Mechanically Verified LISP Intepreters

Magnus O. Myreen

University of Cambridge, UK

Nov 2008

Verified LISP interpreters

Why LISP?

The simplest real-world functional language for which I could verify an implementation.

The specification?

Given a string denoting an s-expression, the interpreters evaluates the expression and produce a string describing the result.

Interpreters?

Yes, multiple: ARM, PowerPC and x86 implementations.

This talk

Describes ideas behind proof techniques instead of detailed proofs.

1. machine-code specifications
2. decompilation into logic
3. proof-producing compilation
4. LISP proofs
5. summary, lessons learnt

Part 1. Machine code

— underlying models, Hoare triples

Machine code

Underlying processor models:

- **ARM** – developed by Anthony Fox, verified against a register-transfer lever model of an ARM processor;
- **x86** – developed together with Susmit Sarkar, Peter Sewell, Scott Owens, etc, heavily tested against a real processor;
- **PowerPC** – a HOL4 translation of Xavier Leroy's PowerPC model, used in his proof of an optimising C compiler.

Large detailed models...

Machine code, x86

Example, specification of x86 decoding:

```
"  8B /r     |  MOV r32, r/m32  ";
"  B8+rd id  |  MOV r32, imm32  ";
```

Snippet from operational semantics:

```
x86 exec ii (Xbinop binop_name ds) len = parT_unit
(seqT (read_eip ii) (λx. write_eip ii (x + len)))
(seqT
  (parT (read_src ea ii ds) (read_dest ea ii ds))
  (λ((ea_src , val_src), (ea_dest , val_dest)).
   write_binop ii binop_name val_dest val_src ea_dest))
```

Machine code, x86

Even 'simple' instructions get complex definition.

Sequential op.sem. evaluated for instruction "40" (i.e. `inc eax`):

$$
\begin{array}{l}
\mathit{x86\_read\_reg}\ \mathrm{EAX}\ \mathit{state} = \mathit{eax}\ \wedge \\
\mathit{x86\_read\_eip}\ \mathit{state} = \mathit{eip}\ \wedge \\
\mathit{x86\_read\_mem}\ \mathit{eip}\ \mathit{state} = \mathrm{some}\ \mathtt{0x40} \Rightarrow \\
\quad \mathit{x86\_next}\ \mathit{state} = \\
\quad\quad \mathrm{some}\ (\mathit{x86\_write\_reg}\ \mathrm{EAX}\ (\mathit{eax} + 1) \\
\quad\quad\quad\quad (\mathit{x86\_write\_eip}\ (\mathit{eip} + 1) \\
\quad\quad\quad\quad (\mathit{x86\_write\_eflag}\ \mathrm{AF}\ \mathrm{none} \\
\quad\quad\quad\quad (\mathit{x86\_write\_eflag}\ \mathrm{SF}\ (\mathrm{some}\ (\mathit{sign\_of}\,(\mathit{eax} + 1))) \\
\quad\quad\quad\quad (\mathit{x86\_write\_eflag}\ \mathrm{ZF}\ (\mathrm{some}\ (\mathit{eax} + 1 = 0)) \\
\quad\quad\quad\quad (\mathit{x86\_write\_eflag}\ \mathrm{PF}\ (\mathrm{some}\ (\mathit{parity\_of}\,(\mathit{eax} + 1))) \\
\quad\quad\quad\quad (\mathit{x86\_write\_eflag}\ \mathrm{OF}\ \mathrm{none}\ \mathit{state})))))))
\end{array}
$$

Machine code, specifications

Clearly some abbreviations are needed!

A machine-code Hoare triple:

$$\{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S}\,\}$$
$$p : 40$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S}\,\}$$

Here $\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag\ AF}\ a * \mathsf{eflag\ SF}\ s * \mathsf{eflag\ ZF}\ z * \ldots$.

