Verified LISP implementations on ARM, x86 and PowerPC

Magnus O. Myreen and Michael J. C. Gordon

Computer Laboratory, University of Cambridge, UK

Abstract. This paper reports on a case study, which we believe is the first to produce a formally verified end-to-end implementation of a functional programming language running on commercial processors. Interpreters for the core of McCarthy's LISP 1.5 were implemented in ARM, x86 and PowerPC machine code, and proved to correctly parse, evaluate and print LISP s-expressions. The proof of evaluation required working on top of verified implementations of memory allocation and garbage collection. All proofs are mechanised in the HOL4 theorem prover.

1 Introduction

Explicit pointer manipulation is an endless source of errors in low-level programs. Functional programming languages hide pointers and thereby achieve a more abstract programming environment. The downside with functional programming (and Java/C# programming) is that the programmer has to trust automatic memory management routines built into run-time environments.

In this paper we report on a case study, which we believe is the first to produce a formally verified end-to-end implementation of a functional programming language. We have implemented, in ARM, x86 and PowerPC machine code, a program which parses, evaluates and prints LISP; and furthermore formally proved that our implementation respects a semantics of the core of LISP 1.5 [6]. Instead of assuming correctness of run-time routines, we build on a verified implementation of allocation and garbage collection.

For a flavour of what we have implemented and proved consider an example: if our implementation is supplied with the following call to `pascal-triangle`,

```
(pascal-triangle '((1)) '6)
```

it parses the string, evaluates the expression and prints a string,

```
((1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
```

where `pascal-triangle` had been supplied to it as

```
(label pascal-triangle
  (lambda (rest n)
    (cond ((equal n '0) rest)
          ('t (pascal-triangle
                (cons (pascal-next '0 (car rest)) rest) (- n '1))))))
```

with auxiliary function:

```
(label pascal-next
  (lambda (p xs)
    (cond ((atomp xs) (cons p 'nil))
          ('t (cons (+ p (car xs)) (pascal-next (car xs) (cdr xs)))))))
```

The theorem we have proved about our LISP implementation can be used to show e.g. that running `pascal-triangle` will terminate and print the first $n+1$ rows of Pascal's triangle, without a premature exit due to lack of heap space. One can use our theorem to derive sufficient conditions on the inputs to guarantee that there will be enough heap space.

We envision that our verified LISP interpreter will provide a platform on top of which formally verified software can be produced with much greater ease than at lower levels of abstraction, i.e. in languages where pointers are made explicit.

Why LISP? We chose to implement and verify a LISP interpreter since LISP has a neat definition of both syntax and semantics [12] and is still a very powerful language as one can see, for example, in the success of ACL2 [8]. By choosing LISP we avoided verifying machine code which performs static type checking.

Our proofs [14] are mechanised in the HOL4 theorem prover [19].

2 Methodology

Instead of delving into the many detailed invariants developed for our proofs, this paper will concentrate on describing the methodology we used:

- ▷ First, machine code for various LISP primitives, such as car, cdr, cons, was written and verified (Section 3);
  - The correctness of each code snippets is expressed as a machine-code Hoare triple [15]: $\{\, pre * \mathsf{pc}\ p \,\}\ p : code\ \{\, post * \mathsf{pc}\ (p + exit) \,\}$.
  - For cons and equal we used previously developed proof automation [15], which allows for proof reuse in between different machine languages.
- ▷ Second, the verified LISP primitives were input into a proof-producing compiler in such a way that the compiler can view the processors as a machine with six registers containing LISP s-expressions (Section 4);
  - The compiler [16] we use maps tail-recursive functions, defined in the logic of HOL4, down to machine code and proves that the generated code executes the original HOL4 functions.
  - Theorems describing the LISP primitives were input into the compiler, which can use them as building blocks when deriving new code/proofs.

▷ Third, LISP evaluation was defined as a (partially-specified) tail-recursive function lisp_eval, and then compiled into machine code using the compiler mentioned above (Section 5).

- LISP evaluation was defined as a tail-recursive function which only uses expressions/names for which the compiler has verified building blocks.
- lisp_eval maintains a stack and a symbol-value list.

