Verification of LISP Interpreters

Magnus O. Myreen

Computer Laboratory, University of Cambridge, Cambridge, UK magnus.myreen@cl.cam.ac.uk

Abstract. LISP interpreters provide a clean abstraction over machine details such as bounded arithmetic and array-like memory. We describe our approach and partial results in verifying LISP interpreters implemented in ARM, PowerPC and x86 machine code.

1 Introduction

The aim of this project is to create verified machine-code implementations of interpreters for a language similar to LISP 1.5 [4]. The target is to deliver verified LISP interpreters in ARM, PowerPC and x86 machine code. Benefits:

1. LISP interpreters provide a clean abstraction of mathematically unclean machine specific details (boundedness of arithmetic, array-like memory etc.). Hence future program proofs can assume a clean abstraction.
2. Subsets of LISP might turn out to be good targets for compilers from proof tools such as HOL4, Isabelle and Coq.
3. Verified LISP interpreters can serve as a proved-to-be-correct environments for evaluating ACL2 s-expressions.

The fact that the proof goes down to detailed models of commercial machine languages distinguishes our project from the VLISP project [1] which stopped at the algorithm level. All proofs are carried out within HOL4 [7].

2 Approach

Our three-stage approach for creating verified machine-code implementations of LISP interpreters:

a. write efficient machine-code implementations of primitive operations over s-expressions held in a garbage collected heap (`cons`, `car`, `cdr`, `eq`, `+` etc.).

b. verify primitive heap operations with aid of automatic decompilation into recursive HOL functions (a form of automatic reverse engineering).

c. automatically plug together verified elementary operations to create verified LISP interpreters (a form of proof-producing compilation).

The key research challenge lies in steps *b* and *c*. The solutions we are investigating are centered around a new approach for proving (fully-automatically) that the functional behavior of machine code is described by a recursive function. An illustration of this proof technique is given in the next section. Subsequent sections describe its applications to automatic proof-producing decompilation and compilation. Our approach to decompilation is detailed in Myreen et al. [6]. The compilation work is a result of collaboration with Li, Slind and Owens [2].

2.1 Proving that a recursive function is executed by machine code

We illustrate our proof technique with an example. The illustrated technique can be applied one loop at a time. Subsequent proofs can use previously proved specification, hence nested loops, procedure calls and even some non-properly nested loops can be handled.

Consider the following ARM code, which repeatedly subtracts 10 from register 1 until its (unsigned) value is less than 10,

```
L: cmp   r1,#10      | compare r1 with 10
   subcs r1,r1,#10   | subtract 10 from r1,  if cmp gave 10 ≤ r1
   bcs   L           | jump to top,          if cmp gave 10 ≤ r1
```

The following function describes the behavior of the above ARM code.

$$f(x) = \mathsf{if}\ 10 \leq x\ \mathsf{then}\ f(x{-}10)\ \mathsf{else}\ x \tag{1}$$

The correspondence between f and the ARM code (call it *code*) can be stated as follows. Let $\mathsf{pc}\ p$ assert that the program counter has value p and similarly let $\mathsf{r1}\ x$ state that register 1 has value x.

$$\{\, \mathsf{r1}\ x * \mathsf{pc}\ p \,\}\ \ code\ \ \{\, \mathsf{r1}\ f(x) * \mathsf{pc}\ (p{+}3) \,\} \tag{2}$$

Informally this machine-code specification [5, 6] states: given a state where register 1 has value x and the program counter is p, *code* will reach a state where register 1 has value $f(x)$ and the program counter is $p{+}3$.

