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)]

\[
\text{evaluate } \text{code } s1 = (res,s2) \land res \neq \text{Rfail} \land \\
\text{state\_rel } s1 \ t1 \Rightarrow \\
\exists t2. \ \text{evaluate } (\text{compile } \text{code}) \ t1 = (res,t2) \land \\
\text{state\_rel } s2 \ t2
\]
Last lecture (cont.)
Sketched verification of a toy imperative compiler.

Source language:

- \( t = \text{Dec} \text{ string} \ t \)
  - \( \text{Exp} \ e \)
  - \( \text{Break} \)
  - \( \text{Seq} \ t \ t \)
  - \( \text{If} \ e \ t \ t \)
  - \( \text{For} \ e \ e \ t \)

- \( e = \text{Var} \text{ string} \)
  - \( \text{Num} \text{ int} \)
  - \( \text{Add} \ e \ e \)
  - \( \text{Assign} \text{ string} \ e \)

Target language:

- \( \text{inst} = \text{Add} \text{ reg} \text{ reg} \text{ reg} \text{ reg} \)
  - \( \text{Int} \text{ reg} \text{ int} \)
  - \( \text{Jmp} \text{ offset} \)
  - \( \text{JmpIf} \text{ reg} \text{ offset} \)
We can make it less toy.

Source language:

\[
\begin{align*}
t &= \text{Dec}\ string\ t \\
&| \quad \text{Exp}\ e \\
&| \quad \text{Break} \\
&| \quad \text{Seq}\ t\ t \\
&| \quad \text{If}\ e\ t\ t \\
&| \quad \text{For}\ e\ e\ t \\
e &= \text{Var}\ string \\
&| \quad \text{Num}\ \text{word32} \\
&| \quad \text{Add}\ e\ e \\
&| \quad \text{Assign}\ string\ e \\
&| \quad \text{MemOp} \ ... \\
\end{align*}
\]

Target language:

\[
\begin{align*}
\text{inst} &= \text{Add}\ reg\ reg\ reg\ reg \\
&| \quad \text{Int}\ reg\ \text{word32} \\
&| \quad \text{Jmp}\ offset \\
&| \quad \text{JmpIf}\ reg\ offset \\
&| \quad \text{Load}\ reg\ reg \\
&| \quad \text{Store}\ reg\ reg \\
\end{align*}
\]

We can use machine types and have a memory.

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

Sketched verification of a toy imperative compiler.
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

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:**

\[
v = \begin{array}{l}
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{CodePtr num} \\
\text{RefPtr num}
\end{array}
\]

- mathematical integers
- tuples / vectors of any length

**Value types after:**

- machine words (32- or 64-bit)
- state contains a memory modelled as finite map

No function values. No size limits.

Implementing the ML abstractions

**Value type before:**

\[ v = \]

- Number int
- Word64 word64
- Block num (v list)
- CodePtr num
- RefPtr num

**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:

\[
V = \\
\quad \text{Number int} \\
\quad \text{Word64 word64} \\
\quad \text{Block num (v list)} \\
\quad \text{CodePtr num} \\
\quad \text{RefPtr num}
\]

Value types after:

- machine words (32- or 64-bit)
- state contains a memory modelled as finite map

mathematical integers

This refinement is complicated.
It cannot be perfect because the target has fixed size limits.
In Lecture 1, we proved theorems of the form:

\[
\text{evaluate \ code } s1 = (res,s2) \land res \neq \text{Rfail} \land \\
\text{state\_rel } s1 \ t1 \Rightarrow \\
\exists t2. \text{ evaluate (compile code) } t1 = (res,t2) \land \\
\text{state\_rel } s2 \ t2
\]
The tools from Lecture 1 adapted

Modified:

\[
\text{evaluate code } s1 = (\text{res}, s2) \land \text{res} \neq \text{Rfail} \land \\
\text{state_rel } s1 \ t1 \Rightarrow \\
\exists t2 \text{ res2}. \ \text{evaluate (compile code)} \ t1 = (\text{res2}, t2) \land \\
(\text{state_rel } s2 \ t2 \land \text{res2} = \text{res} \lor \\
\text{res2} = \text{RHItHardLimit} \land t2.\text{IO isPrefix s2.IO})
\]