In HOL4 syntax:

```
SPEC X86_MODEL
  (xR EAX a * xEIP p * xS)
  {(p,[0x40w])}
  (xR EAX (a+1w) * xEIP (p+1w) * xS)
```

Machine code, Hoare triple

The Hoare triple uses a separating conjunction $*$, defined over sets:

$$(p * q)\ s \ = \ \exists v\ u.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

Machine code, Hoare triple

The Hoare triple uses a separating conjunction $*$, defined over sets:

$$(p * q)\ s\ =\ \exists v\ u.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

We define translations from processor states to sets, for instance $x86\_to\_set$ can produce:

$$\{\ \ \text{xReg EAX } 5, \text{xReg EDX } 56, \text{xReg ECX } 89, \ldots,$$
$$\text{xMem } 0 \text{ none}, \text{xMem } 1 \text{ (some } 67), \text{xMem } 2 \text{ (some } 255), \ldots,$$
$$\text{xStatus AF (some } \mathit{true}), \text{xStatus ZF none}, \ldots\}$$

Let $\mathrm{R}\ \mathrm{a}\ x = \lambda s.\ (s = \{\text{xReg a } x\})$.

$$(\mathrm{R}\ a\ x * \mathrm{R}\ b\ y * p)\ (x86\_to\_set\ s) \Rightarrow a \neq b \wedge (x86\_read\_reg\ a\ s = x)$$

Machine code, Hoare triple definition

The Hoare triple's definition

$$\{p\}\, c\, \{q\} \;=\; \forall r\ s.\ (p * \text{code } c * r)\ (\mathit{to\_set}(s)) \Rightarrow$$
$$\exists n.\ (q * \text{code } c * r)\ (\mathit{to\_set}(\mathit{next}^n(s)))$$

Covers functional correctness, termination, resource usage.

$$\{\, \text{R EAX } a \,*\, \text{EIP } p \,*\, \text{S} \,\}$$
$$p : 40$$
$$\{\, \text{R EAX } (a{+}1) \,*\, \text{EIP } (p{+}1) \,*\, \text{S} \,\}$$

Machine code, memory accesses

A memory load:

$$a \in \text{domain } f \wedge \mathit{aligned}(a) \Rightarrow$$
$$\{\, \text{R ESI } a * \text{M } f * \text{EIP } p * \text{S} \,\}$$
$$p : \texttt{31C0}$$
$$\{\, \text{R ESI } (f(a)) * \text{M } f * \text{EIP } (p{+}2) * \text{S} \,\}$$

where $\text{M } f = \lambda s.\ (s = \{\text{xMem a (some } f(a)) \mid a \in \text{domain } f\})$.

Machine code, Hoare triple rules

compose: $\{p\}\, c\, \{m\} \wedge \{m\}\, c'\, \{q\} \;\Rightarrow\; \{p\}\, c \cup c'\, \{q\}$

frame: $\{p\}\, c\, \{q\} \;\Rightarrow\; \forall r.\ \{p * r\}\, c\, \{q * r\}$

where cond $g = \lambda s.\ (s = \{\}) \wedge g$.

Machine code, Hoare triple rules

compose: $\{p\}\, c\, \{m\} \wedge \{m\}\, c'\, \{q\} \;\Rightarrow\; \{p\}\, c \cup c'\, \{q\}$

frame: $\{p\}\, c\, \{q\} \;\Rightarrow\; \forall r.\ \{p * r\}\, c\, \{q * r\}$

cond: $\{p * \text{cond}\ g\}\, c\, \{q\} \;=\; g \Rightarrow \{p\}\, c\, \{q\}$

exists: $\{\exists x.\ p(x)\}\, c\, \{q\} \;=\; \forall x.\ \{p(x)\}\, c\, \{q\}$

where cond $g = \lambda s.\ (s = \{\}) \wedge g$.

Machine code, Hoare triple rules

compose: $\{p\}\, c\, \{m\} \wedge \{m\}\, c'\, \{q\} \;\Rightarrow\; \{p\}\, c \cup c'\, \{q\}$

frame: $\{p\}\, c\, \{q\} \;\Rightarrow\; \forall r.\ \{p * r\}\, c\, \{q * r\}$

cond: $\{p * \text{cond } g\}\, c\, \{q\} \;=\; g \Rightarrow \{p\}\, c\, \{q\}$

exists: $\{\exists x.\ p(x)\}\, c\, \{q\} \;=\; \forall x.\ \{p(x)\}\, c\, \{q\}$

id: $\{p\}\, c\, \{p\}$

extend: $\{p\}\, c\, \{q\} \;\Rightarrow\; \forall c'.\ \{p\}\, c \cup c'\, \{q\}$

where cond $g = \lambda s.\ (s = \{\}) \wedge g$.

Machine code, manual proofs

Tried to do proofs manually in HOL4, very tiresome.

Proved Schorr-Waite implementation and used it the verification of an in-place mark-and-sweep garbage collector.