Such a theorem can be proved automatically in two steps: first compose specifications for the individual instructions; and then instantiate a special loop rule. The result of composing the specifications of the individual instructions produces two specifications:

$$10 \leq x\ \Rightarrow\ \{\, \mathsf{r1}\ x * \mathsf{pc}\ p \,\}\ \ code\ \ \{\, \mathsf{r1}\ (x{-}10) * \mathsf{pc}\ p \,\} \tag{3}$$

$$x < 10\ \Rightarrow\ \{\, \mathsf{r1}\ x * \mathsf{pc}\ p \,\}\ \ code\ \ \{\, \mathsf{r1}\ x * \mathsf{pc}\ (p{+}3) \,\} \tag{4}$$

The desired specification (2) is a consequence of the following loop rule instantiated with $\mathsf{res} = \lambda x.\ \mathsf{r1}\ x * \mathsf{pc}\ p$ and $\mathsf{res'} = \lambda x.\ \mathsf{r1}\ x * \mathsf{pc}\ (p{+}3)$:

$$\begin{aligned} \forall \mathsf{res}\ \mathsf{res'}\ c.\ \ & (\forall x.\ \ G(x) \Rightarrow \{\mathsf{res}\ x\}\, c\, \{\mathsf{res}\ \ F(x)\}) \wedge \\ & (\forall x.\ \neg G(x) \Rightarrow \{\mathsf{res}\ x\}\, c\, \{\mathsf{res'}\ D(x)\}) \Rightarrow \\ & (\forall x.\ pre(x) \Rightarrow \{\mathsf{res}\ x\}\, c\, \{\mathsf{res'}\ tailrec(x)\}) \end{aligned}$$

Here *tailrec* is the generic form of any tail-recursion and *pre* ensures termination:

$$tailrec(x) = \mathsf{if}\ G(x)\ \mathsf{then}\ tailrec(F(x))\ \mathsf{else}\ D(x)$$

$$pre(x) = \mathsf{if}\ G(x)\ \mathsf{then}\ pre(F(x))\ \mathsf{else}\ true$$

tailrec was defined using a trick by Moore and Manolios [3] and *pre* is defined as $pre(x) = \exists n.\ \neg G(step(n, F, x))$ with *step* defined recursively as:

$$step(0, F, x) = x$$

$$step(n{+}1, F, x) = step(n, F, F(x))$$

The proof of the loop rule above is described in Myreen et al. [6].

2.2 Automatic reverse engineering – decompiling into HOL

Machine code is a sequence of machine words, e.g. the ARM program used in the section above is the following in hexadecimal encoding:

```
E351000A 2241100A 2AFFFFFC
```

When decompiling into HOL, we start with a sequence of machine words representing the code. The goal is to construct a HOL function that describes the effect of the code, and to prove automatically that the HOL function corresponds to the machine code. Decompilation can be fully automated as follows:

1. Automatically derive basic specifications for each individual instruction.
2. Analise the specifications and from them generate a tail-recursive function describing the code, e.g. the following f is generated for the code above:

$$f(r_1) = \mathsf{if}\ 10 \leq r_1\ \mathsf{then}\ f(r_1{-}10)\ \mathsf{else}\ r_1$$

3. Run the proof procedure described above to prove that f is executed by the original machine code.

2.3 Automatic proof-producing compilation

Given a function f that operates over machine words, e.g.

$$f(x) = \mathsf{if}\ x < 10\ \mathsf{then}\ x\ \mathsf{else}\ f(x{-}10)$$

we can construct (and automatically prove it equivalent to) a function g where variable names, let-expressions and if-statements correspond to machine code.

$$\begin{array}{l} g(r_1) = \mathsf{if}\ r_1 < 10\ \mathsf{then}\ r_1\ \mathsf{else} \\ \qquad\qquad \mathsf{let}\ r_1 = r_1{-}10\ \mathsf{in} \\ \qquad\qquad g(r_1) \end{array}$$

Such functions are easy to turn into assembly code, which can then be turned into machine code by off-the-shelf assemblers. The assembler-generated machine code is proved correct by the automation described in Section 2.1.

3 Resulting specification

The proof automation described above is based on automatically composing specifications to construct correctness theorems. By default, the initial specifications are automatically derived specification for individual machine instructions. However, the user can instead supply the compiler/decompiler with alternative specification in order to build on previously proved specifications.