- 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:

\[
\text{evaluate code } s_1 = (res, s_2) \land res \neq \text{Rfail} \land \\
\text{state_rel } s_1 \ t_1 \Rightarrow \\
\exists t_2 \ res_2. \text{ evaluate } (\text{compile code}) \ t_1 = (res_2, t_2) \land \\
(\text{state_rel } s_2 \ t_2 \land res_2 = res \lor \\
res_2 = \text{RHitHardLimit} \land t_2.\text{IO} \text{ isPrefix } s_2.\text{IO})
\]

The top-level correctness theorem becomes weaker:

\[
\text{machine_semantics } (\text{compile } \text{prog}) \\
\in \text{extend_with_resource_limit } (\text{semantics } \text{prog})
\]

produce a set of all prefixes ... ... of these behaviours
The tools from Lecture 1 adapted

In our simple setting \textit{without} I/O:

\begin{align*}
\text{extend\_with\_resource\_limit} \text{ Crash} &= \{ \text{ Crash } \} \\
\text{extend\_with\_resource\_limit} \text{ Terminate} &= \{ \text{ Terminate } \} \\
\text{extend\_with\_resource\_limit} \text{ Diverge} &= \{ \text{ Diverge, Terminate } \}
\end{align*}

\textbf{Question: Is this too weak?}

The top-level correctness theorem becomes weaker:

\[
\text{machine\_semantics} \left( \text{compile} \ prog \right) \ni \text{extend\_with\_resource\_limit} \left( \text{semantics} \ prog \right)
\]

produce a set of all prefixes … … of these behaviours
The tools from Lecture 1 adapted

Modified:

\[
\text{evaluate code } s1 = (\text{res}, s2) \land \text{res} \neq \text{Rfail} \land \\
\text{state_rel } s1 \ t1 \Rightarrow \\
\exists t2 \ \text{res2. evaluate (compile code) } t1 = (\text{res2}, t2) \land \\
(\text{state_rel } s2 \ t2 \land \text{res2} = \text{res} \lor \\
\text{res2} = \text{RHitHardLimit} \land t2.IO \text{ isPrefix } s2.I0)
\]

The top-level correctness theorem becomes weaker:

\[
\text{machine_semantics (compile } \text{prog) } \\
\in \text{extend_with_resource_limit (semantics } \text{prog)}
\]
State relation defines abstraction

CakeML’s abs. complicated

Value type before:

\[
\begin{align*}
v &= \\
& \quad \text{Number int} \\
& \quad | \text{Word64 word64} \\
& \quad | \text{Block num (v list)} \\
& \quad | \text{CodePtr num} \\
& \quad | \text{RefPtr num}
\end{align*}
\]

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:

\[
\begin{align*}
\text{sexp} &= \\
& \quad \text{Num word30} \\
& \quad | \text{Sym word30} \\
& \quad | \text{Dot sexp sexp}
\end{align*}
\]

Value types after: simplified Lisp s-expression
state contains a memory modelled as finite map

fixed & simple
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 \( 4 \times v + 1 \) (ends in bits ‘01’)
- **symbol** value \( s \) represented as word \( 4 \times s + 2 \) (ends in bits ‘10’)

**Example:** \((1 \ 2)\) i.e. \((1 \ . \ (2 \ . \ nil))\)

Representation: 200

With memory:

<table>
<thead>
<tr>
<th>word: 5</th>
<th>word: 400</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>word: 9</td>
<td>word: 2</td>
<td>...</td>
</tr>
</tbody>
</table>

address 200

address 400

‘nil’ symbol number 0
Value relation

\[
\text{val\_rel (Val n) w (m,b,i)} = (w = 4 \times n + 1)
\]

\[
\text{val\_rel (Sym n) w (m,b,i)} = (w = 4 \times n + 2)
\]

\[
\text{val\_rel (Dot x1 x2) w (m,b,i)} = \\
\text{word\_aligned w} \land b \leq w \land w+4 < i \land \\
w+0 \in \text{domain m} \land \text{val\_rel x1 (m(w+0))} (m,b,i) \land \\
w+4 \in \text{domain m} \land \text{val\_rel x2 (m(w+4))} (m,b,i)
\]
State relation

\[
\text{state\_rel } s \ t = \\
\quad s.\text{clock} = t.\text{clock} \land \\
\quad (\forall k \ \text{val.} \\
\quad \quad \text{lookup } k \ s.\text{store} = \text{Some } \text{val} \Rightarrow \\
\quad \quad \exists w. \ \text{lookup } k \ t.\text{store} = \text{Some } w \land \\
\quad \quad \quad \text{val\_rel } w \ \text{val} (t.\text{memory},t.b,t.i) ) \land \\
\quad \text{let } s_1 = \{ a \mid t.b \leq a \land a < t.e \} \ \text{in} \\
\quad \text{let } s_2 = \{ a \mid t.b2 \leq a \land a < t.e2 \} \ \text{in} \\
\quad \quad \text{domain } m = s_1 \cup s_2 \land s_1 \cap s_2 = \{\} \land \\
\quad \quad t.b \leq t.i \leq t.e \land t.b2 \leq t.e2 \land \\
\quad \quad \text{every word\_aligned } [t.b,t.i,t.e,t.b2,t.e2]
\]
State relation

