qquad \mathsf{let}\ m = m[i \mapsto \mathsf{Block}\ (as, l, d)]\ \mathsf{in} \\
\qquad\quad \mathsf{loop}\ (i + l + 1, j, m)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{collector}\ (roots, b, i, e, b_2, e_2, m)\ = \\
\quad \mathsf{let}\ (b_2, e_2, b, e) = (b, e, b_2, e_2)\ \mathsf{in} \\
\quad \mathsf{let}\ (roots, j, m) = \mathsf{move\_list}\ (roots, b, m)\ \mathsf{in} \\
\quad \mathsf{let}\ (i, m) = \mathsf{loop}\ (b, j, m)\ \mathsf{in} \\
\quad \mathsf{let}\ m = \mathsf{cut}\ (b, i)\ m\ \mathsf{in} \\
\qquad (roots, b, i, e, b_2, e_2, m)
\end{array}
$$

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$$\forall h\ roots\ roots_2\ x\ y.$$
$$\text{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \text{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow$$
$$\exists h_2.\ \text{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\text{gc}} (h_2, roots_2)$$

some new abstract heap exists such that ...

specification is met

Proof: again uses a lengthy invariant

$$\text{mem\_inv}\ (h_0, roots_0, h, f)\ (b, i, j, e, b_2, e_2, m, z) =$$
$$b \le i \le j \le e \wedge (e < b_2 \vee e_2 < b)\ \wedge$$
$$(\forall a.\ a \notin b_2 ... e_2 \cup b ... j\ \Longrightarrow\ m(a) = \text{Emp})\ \wedge$$
$$\text{part\_heap}\ (b, i)\ m\ (i - b) \wedge \text{part\_heap}\ (i, j)\ m$$
$$(\exists k.\ \text{part\_heap}\ (b_2, e_2)\ m\ k \wedge k \le e - j)\ \wedge$$
$$\text{ref\_mem}\ (h, f)\ m \wedge \text{ok\_heap}\ (h_0, roots_0)\ \wedge$$
$$(h_0, \{\}, \{\}, \text{set}\ roots_0, \text{id}) \xrightarrow{\text{step}} *\ (h, \text{domain}\ h \cap (b ... i), \text{domain}\ h \cap (i ... j), z, f)$$

most is trivial book-keeping (where/how state is repr.)

reason for correctness inherited

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

Carefully written code for other architectures (x86, PowerPC etc.) decompiles to the same function in logic. Proof reuse!

Decompilation produces functions, e.g.

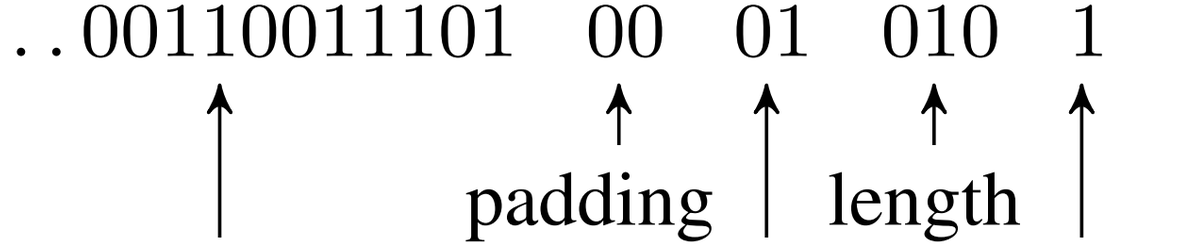
$$
\begin{array}{l}
\text{mc\_move\_loop}\ (r_2, r_3, r_4, g) = \\
\quad \text{if}\ r_4 = 0\ \text{then}\ (r_2, r_3, r_4, g)\ \text{else} \\
\quad\quad \text{let}\ r_5 = g(r_2)\ \text{in} \\
\quad\quad \text{let}\ r_4 = r_4 - 1\ \text{in} \\
\quad\quad \text{let}\ r_2 = r_2 + 4\ \text{in} \\
\quad\quad \text{let}\ g = g[r_3 \mapsto r_5]\ \text{in} \\
\quad\quad \text{let}\ r_3 = r_3 + 4\ \text{in} \\
\quad\quad\quad \text{mc\_move\_loop}\ (r_2, r_3, r_4, g)
\end{array}
$$

one proves this implements abstract version

A glimpse at CakeML's pointers

Configurable data representation:

Example pointer value:



These can be left out

Speeds up pattern matching, if present

```
v =
  Number int
| Word64 word64
| Block num (v list)
| CodePtr num
| RefPtr num
```

"the tag"

What we learnt

ML compilers cannot preserve semantics completely:

```
machine_semantics (compile prog)
  ∈ extend_with_resource_limit (semantics prog)
```

We add get-out clause to simulation theorems:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2 res2. evaluate (compile code) t1 = (res2,t2) ∧
          (state_rel s2 t2 ∧ res2 = res ∨
           res2 = RHitHardLimit ∧ t2.IO isPrefix s2.IO)
```

A verified GC is needed.

Remember: separate complex algorithm proofs from implementation proofs (to keep sane and enable reuse)

```
machine_semantics (compile prog)
  ∈ extend_with_resource_limit (semantics prog)
```

Extra: discussion about top-level theorem