▷ Fourth, to gain confidence that lisp_eval implements 'LISP evaluation', we proved that lisp_eval implements a semantics of LISP 1.5 [12] (Section 6).

- Our relational semantics of LISP [6] is a formalisation of a subset of McCarthy's original LISP 1.5 [12], with dynamic binding.
- The semantics abstracts the stack and certain evaluation orders.

▷ Finally, the verified LISP interpreters were sandwiched between a verified parser and printer to produce string-to-string theorems describing the behaviour of the entire implementation (Section 7).

- The parser and printer code, respectively, sets up and tears down an appropriate heap for s-expressions.

Sections 8 and 9 give quantitative data on the effort and discuss related work, respectively. Some definitions and proofs are presented in the Appendixes.

3 LISP primitives

LISP programs are expressed in and operate over s-expressions, expressions that are either a (natural) number, a symbol or a pair of s-expressions. In HOL, s-expressions are readily modelled using a data-type with constructors:

$$\begin{aligned} \mathsf{Num} &: \mathbb{N} \to \mathsf{SExp} \\ \mathsf{Sym} &: \mathsf{string} \to \mathsf{SExp} \\ \mathsf{Dot} &: \mathsf{SExp} \to \mathsf{SExp} \to \mathsf{SExp} \end{aligned}$$

LISP programs and s-expressions are conventionally written in an abbreviated string form. A few examples will illustrate the correspondence, which is given a formal definition in Appendix D.

`(car x)`	means	Dot (Sym "car") (Dot (Sym "x") (Sym "nil"))
`(1 2 3)`	means	Dot (Num 1) (Dot (Num 2) (Dot (Num 3) (Sym "nil")))
`'f`	means	Dot (Sym "quote") (Dot (Sym "f") (Sym "nil"))
`(4 . 5)`	means	Dot (Num 4) (Num 5)

Some basic LISP primitives are defined over SExp as follows:

$$\begin{aligned} \mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\ \mathsf{cdr}\ (\mathsf{Dot}\ x\ y) &= y \end{aligned}$$

$$\mathsf{cons}\ x\ y = \mathsf{Dot}\ x\ y$$

$$\begin{aligned}
\mathsf{plus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m+n)\\
\mathsf{minus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m-n)\\
\mathsf{times}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m\times n)\\
\mathsf{division}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m\ \mathsf{div}\ n)\\
\mathsf{modulus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m\ \mathsf{mod}\ n)
\end{aligned}$$

$$\begin{aligned}
\mathsf{equal}\ x\ y &= \mathsf{if}\ x=y\ \mathsf{then}\ \mathsf{Sym}\ \texttt{"t"}\ \mathsf{else}\ \mathsf{Sym}\ \texttt{"nil"}\\
\mathsf{less}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{if}\ m<n\ \mathsf{then}\ \mathsf{Sym}\ \texttt{"t"}\ \mathsf{else}\ \mathsf{Sym}\ \texttt{"nil"}
\end{aligned}$$

In the definition of equal, expression $x = y$ tests standard structural equality.

3.1 Specification of primitive operations

Before writing and verifying the machine code implementing primitive LISP operations, a decision had to be made how to represent Num, Sym and Dot on a real machine. To keep memory usage to a minimum each Dot-pair is represented as a block of two pointers stored consecutively on the heap, each Num n is represented as a 32-bit word containing $4 \times n + 2$ (i.e. only natural numbers $0 \leq n < 2^{30}$ are representable), and each Sym s is represented as a 32-bit word containing $4{\times}i{+}3$, where i is the row number of symbol s in a symbol table which, in our implementation, is a linked-list kept outside of the garbage-collected heap.

Here '+2' and '+3' are used as tags to make sure that the garbage collector can distinguish Num and Sym values from proper pointers. Pointers to Dot-pairs are word-aligned, i.e. a mod $4 = 0$, a condition the collector tests by computing $a \,\&\, 3 = 0$, where & is bitwise-and.