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

\[
\begin{align*}
&\text{s-exps here} & \text{free memory} \\
\end{align*}
\]

\[
\begin{align*}
\text{free memory used by copying GC} \\
\text{target region for ‘copy’}
\end{align*}
\]

\[
\begin{align*}
&b \quad i \quad e \quad b2 \quad e2 \\
&\text{let } s1 = \{ a \mid t.b \leq a \land a < t.e \} \text{ in} \\
&\text{let } s2 = \{ a \mid t.b2 \leq a \land a < t.e2 \} \text{ in} \\
&\text{domain } m = s1 \cup s2 \land s1 \cap s2 = \{\} \land \\
&t.b \leq t.i \leq t.e \land t.b2 \leq t.e2 \land \\
&\text{every word_aligned} \ [t.b,t.i,t.e,t.b2,t.e2]
\end{align*}
\]
Lemmas about state relation

**Operation in Lisp:**

\[
\text{cdr } (\text{Dot } x_1 \ x_2) = \text{Some } x_2 \\
\text{cdr } _\ = \text{None}
\]

**Lemma for compiler proof:**

\[
\text{cdr } y = \text{Some } x \land \text{val_rel } y \ w (m,b,i) \Rightarrow \\
w+4 \in \text{domain } m \land \text{val_rel } x (m(w+4)) (m,b,i) \land \text{word_aligned} (w+4)
\]

loading an address corresponds to \textbf{cdr}
val_rel (Dot x1 x2) w (m,b,i) =
word_aligned w \land b \leq w \land w+4 < i \land
w+0 \in \text{domain } m \land \text{val_rel } x1 (m(w+0)) (m,b,i) \land
w+4 \in \text{domain } m \land \text{val_rel } x2 (m(w+4)) (m,b,i)

\textbf{Operation in Lisp:}

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

\textbf{Lemma for compiler proof:}

cdr y = Some x \land \text{val_rel } y w (m,b,i) \Rightarrow
w+4 \in \text{domain } m \land \text{val_rel } x (m(w+4)) (m,b,i) \land
\text{word_aligned } (w+4)
Implementation in compiler:

\[
\text{compile\_exp (Cdr e)} = \text{MemOp Load (Add (compile\_exp e) (Num 4))}
\]

Lemma for compiler proof:

\[
\text{cdr y} = \text{Some x} \land \text{val\_rel y w (m,b,i)} \Rightarrow \\
\text{w+4} \in \text{domain m} \land \text{val\_rel x (m(w+4)) (m,b,i)} \land \\
\text{word\_aligned (w+4)}
\]

loading an address corresponds to cdr
Lemmas about state relation

Operation in Lisp:

\[
\text{plus (Val } w_1) (\text{Val } w_2) = \text{Some (Val } (w_1 + w_2)) \\
\text{plus } \_ \_ \_ = \text{None}
\]

Lemma for compiler proof:

\[
\text{plus } x \; y = \text{Some res } \wedge \\
\text{val}_\text{rel} \; x \; w_1 \; (m,b,i) \wedge \\
\text{val}_\text{rel} \; y \; w_2 \; (m,b,i) \Rightarrow \\
\text{val}_\text{rel} \; \text{res } (w_1 + w_2 - 1) \; (m,b,i)
\]

why do we subtract one?
Reminder about the definition:

\[
\text{val\_rel (Val } n) \ w \ (m,b,e) = (w = 4 \times n + 1)
\]

Operation in Lisp:

\[
\text{plus (Val } w1) (Val w2) = \text{Some (Val } (w1 + w2))
\]
\[
\text{plus } \_ \_ \_ = \text{None}
\]

Lemma for compiler proof:

\[
\text{plus } x \ y = \text{Some } \text{res } \wedge
\]
\[
\text{val\_rel } x \ w1 \ (m,b,i) \wedge
\]
\[
\text{val\_rel } y \ w2 \ (m,b,i) \Rightarrow
\]
\[
\text{val\_rel } \text{res } (w1 + w2 - 1) \ (m,b,i)
\]