Proof was unsatisfactory: long, tedious and tied to the ARM model.

Part 2. Decompilation into logic

— automating machine code proofs

Decompilation, overview

Conventional approach:

1. user annotates program with assertions
2. tool generates verification conditions (VCs)
3. user proves VCs

Decompilation approach:

1. tool translates program into recursive function
2. user proves function correct

Decompilation, idea

Given a while-program:

```
a := 0;
while (n ≠ 0) do
  a := a + 1;
  n := n - 2
end
```

Decompilation, idea

Given a while-program:

```
a := 0;
while (n ≠ 0) do
  a := a + 1;
  n := n - 2
end
```

automatic transformation produces:

```
f(a,n) = let a = 0 in g(a,n)

g(a,n) = if n = 0 then (a,n) else
            let a = a + 1 in
            let n = n - 2 in
              g(a,n)
```

Decompilation, certificate

"while-programs as recursive functions" – an idea by McCarthy from 1960s.

Novelty: automated in theorem prover, and produce a proof:

Decompilation, certificate

"while-programs as recursive functions" – an idea by McCarthy from 1960s.

Novelty: automated in theorem prover, and produce a proof:
The automatically proved statement:

```
f_pre(a,n) ⇒
HOARE_TRIPLE
  (VAR "a" a * VAR "n" n)
  (a := 0; while (n ≠ 0) do a := a + 1; n := n - 2 end)
  (let (a2,n2) = f(a,n) in (VAR "a" a2 * VAR "n" n2))

where  f_pre(a,n) = let a = 0 in g_pre(a,n)

       g_pre(a,n) = if n = 0 then true else
                      let a = a + 1 in
                      let n = n - 2 in
                        g_pre(a,n)
```

Decompilation, verification

Suppose we want to prove the while-program.

We look at the generated function:

```
f(a,n) = let a = 0 in g(a,n)

g(a,n) = if n = 0 then (a,n) else
           let a = a + 1 in
           let n = n - 2 in
             g(a,n)
```

prove that it computes the desired result:

$\forall$n a. 0 $\leq$ n $\Rightarrow$ (g(a,2$\times$n) = (n + a, 0)) $\wedge$ g_pre(a,2$\times$n)

$\forall$n a. 0 $\leq$ n $\wedge$ EVEN n $\Rightarrow$ (f(a,n) = (n DIV 2, 0)) $\wedge$ f_pre(a,n)

Very simple proofs (7-line HOL4-proof).

Decompilation, using the certificate

We now have two theorems:

```
∀n a. 0 ≤ n ∧ EVEN n ⇒ (f(a,n) = (n DIV 2, 0)) ∧ f_pre(a,n)

f_pre(a,n) ⇒
HOARE_TRIPLE
  (VAR "a" a * VAR "n" n)
  (a := 0; while (n ≠ 0) do a := a + 1; n := n - 2 end)
  (let (a2,n2) = f(a,n) in (VAR "a" a2 * VAR "n" n2))
```

Decompilation, using the certificate

We now have two theorems:

$\forall$n a. 0 $\leq$ n $\wedge$ EVEN n $\Rightarrow$ (f(a,n) = (n DIV 2, 0)) $\wedge$ f_pre(a,n)

```
f_pre(a,n) ⇒
HOARE_TRIPLE
  (VAR "a" a * VAR "n" n)
  (a := 0; while (n ≠ 0) do a := a + 1; n := n - 2 end)
  (let (a2,n2) = f(a,n) in (VAR "a" a2 * VAR "n" n2))
```

It is now easy to prove the code:

```
0 ≤ n ∧ EVEN n ⇒
HOARE_TRIPLE
  (VAR "a" a * VAR "n" n)
  (a := 0; while (n ≠ 0) do a := a + 1; n := n - 2 end)
  (VAR "a" (n DIV 2) * VAR "n" 0)
```

Comparison with the VC approach

Annotate the program:

```
{ pre n }
a := 0;
while (n ≠ 0) do { inv n } [ variant ]
  a := a + 1;
  n := n - 2
end
{ post n }
```

Comparison with the VC approach

Annotate the program:

```
{ pre n }
a := 0;
while (n ≠ 0) do { inv n } [ variant ]
  a := a + 1;
  n := n - 2
end
{ post n }
```