We make use of this feature for implementation of LISP interpreters. First an abstract data-type was defined for s-expressions: $\mathsf{Dot}\ x\ y$ is a pair, $\mathsf{Num}\ i$ is an unbounded integer i, and $\mathsf{Str}\ s$ is a character string s. Some basic operations:

$$\begin{array}{rl} \mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\ \mathsf{cdr}\ (\mathsf{Dot}\ x\ y) &= y \\ \\ \mathsf{ltype}\ (\mathsf{Num}\ w) &= \mathsf{Num}\ 0 \\ \mathsf{ltype}\ (\mathsf{Str}\ s) &= \mathsf{Num}\ 1 \\ \mathsf{ltype}\ (\mathsf{Dot}\ x\ y) &= \mathsf{Num}\ 2 \\ \\ \mathsf{size}\ (\mathsf{Num}\ w) &= 0 \\ \mathsf{size}\ (\mathsf{Str}\ s) &= 0 \\ \mathsf{size}\ (\mathsf{Dot}\ x\ y) &= 1 + \mathsf{size}\ x + \mathsf{size}\ y \end{array}$$

A new resource specification heap was defined which relates variables $task$, exp, x, y, s, env of the new type to concrete memory representations using a coupling invariant $machine\_repr$. Here a is the address of the heap and l is its capacity.

$$\begin{array}{l} \mathsf{heap}\ (a,l)\ (task, exp, x, y, s, env)\ = \\ \quad \exists r_3\ r_4\ r_5\ r_6\ r_7\ r_8\ f. \\ \qquad \mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{r5}\ r_5 * \mathsf{r6}\ r_6 * \mathsf{r7}\ r_7 * \mathsf{r8}\ r_8 * \mathsf{r9}\ r_9 * \mathsf{memory}\ f\ * \\ \qquad machine\_repr\ (task, exp, x, y, s, env, l)\ (r_3, r_4, r_5, r_6, r_7, r_8, a, f) \end{array}$$

Machine-code implementations for basic LISP operations were verified using decompilation followed by some manual proofs, e.g. ARM code for car of exp:

$$\begin{array}{l}\mathsf{ltype}\ exp = \mathsf{Num}\ 2\ \Rightarrow \\ \{\ \mathsf{heap}\ (a,l)\ (task, exp, x, y, s, env) * \mathsf{pc}\ p\ \} \\ \quad p : \texttt{E5933000} \\ \{\ \mathsf{heap}\ (a,l)\ (task, (\mathsf{car}\ exp), x, y, s, env) * \mathsf{pc}\ (p{+}1)\ \}\end{array}$$

A memory allocator with a built-in Cheney-collector was used to implement creation of a new pair Dot x y. The precondition of this operation requires the heap to have enough space to accommodate a new cons-cell.

$$\begin{array}{l}(\mathsf{size}\ task + \mathsf{size}\ exp + \mathsf{size}\ x + \mathsf{size}\ y + \mathsf{size}\ s + \mathsf{size}\ env) < l\ \Rightarrow \\ \{\ \mathsf{heap}\ (a,l)\ (task, exp, x, y, s, env) * \mathsf{pc}\ p\ \} \\ \quad p : \text{... the allocator code ...} \\ \{\ \mathsf{heap}\ (a,l)\ (task, (\mathsf{Dot}\ x\ y), x, y, s, env) * \mathsf{pc}\ (p{+}87)\ \}\end{array}$$