30-bit wrap-around semantics

why do we subtract one?
Lemmas about state relation

**Operation in Lisp:**

\[
\text{cons } x_1 \ x_2 = \text{Some } (\text{Dot } x_1 \ x_2)
\]

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

\[
\text{cons } x \ y = \text{Some } \text{res } \land \\
\text{val_rel } x \ w_1 (m,b,i) \land \\
\text{val_rel } y \ w_2 (m,b,i) \Rightarrow \\
\text{val_rel } \text{res } i \ (m[i \mapsto w_1, \ i+4 \mapsto w_2],b,i+8)
\]

*might wrap around*

*memory is updated*

*... might not be safe*
Lemmas about state relation

**Operation in Lisp:**

\[
\text{cons } x_1 \ x_2 = \text{Some} \ (\text{Dot} \ x_1 \ x_2)
\]

**A better one:**

\[
\begin{align*}
\text{cons } x \ y &= \text{Some} \ \text{res} \land \text{state} \ _\text{rel} \ s \ t \land \ t.i + 8 \leq t.e \\
\text{val} \ _\text{rel} \ x \ w_1 \ (t.\text{memory}, t.b, t.i) \land \text{val} \ _\text{rel} \ y \ w_2 \ (t.\text{memory}, t.b, t.i) \Rightarrow \\
\text{val} \ _\text{rel} \ \text{res} \ t.i \ (t.\text{memory}[i \rightarrow w_1, i+4 \rightarrow w_2], t.b, t.i+8) \land \ \\
t.i \in \text{domain} \ m \land t.i + 4 \in \text{domain} \ m
\end{align*}
\]
Implementation of cons

**Pseudo code:**

if not (t.i + 8 ≤ t.e) then:

    call alloc subroutine

    mem[t.i] = w1
    mem[t.i+4] = w2
    t.i := t.i + 8

run the copying GC

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. \textit{state}_rel s t \Rightarrow \textit{state}_rel s (\textit{run}_gc t)

running the GC ...

... has no impact on the abstract state.
Garbage collection in CakeML

- **GC introduced**
  - Remove data abstraction
  - Simplify program
  - Select target instructions
  - Perform SSA-like renaming
  - Force two-reg code (if req.)
  - Remove deadcode
  - Remove data abstraction
  - Simplify program
  - Select target instructions
  - Perform SSA-like renaming
  - Force two-reg code (if req.)
  - Remove deadcode
  - Concretise stack
  - Implement GC primitive

- **GC calls concretised**
  - Concretise stack
  - Implement GC primitive

- **GC implemented**
  - Concretise stack
  - Implement GC primitive
Copying GC illustration
(a sketch on the blackboard)

Question
How do we model and verify the GC?
Memory as a graph

Representation in memory:

```
23 12 1 2 0 0 0 0 33 ...
```

"(1 . 2)" pointer

```
a c b d nil ...
```

"(a b c d)" pointer
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.

\[
\text{heap_addr ::= LHS num | RHS 'ptr_data} \\
\text{heap_node ::= (heap_addr list, 'data)}
\]

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

**GC must not** delete reachable nodes. Reachable:

\[
\begin{align*}
\text{set roots} & \rightarrow \text{reach (h, roots)} \\
\text{set as, data, reach (h, roots)} & \rightarrow \text{filter (h, roots)}
\end{align*}
\]

**A full GC** ought to only keep reachable nodes:

\[
\text{filter (h, roots)} = (h \upharpoonright (\text{reach (h, roots)}), \text{roots})
\]
Moving

A moving GC is allowed to rename addresses:

We are allowed to apply a renaming function.

\[
\text{domain (rename } f \ h) = \text{image } f \ (\text{domain } h) \\
\text{(rename } f \ h)(f(x)) = (\text{map } f \ as, d) \quad \text{whenever } h(x) = (as, d)
\]

Define:

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

The specification of a full moving GC:

\[
x \xrightarrow{\text{gc}} y = (\text{filter } x) \xrightarrow{\text{translate}} y
\]
Example

Initial heap graph (with roots marked red):

After GC:
Once all of the addresses have been part of earlier move operations, we define a valid rearrangement as a relation
\[ a \in z \land b \notin \text{domain } h \land f(a) = a \land f(b) = b \land h(a) = (as, d) \]
\[ (h, x, y, z, f) \xrightarrow{\text{step}} (h[b \mapsto (as, d)] \cap \{a\}, x, y \cup \{b\}, z \cup \text{set } as, f[a \mapsto b][b \mapsto a]) \]