Define:

```
pre n = λs. (s("n") = n) ∧ EVEN n ∧ 0 ≤ n
post n = λs. (s("a") = n DIV 2) ∧ (s("n") = 0)
inv n = λs. (n = 2×s("a")+s("n")) ∧ EVEN s("n") ∧ 0 ≤ s("n")
variant = λs. s("n")
```

Comparison with the VC approach

If the user proves the verification conditions, then we have:

```
∀n. HOARE_TRIPLE (pre n) (a := 0; while ...) (post n)
```

Summary of comparison:

Comparison with the VC approach

If the user proves the verification conditions, then we have:

```
∀n. HOARE_TRIPLE (pre n) (a := 0; while ...) (post n)
```

Summary of comparison:

- VC proof requires more user input and is longer;

Comparison with the VC approach

If the user proves the verification conditions, then we have:

```
∀n. HOARE_TRIPLE (pre n) (a := 0; while ...) (post n)
```

Summary of comparison:

- VC proof requires more user input and is longer;
- VC proof requires the user to invent an invariant expression:

  ```
  n = 2 × s("a") + s("n")
  ```

  the new proof only required stating the desired result of the remaining part of the loop:

  ```
  g(a, 2 × n) = (n + a, 0)
  ```

Comparison with the VC approach

If the user proves the verification conditions, then we have:

```
∀n. HOARE_TRIPLE (pre n) (a := 0; while ...) (post n)
```

Summary of comparison:

- VC proof requires more user input and is longer;
- VC proof requires the user to invent an invariant expression:

  ```
  n = 2 × s("a") + s("n")
  ```

  the new proof only required stating the desired result of the remaining part of the loop:

  ```
  g(a, 2 × n) = (n + a, 0)
  ```

- VC proof uses a variant where the new proof uses induction;

Comparison with the VC approach

If the user proves the verification conditions, then we have:

```
∀n. HOARE_TRIPLE (pre n) (a := 0; while ...) (post n)
```

Summary of comparison:

- VC proof requires more user input and is longer;
- VC proof requires the user to invent an invariant expression:

  ```
  n = 2 × s("a") + s("n")
  ```

  the new proof only required stating the desired result of the remaining part of the loop:

  ```
  g(a, 2 × n) = (n + a, 0)
  ```

- VC proof uses a variant where the new proof uses induction;
- VC proof deals directly with the state `s`, the other does not.

Decompilation, core ideas

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$\textit{tailrec}_{G,F,D}(x) = \text{if } G(x) \text{ then } \textit{tailrec}_{G,F,D}(F(x)) \text{ else } D(x)$$

Decompilation, core ideas

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$\mathit{tailrec}_{G,F,D}(x) = \text{if } G(x) \text{ then } \mathit{tailrec}_{G,F,D}(F(x)) \text{ else } D(x)$$

2. specify termination for value x as

$$\mathit{pre}_{G,F}(x) = \exists n.\ \neg(G(F^n(x)))$$

Decompilation, core ideas

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$\mathit{tailrec}_{G,F,D}(x) = \text{if } G(x) \text{ then } \mathit{tailrec}_{G,F,D}(F(x)) \text{ else } D(x)$$

2. specify termination for value x as

$$\mathit{pre}_{G,F}(x) = \exists n.\ \neg(G(F^n(x)))$$

3. but give the user

$$\mathit{pre}_{G,F}(x) = \text{if } G(x) \text{ then } \mathit{pre}_{G,F}(F(x)) \text{ else true}$$

Decompilation, core ideas

4. use loop rule

$$
\begin{array}{l}
(\forall x.\ \texttt{HOARE\_TRIPLE}\ (p(x))\ c\ (p(F(x)))) \Rightarrow \\
(\forall x.\ \mathit{pre}_{G,F}(x) \Rightarrow \\
\qquad \texttt{HOARE\_TRIPLE}\ (p(x))\ (\texttt{while}\ G\ c)\ (p(\mathit{tailrec}_{G,F,id}(x))))
\end{array}
$$

Decompilation, core ideas

4. use loop rule

$$
\begin{array}{l}
(\forall x.\ \texttt{HOARE\_TRIPLE}\ (p(x))\ c\ (p(F(x)))) \Rightarrow \\
(\forall x.\ pre_{G,F}(x) \Rightarrow \\
\qquad \texttt{HOARE\_TRIPLE}\ (p(x))\ (\texttt{while}\ G\ c)\ (p(tailrec_{G,F,id}(x))))
\end{array}
$$