This simple and small representation of SExp allows most LISP primitives from the previous section to be implemented in one or two machine instructions. For example, taking car of register 3 and storing the result in register 4 is implemented on ARM as a load instruction:

```
E5934000   ldr r4,[r3]    (* load into reg 4, memory at address reg 3 *)
```

Similarly, ARM code for performing LISP operation plus of register 3 and 4, and storing the result into register 3 is implemented by:

```
E0833004   add r3,r3,r4   (* reg 3 is assigned value reg 3 + reg 4 *)
E2433002   sub r3,r3,#2   (* reg 3 is assigned value reg 3 - 2 *)
```

The intuition here is: $(4 \times m + 2) + (4 \times n + 2) - 2 = 4 \times (m + n) + 2$.

The correctness of the above implementations of car and plus is expressed formally by the two ARM Hoare triples [15] below. Here lisp $(v_1, v_2, v_3, v_4, v_5, v_6, l)$ is an assertion, defined below, which asserts that a heap with room for l Dot-pairs is located in memory and that s-expressions $v_1 ... v_6$ (each of type SExp) are stored in machine registers. This lisp assertion should be understood as lifting

the level of abstraction to a level where specific machine instructions make the processor seem as if it has six[1] registers containing s-expressions, of type SExp.

$$
\begin{array}{l}
(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \Rightarrow \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\ p : \texttt{E5934000} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}
\end{array}
$$

$$
\begin{array}{l}
(\exists m\ n.\ \mathsf{Num}\ m = v_1 \wedge \mathsf{Num}\ n = v_2 \wedge m{+}n < 2^{30})\ \Rightarrow \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\ p : \texttt{E0833004 E2433002} \\
\{\ \mathsf{lisp}\ (\mathsf{plus}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+8)\ \}
\end{array}
$$

The new assertion is defined for ARM (lisp), x86 (lisp'), and PowerPC (lisp") as maintaining a relation lisp_inv between the abstract state $v_1 ... v_6$ (each of type SExp) and the concrete state $x_1 ... x_6$ (each of type 32-bit word). The details of lisp_inv (defined in Appendix A) and the separating conjunction $*$ (explained in Myreen [14]) are unimportant for this presentation.

$$
\begin{array}{l}
\mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) = \\
\quad \exists x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ m_1\ m_2\ m_3\ a\ temp.\ \ \mathsf{m}\ m_1 * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\qquad \mathsf{r2}\ temp * \mathsf{r3}\ x_1 * \mathsf{r4}\ x_2 * \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \\
\qquad \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m_1, m_2, m_3) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lisp'}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) = \\
\quad \exists x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ m_1\ m_2\ m_3\ a.\ \ \mathsf{m}\ m_1 * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\qquad \mathsf{eax}\ x_1 * \mathsf{ecx}\ x_2 * \mathsf{edx}\ x_3 * \mathsf{ebx}\ x_4 * \mathsf{esi}\ x_5 * \mathsf{edi}\ x_6 * \mathsf{ebp}\ a * \\
\qquad \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m_1, m_2, m_3) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lisp"}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) = \\
\quad \exists x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ m_1\ m_2\ m_3\ a\ temp.\ \ \mathsf{m}\ m_1 * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\qquad \mathsf{r2}\ temp * \mathsf{r3}\ x_1 * \mathsf{r4}\ x_2 * \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \\
\qquad \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m_1, m_2, m_3) \rangle
\end{array}
$$

The following examples will use only lisp defined for ARM.

3.2 Memory layout and specification of 'cons' and 'equal'

Two LISP primitives required code longer than one or two machine instructions, namely cons and equal. Memory allocation, i.e. cons, requires an allocation procedure combined with a garbage collector. However, the top-level specification, which is explained next, hides these facts. Let size count the number of Dot-pairs in an expression.

$$
\begin{array}{rl}
\mathsf{size}\ (\mathsf{Num}\ w) &= 0 \\
\mathsf{size}\ (\mathsf{Sym}\ s) &= 0 \\
\mathsf{size}\ (\mathsf{Dot}\ x\ y) &= 1 + \mathsf{size}\ x + \mathsf{size}\ y
\end{array}
$$

[1] Number six was chosen since six is sufficient and suits the x86 implementation best.