Next: small-step implementation of a multi-colour moving algorithm

\[ a \notin z \land f(a) \neq a \]
\[ (h, x, y, z, f) \xrightarrow{\text{step}} (h[b \mapsto (as, d)], x \cup \{b\}, y \setminus \{b\}, z, f) \]

\[ b \in y \land h(b) = (as, d) \land \text{set } as \cap z = \{\} \]

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

Correctness theorem:

\[ \forall h \ h_2 \ roots \ x \ f. \]
\[ (h, \{\}, \{\}, set \ roots, \ id) \xrightarrow{\text{step}^*} (h_2, x, \{\}, \{\}, f) \land \text{ok_heap} (h, roots) \implies \]
\[ (h, roots) \xrightarrow{\text{gc}} (h_2 \setminus x, \text{map } f \ roots) \]

where \( \text{ok_heap} (h, roots) = \) pointers \( h \cup \text{set } roots \subseteq \text{domain } h \)

pointers \( h = \{ x | \exists a \ as \ d. \ x \in \text{set } as \land h(a) = (as, d) \} \)

terminates when nothing left to-do

any such terminating execution is a valid GC execution

Proof: we prove that an invariant is maintained

\[ \forall x \ s \ t. \ \text{inv } x \ s \land s \xrightarrow{\text{step}} t \implies \text{inv } x \ t \]

and sufficient. Invariant on next slide...
Invariant

The lengthy invariant:

\[
\text{inv } (h_0, \text{roots}) (h, x, y, z, f) = \\
0 \quad \text{let } old = (\text{domain } h \cup \{ a \mid f(a) \neq a \}) - (x \cup y) \text{ in} \\
1 \quad (x \cap y = \{ \}) \land (f \circ f = \text{id}) \land \\
2 \quad \text{pointers } (h|x) \subseteq x \cup y \land \\
3 \quad \text{pointers } (h|x^c) \subseteq old \land \\
4 \quad \text{pointers } (h|y) \cup \text{set roots} \subseteq \text{image } f (x \cup y) \cup z \subseteq \text{reach } (h_0, \text{roots}) \land \\
5 \quad (\forall a. a \in z \implies \text{if } f(a) = a \text{ then } a \in old \text{ else } f(a) \in x \cup y) \land \\
6 \quad (\forall a. f(a) \neq a \implies \neg(a \in x \cup y \iff f(a) \in x \cup y)) \land \\
7 \quad (\forall a. a \in x \cup y \iff f(a) \neq a \land a \in \text{domain } h) \land \\
8 \quad \text{domain } h = \text{image } f (\text{domain } h_0) \land \\
9 \quad (\forall a \text{ as } d. f(a) \in \text{domain } h \land h(f(a)) = (as, d) \implies \\
\hspace{1cm} h_0(a) = \text{if } f(a) \in x \text{ then } (\text{map } f \text{ as, } d) \text{ else } (as, d))
\]

Most effort is spent finding the invariant. 
First-order prover can automate much of the proof.
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:

\[
\begin{align*}
m(a) &= \text{Block } (h(a)) & \text{if } a \in \text{domain } h \\
m(a) &= \text{Ref } (f(a)) & \text{if } a \not\in \text{domain } h \text{ and } f(a) \neq a \\
m(a) &= \text{Emp} & \text{if } a \not\in \text{domain } h \text{ and } f(a) = a
\end{align*}
\]
move \((\text{RHS} \ n, j, m) = (\text{RHS} \ n, j, m)\)

move \((\text{LHS} \ a, j, m) = \text{case} \ m(a) \ of\)

- \(\text{Ref} \ i \rightarrow (\text{LHS} \ i, j, m)\)
- \(\text{Block} \ (as, l, d) \rightarrow\)
  - let \(m = m[a \mapsto \text{Ref} \ j]\) in
  - let \(m = m[j \mapsto \text{Block} \ (as, l, d)]\) in
  - \((\text{LHS} \ j, j + l + 1, m)\)
Implementation with memory

move (RHS \( n, j, m \)) = (RHS \( n, j, m \))
move (LHS \( a, j, m \)) = case \( m(a) \) of
    Ref \( i \to (\text{LHS} \; i, j, m) \)
    | Block (\( as, l, d \)) \to
        let \( m = m[a \to \text{Ref} \; j] \) in
        let \( m = m[j \leftrightarrow \text{Block} \; (as, l, d)] \) in
        (LHS \; j, j + l + 1, m)