For machine code:

5. also work one loop at a time, loop rule:

$$
\begin{array}{l}
(\forall x.\ H\ x \wedge G(x) \Rightarrow \texttt{SPEC}\ m\ (p(x))\ c\ (p(F(x)))) \Rightarrow \\
(\forall x.\ H\ x \wedge \neg G(x) \Rightarrow \texttt{SPEC}\ m\ (p(x))\ c\ (q(D(x)))) \Rightarrow \\
(\forall x.\ pre_{G,F,H}(x) \Rightarrow \texttt{SPEC}\ m\ (p(x))\ c\ (q(tailrec_{G,F,D}(x))))
\end{array}
$$

with

$$
pre_{G,F,H}(x) = H(x) \wedge \text{if } G(x) \text{ then } pre_{G,F,H}(F(x)) \text{ else true}
$$

Decompilation, example

Given some hard-to-read machine code,

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

This transforms to a readable HOL4 function:

$$
\begin{array}{rcl}
f(r_0, r_1, m) & = & \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
g(r_0, r_1, m) & = & \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
& & \quad \text{let } r_0 = r_0{+}1 \text{ in} \\
& & \quad \text{let } r_1 = m(r_1) \text{ in} \\
& & \quad\quad g(r_0, r_1, m)
\end{array}
$$

Decompilation, example

Precondition keeps track of side-conditions:

$$
\begin{array}{rcl}
f\_pre(r_0, r_1, m) & = & \text{let } r_0 = 0 \text{ in } g\_pre(r_0, r_1, m) \\
g\_pre(r_0, r_1, m) & = & \text{if } r_1 = 0 \text{ then } \mathit{true} \text{ else} \\
 & & \quad \text{let } r_0 = r_0{+}1 \text{ in} \\
 & & \quad \text{let } \mathit{cond} = r_1 \in \text{domain } m \wedge \mathit{aligned}(r_1) \ \mathit{in} \\
 & & \quad \text{let } r_1 = m(r_1) \text{ in} \\
 & & \qquad g\_pre(r_0, r_1, m) \wedge \mathit{cond}
\end{array}
$$

Decompilation, example

Certificate:

$$f_{pre}(r_0, r_1, m) \Rightarrow$$
$$\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } (r_0, r_1, m) * \mathsf{PC}\ p \,\}$$
$$p : \mathtt{E3A00000},\ \ p{+}4 : \mathtt{E3510000}\ \ \ldots\ \ p{+}16 : \mathtt{1AFFFFFB}$$
$$\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } f(r_0, r_1, m) * \mathsf{PC}\ (p + 20) \,\}$$

Here $(\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } (r_0, r_1, m) = \mathsf{R0}\ r_0 * \mathsf{R1}\ r_1 * \mathsf{M}\ m$.

Decompilation, proof reuse

Manual verification proof:

$$\begin{array}{lcl} \forall x\ l\ a\ m.\ \mathit{list}(l,a,m) & \Rightarrow & f(x,a,m) = (\mathit{length}(l), 0, m) \\ \forall x\ l\ a\ m.\ \mathit{list}(l,a,m) & \Rightarrow & f_{pre}(x,a,m) \end{array}$$

Note: Proof not tied to ARM model.

In fact, similar x86 code and PowerPC code decompiles to f' and f'' such that $f = f' = f''$. Manual proof can be reused!

Decompilation, proof reuse

Manual verification proof:

$$\forall x\, l\, a\, m.\ \ list(l,a,m) \ \Rightarrow\ f(x,a,m) = (length(l), 0, m)$$
$$\forall x\, l\, a\, m.\ \ list(l,a,m) \ \Rightarrow\ f_{pre}(x,a,m)$$

Note: Proof not tied to ARM model.

In fact, similar x86 code and PowerPC code decompiles to f' and f'' such that $f = f' = f''$. Manual proof can be reused!

Proving $f = f'$ is easy in some case since

$$G = G' \wedge F = F' \wedge D = D' \ \Rightarrow\ tailrec_{G,F,D} = tailrec_{G',F',D'}$$

Part 3. Proof-producing compilation

— generating correct code

Compilation, idea

Decompilation: $\textit{code} \rightarrow \textit{function} \times \textit{certificate}$.

Compilation: $\textit{function} \rightarrow \textit{code} \times \textit{certificate}$.

Compilation, idea

Decompilation: *code* $\rightarrow$ *function* $\times$ *certificate*.