The specification of cons guarantees that its implementation will always succeed as long as the number of reachable Dot-pairs is less than the capacity of the heap, i.e. less than l. This precondition under approximates pointer aliasing.

$$\begin{array}{l}\mathsf{size}\ v_1 + \mathsf{size}\ v_2 + \mathsf{size}\ v_3 + \mathsf{size}\ v_4 + \mathsf{size}\ v_5 + \mathsf{size}\ v_6 < l\ \Rightarrow \\ \{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\ \ p : \texttt{E50A3018 E50A4014 E50A5010 E50A600C ... E51A8004 E51A7008} \\ \{\ \mathsf{lisp}\ (\mathsf{cons}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 332)\ \}\end{array}$$

The implementation of cons includes a copying collector which implements Cheney's algorithm [2]. This copying collector requires the heap to be split into two heap halves of equal size; only one of which is used for heap data at any one point in time. When a collection request is issued, all live elements from the currently used heap half are copied over to the currently unused heap half. The proof of cons is outlined in the first author's PhD thesis [14].

The fact that one half of the heap is left empty might seem to be a waste of space. However, the other heap half need not be left completely unused, as the implementation of equal can make use of it. The LISP primitive equal tests whether two s-expressions are structurally identical by traversing the expression tree as a normal recursive procedure. This recursive traversal requires a stack, but the stack can in this case be built inside the unused heap half as the garbage collector will not be called during the execution of equal. Thus, the implementation of equal uses no external stack and requires no conditions on the size of the expressions v_1 and v_2, as their depths cannot exceed the length of a heap half.

$$\begin{array}{l}\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\ \ p : \texttt{E1530004 03A0300F 0A000025 E50A4014 ... E51A7008 E51A8004} \\ \{\ \mathsf{lisp}\ (\mathsf{equal}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 164)\ \}\end{array}$$

4 Compiling s-expression functions to machine code

The previous sections described the theorems which state that certain machine instructions execute LISP primitives. These theorems can be used to augment the input-language understood by a proof-producing compiler that we have developed [16]. The theorems mentioned above allow the compiler to accept:

$$\begin{array}{l}\mathsf{let}\ v_2 = \mathsf{car}\ v_1\ \mathsf{in}\ ... \\ \mathsf{let}\ v_1 = \mathsf{plus}\ v_1\ v_2\ \mathsf{in}\ ... \\ \mathsf{let}\ v_1 = \mathsf{cons}\ v_1\ v_2\ \mathsf{in}\ ... \\ \mathsf{let}\ v_1 = \mathsf{equal}\ v_1\ v_2\ \mathsf{in}\ ...\end{array}$$

Theorems for basic tests have also been proved in a similar manner, and can be provided to the compiler. For example, the following theorem shows that ARM instruction `E3330003` assigns boolean value ($v_1 = \mathsf{Sym}\ \texttt{"nil"}$) to status bit z.

$$\begin{array}{l}\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \} \\ \ p : \texttt{E3330003} \\ \{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4) * \mathsf{sz}\ (v_1 = \mathsf{Sym}\ \texttt{"nil"}) * \\ \ \exists n\ c\ v.\ \mathsf{sn}\ n * \mathsf{sc}\ c * \mathsf{sv}\ v\ \}\end{array}$$

The compiler can use such theorems to create branches on the expression assigned to status bits. The above theorem adds support for the if-statement:

$$\mathsf{if}\ \ v_1 = \mathsf{Sym}\ \texttt{"nil"}\ \ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$$

Once the compiler was given sufficient Hoare-triple theorems it could be used to compile functions operating over s-expressions into machine code. An example will illustrate the process. From the following function

$$\begin{array}{l}\mathsf{sumlist}(v_1, v_2, v_3) = \mathsf{if}\ v_1 = \mathsf{Sym}\ \texttt{"nil"}\ \mathsf{then}\ (v_1, v_2, v_3)\ \mathsf{else}\\ \qquad\qquad\qquad\qquad\quad \mathsf{let}\ v_3 = \mathsf{car}\ v_1\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\quad \mathsf{let}\ v_1 = \mathsf{cdr}\ v_1\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\quad \mathsf{let}\ v_2 = \mathsf{plus}\ v_2\ v_3\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\qquad \mathsf{sumlist}(v_1, v_2, v_3)\end{array}$$

the compiler produces the theorem below, containing the generated ARM machine code and a precondition $\mathsf{sumlist\_pre}(v_1, v_2, v_3)$.