move_list (\([], j, m\)) = (\([], j, m\))
move_list (\(r::rs, j, m\)) =
    let \( (r, j, m) = \text{move} \; (r, j, m) \) in
    let \( (rs, j, m) = \text{move_list} \; (rs, j, m) \) in
    (r::rs, j, m)

readBlock (Block \( x \)) = x

cut \( (i, j) \; m = \lambda k. \; \text{if } i \leq k \land k < j \; \text{then } m \; k \; \text{else Emp} \)

loop \( (i, j, m) = \)
    if \( i = j \) then \( (i, m) \) else
        let \( (as, l, d) = \text{readBlock} \; (m \; i) \) in
        let \( (as, j, m) = \text{move_list} \; (as, j, m) \) in
        let \( m = m[i \leftrightarrow \text{Block} \; (as, l, d)] \) in
        loop \( (i + l + 1, j, m) \)

collector \( (roots, b, i, e, b_2, e_2, m) = \)
    let \( (b_2, e_2, b, e) = (b, e, b_2, e_2) \) in
    let \( (roots, j, m) = \text{move_list} \; (roots, b, m) \) in
    let \( (i, m) = \text{loop} \; (b, j, m) \) in
    let \( m = \text{cut} \; (b, i) \; m \) in
    (roots, b, i, e, b_2, e_2, m)
Correctness

if abstract state is correctly represented, then ...

Correctness theorem

\[ \forall h \text{ roots roots}_2 x y. \]
\[ \text{ok\_mem\_heap (h, roots)} x \land \text{collector (roots,} x) = (\text{roots}_2, y) \implies \]
\[ \exists h_2. \text{ok\_mem\_heap (h}_2, \text{roots}_2) y \land (h, \text{roots}) \xrightarrow{gc} (h_2, \text{roots}_2) \]

some new abstract heap exists such that ...

specification is met

Proof: again uses a lengthy invariant

\[ \text{mem\_inv (h}_0, \text{roots}_0, h, f) (b, i, j, e, b_2, e_2, m, z) = \]
\[ b \leq i \leq j \leq e \land (e < b_2 \lor e_2 < b) \land \]
\[ (\forall a. a \notin b_2 \ldots e_2 \cup b \ldots j \implies m(a) = \text{Emp}) \land \]
\[ \text{part\_heap (b, i) m (i - b)} \land \text{part\_heap (i, j) m} \land \]
\[ (\exists k. \text{part\_heap (b}_2, e_2) m k \land k \leq e - j) \land \]
\[ \text{ref\_mem (h, f) m} \land \text{ok\_heap (h}_0, \text{roots}_0) \land \]
\[ (h_0, \{\}, \{\}, \text{set roots}_0, \text{id}) \xrightarrow{\text{step}*} (h, \text{domain } h \cap (b \ldots i), \text{domain } h \cap (i \ldots j), z, f) \]

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!

Decompiation produces functions, e.g.

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

one proves this implements abstract version
A glimpse at CakeML’s pointers

Configurable data representation:

Example pointer value:

```
0...00110011101 00 01 010 1
```

<table>
<thead>
<tr>
<th>address value</th>
<th>padding</th>
<th>length</th>
<th>tag</th>
<th>marker</th>
</tr>
</thead>
</table>

\[\text{v} = \begin{cases} 
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{CodePtr num} \\
\text{RefPtr num} 
\end{cases}\]

These can be left out

Speeds up pattern matching, if present

“the tag”
What we learnt

ML compilers cannot preserve semantics completely:

\[
\text{machine}\_\text{semantics} \ (\text{compile} \ prog) \\
\in \text{extend}\_\text{with}\_\text{resource}\_\text{limit} \ (\text{semantics} \ prog)
\]

We add get-out clause to simulation theorems:

\[
\text{evaluate} \ code \ s1 = (res, s2) \land res \neq Rfail \land \\
\text{state}\_\text{rel} \ s1 \ t1 \Rightarrow \\
\exists t2 \ res2. \ \text{evaluate} \ (\text{compile} \ code) \ t1 = (res2, t2) \land \\
(\text{state}\_\text{rel} \ s2 \ t2 \land res2 = res \lor \\
res2 = R\text{HitHardLimit} \land t2.IO \text{ isPrefix} \ s2.IO)
\]

A verified GC is needed.

\textbf{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