Compilation: *function* $\rightarrow$ *code* $\times$ *certificate*.

Compilation of function f:

1. generate code for f;
2. decompile code to produce proved-to-be-correct f';
3. automatically prove $f = f'$.

Note: step 1 can introduce arbitrary optimisations (instruction reordering, conditional execution ...) as long as step 3 succeeds.

Compilation, example

Compiling

$$f(r_1) \ = \ \text{if} \ \ r_1 < 10 \ \ \text{then} \ \ r_1 \ \ \text{else} \ \ \text{let} \ r_1 = r_1 - 10 \ \text{in} \ f(r_1)$$

produces ARM code:

```
E351000A   L:  cmp r1,#10
2241100A       subcs r1,r1,#10
2AFFFFFC       bcs L
```

Compilation, example

Compiling

$$f(r_1) \ = \ \text{if} \ \ r_1 < 10 \ \ \text{then} \ \ r_1 \ \ \text{else} \ \ \text{let} \ r_1 = r_1 - 10 \ \text{in} \ f(r_1)$$

produces ARM code:

```
E351000A   L:  cmp r1,#10
2241100A       subcs r1,r1,#10
2AFFFFFC       bcs L
```

and proves:

$$\{\text{R1}\ r_1 * \text{PC}\ p * \text{S}\}\ p : \texttt{E351000A}, \ldots\ \{\text{R1}\ f(r_1) * \text{PC}\ (p{+}12) * \text{S}\}$$

Compilation, example

Compiling

$$f(r_1) \ = \ \text{if} \ \ r_1 < 10 \ \ \text{then} \ \ r_1 \ \ \text{else} \ \ \text{let} \ r_1 = r_1 - 10 \ \text{in} \ f(r_1)$$

produces ARM code:

```
E351000A   L:  cmp r1,#10
2241100A       subcs r1,r1,#10
2AFFFFFC       bcs L
```

and proves:

$$\{\text{R1}\ r_1 * \text{PC}\ p * \text{S}\}\ p : \texttt{E351000A}, \ldots\ \{\text{R1}\ f(r_1) * \text{PC}\ (p{+}12) * \text{S}\}$$

Extension: If we prove "$f(x) = x$ mod 10", then compiler can be made to understand "let $r_1 = r_1$ mod 10 in", for ARM.

Compilation, input language

$$
\begin{array}{rcl}
\textit{input} & ::= & f(v, v, ..., v) = \textit{rhs} \\
\textit{rhs} & ::= & \text{let } r = \textit{exp} \text{ in } \textit{rhs} \quad | \quad \text{let } s = r \text{ in } \textit{rhs} \\
 & | & \text{let } m = m[\,\textit{address} \mapsto r\,] \text{ in } \textit{rhs} \\
 & | & \text{let } (v, v, ..., v) = g(v, v, ..., v) \text{ in } \textit{rhs} \\
 & | & \text{if } \textit{guard} \text{ then } \textit{rhs} \text{ else } \textit{rhs} \\
 & | & f(v, v, ..., v) \quad | \quad (v, v, ..., v) \\
\textit{exp} & ::= & x \mid \neg\, x \mid s \mid i_{32} \mid x \;\textit{binop}\; x \mid m \;\textit{address} \mid x \ll i_5 \mid x \gg i_5 \\
\textit{binop} & ::= & + \mid - \mid \times \mid \& \mid ?? \mid !! \\
\textit{compare} & ::= & < \mid \le \mid > \mid \ge \mid <. \mid \le. \mid >. \mid \ge. \mid = \\
\textit{guard} & ::= & \neg\; \textit{guard} \mid x \;\textit{compare}\; x \mid x \;\&\; x = 0 \\
\textit{address} & ::= & r \mid r + i_7 \mid r - i_7 \\
x & ::= & r \mid i_8 \\
v & ::= & r \mid s \mid m
\end{array}
$$

Part 4. LISP proofs

— decompiling primitives, compiling interpreters

LISP proofs, strategy

To produce verified LISP interpreters:

1. write ARM implementation of car, cdr, cons, etc.