When the above specifications were supplied to the compiler it knows what machine code to generate for two new commands: one for calculating car of exp

$$\mathsf{let}\ exp = \mathsf{car}\ exp\ \mathsf{in}$$

and one for producing a new Dot-pair:

$$\mathsf{let}\ exp = \mathsf{Dot}\ x\ y\ \mathsf{in}$$

Once the compilers language has been extended with sufficiently many such primitive operations, a LISP interpreter can be compiled using the compilation technique from above, since our compilation approach essentially only plugs together previously proved specifications. The top-level specification function defining a simple LISP interpreter *lisp_eval* is sketched in Figure 1 (on the following page). The correctness theorem will be of the form:

$$\begin{array}{l}lisp\_eval\_pre(task, exp, x, y, s, env)\ \Rightarrow \\ \{\ \mathsf{heap}\ (a,l)\ (task, exp, x, y, s, env) * \mathsf{pc}\ p\ \} \\ \quad p : \text{... code ...} \\ \{\ \mathsf{heap}\ (a,l)\ (lisp\_eval(task, exp, x, y, s, env)) * \mathsf{pc}\ (p{+}code\_length)\ \}\end{array}$$

Here *lisp_eval_pre* is an automatically generated predicate which has collected the various side-conditions that need to be true for proper execution of the machine-code implementations, i.e. *lisp_eval_pre* is a function which returns true only if the input makes *lisp_eval* terminate and the capacity of the heap is not exceeded during execution for this input.

Acknowledgments. The author would like to thank Mike Gordon, Konrad Slind, Thomas Tuerk, Anthony Fox, Larry Paulson and John Regehr for research discussions, and is grateful for funding from Osk.Huttusen säätiö, Finland, and EPSRC, UK.

References

1. Joshua Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.
2. Guodong Li, Scott Owens, and Konrad Slind. A proof-producing software compiler for a subset of higher order logic. In *European Symposium on Programming (ESOP)*, LNCS, pages 205–219. Springer-Verlag, 2007.
3. Panagiotis Manolios and J. Strother Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.
4. John McCarthy. *LISP 1.5 Programmer's Manual.* The MIT Press, 1962.
5. Magnus O. Myreen and Michael J.C. Gordon. A Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 568–582. Springer-Verlag, 2007.
6. Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2008.
7. Michael Norrish and Konrad Slind. *The HOL4 theorem proving system.* SourceForge: www.sourceforge.net, 2001.

```
TASK_EVAL = NUM 0
TASK_CONT = NUM 1

LISP_EVAL0 (task,exp,x,y,s,env) =
  if x = STR "'" then
    let task = TASK_CONT in
      (task,exp,x,y,s,env)
  else
    ...

LISP_EVAL1 (task,exp,x,y,s,env) = ...
LISP_EVAL2 (task,exp,x,y,s,env) = ...
LISP_LOOKUP (exp,x,y,env) = ...
LISP_POP (task,x,y,s) = ...

LISP_EVAL (task,exp,x,y,s,env) =
  if task = TASK_EVAL then
    let task = TASK_CONT in
    let x = LTYPE exp in
      if x = NUM 0 then (* exp is NUM *)
        LISP_EVAL (task,exp,x,y,s,env)
      else if x = NUM 1 then (* exp is STR *)
        let (exp,x,y,env) = LISP_LOOKUP (exp,x,y,env) in
          LISP_EVAL (task,exp,x,y,s,env)
      else (* if x = NUM 2 then, *) (* exp is DOT *)
        let (x,exp) = (CAR exp, CDR exp) in
        let (exp,y) = (CAR exp, CDR exp) in
        let (task,exp,x,y,s,env) = LISP_EVAL0 (task,exp,x,y,s,env) in
          LISP_EVAL (task,exp,x,y,s,env)
  else (* if task = TASK_CONT then *)
    let (task,x,y,s) = LISP_POP (task,x,y,s) in
      if x = STR "nil" then (* evaluation complete, exit *)
        (task,exp,x,y,s,env)
      else
        if task = STR "nil" then (* one arg has been evaluated *)
          let (task,exp,x,y,s,env) = LISP_EVAL1 (task,exp,x,y,s,env) in
            LISP_EVAL (task,exp,x,y,s,env)
        else (* two args have been evaluated *)
          let (task,exp,x,y,s,env) = LISP_EVAL2 (task,exp,x,y,s,env) in
            LISP_EVAL (task,exp,x,y,s,env)
```

Fig. 1. Parts of the definition of *lisp_eval* in HOL4.