LISP proofs, strategy

To produce verified LISP interpreters:

1. write ARM implementation of car, cdr, cons, etc.
2. verify these operations using decompilation;

LISP proofs, strategy

To produce verified LISP interpreters:

1. write ARM implementation of car, cdr, cons, etc.
2. verify these operations using decompilation;
3. reuse proofs for PowerPC and x86;

LISP proofs, strategy

To produce verified LISP interpreters:

1. write ARM implementation of car, cdr, cons, etc.
2. verify these operations using decompilation;
3. reuse proofs for PowerPC and x86;
4. define a *lisp_eval* as tail-recursive function, using car, cdr etc.

LISP proofs, strategy

To produce verified LISP interpreters:

1. write ARM implementation of car, cdr, cons, etc.
2. verify these operations using decompilation;
3. reuse proofs for PowerPC and x86;
4. define a *lisp_eval* as tail-recursive function, using car, cdr etc.
5. compile *lisp_eval* using proof-producing compilation.

LISP proofs, primitives

First define s-expression as HOL data-type:

$$x \ ::= \ \mathsf{Dot}\ x\ x \mid \mathsf{Num}\ n \mid \mathsf{Str}\ s$$

where n is natural numbers and s is strings.

LISP proofs, primitives

First define s-expression as HOL data-type:

$$x \ ::= \ \mathsf{Dot}\ x\ x \mid \mathsf{Num}\ n \mid \mathsf{Str}\ s$$

where n is natural numbers and s is strings.

Define basic LISP operations:

$$\begin{aligned} \mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\ \mathsf{cdr}\ (\mathsf{Dot}\ x\ y) &= y \\ \mathsf{cons}\ x\ y &= \mathsf{Dot}\ x\ y \\ \mathsf{plus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m+n) \end{aligned}$$

LISP proofs, primitives proved

The specification for ARM instruction executing ‘car’:

$$(\exists x\ y.\ v_1 = \mathrm{Dot}\ x\ y) \Rightarrow$$

$$\{\ \mathrm{LISP}\ \mathit{limit}\ (v_1, v_2, v_3, v_4, v_5, v_6) * \mathrm{PC}\ p\ \}$$
$$p : \mathtt{35E30003}$$
$$\{\ \mathrm{LISP}\ \mathit{limit}\ ((\mathrm{car}\ v_1), v_2, v_3, v_4, v_5, v_6) * \mathrm{PC}\ (p{+}1)\ \}$$

where

$$\begin{array}{l} \mathrm{LISP}\ \mathit{limit}\ (v_1, v_2, v_3, v_4, v_5, v_6)\ = \\ \quad \exists r_3\ r_4\ r_5\ r_6\ r_7\ r_8\ r_9\ m\ t. \\ \qquad \mathrm{R3}\ r_3 * \mathrm{R4}\ r_4 * \mathrm{R5}\ r_5 * \mathrm{R6}\ r_6 * \mathrm{R7}\ r_7 * \mathrm{R8}\ r_8 * \mathrm{R9}\ r_9 * \mathrm{M}\ m * \mathrm{M}\ t * \\ \qquad \mathrm{cond}(\mathrm{lisp\_inv}\ \mathit{limit}\ (v_1, v_2, v_3, v_4, v_5, v_6)\ (r_3, r_4, r_5, r_6, r_7, r_8, r_9, m, t)) \end{array}$$

with ‘lisp_inv’ relating abstract and concrete states.

LISP proofs, cons

The specification for 'cons':

$$size\ v_1 + size\ v_2 + size\ v_3 + size\ v_4 + size\ v_5 + size\ v_6 < limit \Rightarrow$$

$$\{\, \text{LISP}\ limit\ (v_1, v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC}\ p \,\}$$
$$p : \ldots code \ldots$$
$$\{\, \text{LISP}\ limit\ ((\text{cons}\ v_1\ v_2), v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC}\ (p{+}336) \,\}$$

where

$$\begin{aligned} \text{size}\ (\text{Str}\ s) &= 0 \\ \text{size}\ (\text{Num}\ n) &= 0 \\ \text{size}\ (\text{Dot}\ x\ y) &= \text{size}\ x + \text{size}\ y + 1 \end{aligned}$$

LISP proofs, memory usage

Memory layout:

```
(heap half 1)           (heap half 2)           (table)
[xxxxxxxxxxxx.........] [.....................] [symbols]
```

LISP proofs, memory usage

Memory layout:

```
(heap half 1)             (heap half 2)             (table)
[xxxxxxxxxxxx.........] [.....................] [symbols]
```

Memory not wasted:

$$\{\,\text{LISP}\ \mathit{limit}\ (v_1, v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC}\ p\,\}$$
$$p : \mathit{...code...}$$
$$\{\,\text{LISP}\ \mathit{limit}\ ((\text{equal}\ v_1\ v_2), v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC}\ (p{+}232)\,\}$$

where

$$\text{equal}\ x\ y \;=\; \text{if}\ x = y\ \text{then Str ``T'' else Str ``nil''}$$

LISP proofs, compile

Verified primitives supplied to compiler, makes it understand:

$$\begin{array}{l} \text{let } v_1 = \text{cons } v_1 \; v_2 \text{ in} \\ \text{let } v_1 = \text{equal } v_1 \; v_2 \text{ in} \\ \text{let } v_1 = \text{car } v_1 \text{ in} \\ \text{let } v_4 = \text{cdr } v_2 \text{ in} \\ \ldots \end{array}$$

LISP proofs, compile

Example:

$$f(v_1, v_2, v_3, v_4, v_5, v_6) \;=\; \begin{array}{l} \text{if } v_2 = \text{Str "nil" then} \\ \quad (v_1, v_2, v_3, v_4, v_5, v_6) \\ \text{else} \\ \quad \text{let } v_2 = \text{cdr } v_2 \text{ in} \\ \quad \text{let } v_1 = \text{cons } v_1\ v_2 \text{ in} \\ \quad\quad f(v_1, v_2, v_3, v_4, v_5, v_6) \end{array}$$

compiles to:

$$f_{pre}(v_1, v_2, v_3, v_4, v_5, v_6, \mathit{limit}) \Rightarrow$$

$$\{\, \text{LISP } \mathit{limit}\ (v_1, v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC } p \,\}$$
$$p : \mathit{...code...}$$
$$\{\, \text{LISP } \mathit{limit}\ f(v_1, v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC } (p{+}356) \,\}$$

LISP proofs, eval

So *lisp_eval* is defined as tail-recursion with

$$\begin{array}{lll} v_1 & - & \text{s-exp to be evaluated} \\ v_2 & - & \text{temp var 1} \\ v_3 & - & \text{temp var 2} \\ v_4 & - & \text{stack/continuation} \\ v_5 & - & \text{store: list of symbol, value pairs} \\ v_6 & - & \text{task variable} \end{array}$$

compiles to:

$$\mathit{lisp\_eval}_{pre}(v_1, v_2, v_3, v_4, v_5, v_6, \mathit{limit}) \Rightarrow$$

$$\{\,\text{LISP}\ \mathit{limit}\ (v_1, v_2, v_3, v_4, v_5, v_6)) * \text{S} * \text{PC}\ p\,\}$$
$$p : \ldots code \ldots$$
$$\{\,\text{LISP}\ \mathit{limit}\ \mathit{lisp\_eval}(v_1, v_2, v_3, v_4, v_5, v_6) * \text{S} * \text{PC}\ (p{+}1452)\,\}$$

Future work

(not-yet-proved) specification with parsing and printing:

$$\mathit{lisp\_eval}_{\mathit{pre}}(\mathit{exp}, \mathit{limit}) \Rightarrow$$

$$\{\, \text{String } a \; (\mathit{sexp2str}(\mathit{exp})) * \ldots * \text{S} * \text{PC } p \,\}$$
$$p : \ldots\mathit{code}\ldots$$
$$\{\, \text{String } a \; (\mathit{sexp2str}(\mathit{lisp\_eval}(\mathit{exp}))) * \ldots * \text{S} * \text{PC } (p{+}\ldots) \,\}$$

Further work

1. finish string-to-string specification
2. add support for bignum, rationals, complex rationals...
3. verify an ACL2 evaluator

Part 5. Summary

— lessons learnt

Summary

Verified LISP interpreters were produced by:

1. verifying ARM implementations of car, cdr, cons, etc. using decompilation
2. reusing proofs for PowerPC and x86;
3. defining a *lisp_eval* as tail-recursive function;
4. compiling *lisp_eval* using proof-producing compilation.

Summary

Verified LISP interpreters were produced by:

1. verifying ARM implementations of car, cdr, cons, etc. using decompilation
2. reusing proofs for PowerPC and x86;
3. defining a *lisp_eval* as tail-recursive function;
4. compiling *lisp_eval* using proof-producing compilation.

Lesson learnt:

"Make the proof easy for the theorem prover" – Mike Gordon.

(It made sense to turn everything into recursive functions.)