$\mathsf{sumlist\_pre}(v_1, v_2, v_3)\ \Rightarrow$
$\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
p : `E3330003 0A000004 E5935000 E5934004 E0844005 E2444002 EAFFFFF8`
$\{\ \mathsf{let}\ (v_1, v_2, v_3) = \mathsf{sumlist}(v_1, v_2, v_3)\ \mathsf{in}$
$\mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 28) * \mathsf{s}\ \}$

The proof performed by the compiler is outlined in Appendix C, where the precondition $\mathsf{sumlist\_pre}(v_1, v_2, v_3)$ is also defined. The automatically generated pre-functions collect side conditions that must be true for proper execution of the code, e.g. when cons is used the pre-functions collect the requirements on not exceeding the heap limit l.

5 Assembling the LISP evaluator

LISP evaluation was defined as a large tail-recursive function lisp_eval and then compiled, to ARM, PowerPC and x86, to produce theorems of the following form. The theorem below states that the generated ARM code executes lisp_eval for inputs that do not violate any of the side conditions gathered in lisp_eval_pre.

$\mathsf{lisp\_eval\_pre}(v_1, v_2, v_3, v_4, v_5, v_6, l)\ \Rightarrow$
$\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
p : `E3360003 1A0001D1 E3A0600F E3130001 0A000009 ... EAFFF85D`
$\{\ \mathsf{lisp}\ (\mathsf{lisp\_eval}(v_1, v_2, v_3, v_4, v_5, v_6, l)) * \mathsf{pc}\ (p + 7816) * \mathsf{s}\ \}$

lisp_eval evaluates the expression stored in v_1, input v_6 is a list of symbol-value pairs against which symbols in v_1 are evaluated, inputs v_2, v_3, v_4 and v_5 are used as temporaries that are to be initialised with Sym `"nil"`. The heap limit l had to be passed into lisp_eval due to an implementation restriction which requires lisp_eval_pre to input the same variables as lisp_eval. The side condition lisp_eval_pre uses l to state restrictions on applications of cons.

6 Evaluator implements McCarthy's LISP 1.5

The previous sections, and Appendix C, described how a function lisp_eval was compiled down to machine code. The compiler generated some code and derived a theorem which states that the generated code correctly implements lisp_eval. However, the compiler does not (and cannot) give any evidence that lisp_eval in fact implements 'LISP evaluation'. The definition of lisp_eval is long and full of tedious details of how the intermediate stack is maintained and used, and thus it is far from obvious that lisp_eval corresponds to 'LISP evaluation'.

In order to gain confidence that the generated machine code actually implements LISP evaluation, we proved that lisp_eval implements a clean relational semantics of LISP 1.5 [6]. Our relational semantics of LISP 1.5 is defined in terms of three mutually recursive relations $\rightarrow_{eval}$, $\rightarrow_{eval\_list}$ and $\rightarrow_{app}$. Here $(fn, [arg_1; \cdots; arg_n], \rho) \rightarrow_{app} s$ means that $fn[arg_1; \cdots; arg_n] = s$ if the free variables in fn have values specified by an environment ρ; similarly $(e, \rho) \rightarrow_{eval} s$ holds if term e evaluates to s-expression s with respect to environment ρ; and $(el, \rho) \rightarrow_{eval\_list} sl$ holds if list el of expressions evaluates to list sl of expressions with respect to ρ. Here k denotes built-in function names and c constants. For details refer to Gordon [6] and Appendix A in Myreen [14].

$$\frac{\mathsf{ok\_name}\ v}{(v, \rho) \rightarrow_{eval} \rho(v)} \qquad \frac{}{(c, \rho) \rightarrow_{eval} c} \qquad \frac{}{([\,], \rho) \rightarrow_{eval} \texttt{nil}}$$

$$\frac{(p, \rho) \rightarrow_{eval} \texttt{nil} \wedge ([gl], \rho) \rightarrow_{eval} s}{([p \rightarrow e; gl], \rho) \rightarrow_{eval} s} \qquad \frac{(p, \rho) \rightarrow_{eval} x \wedge x \neq \texttt{nil} \wedge (e, \rho) \rightarrow_{eval} s}{([p \rightarrow e; gl], \rho) \rightarrow_{eval} s}$$

$$\frac{\mathsf{can\_apply}\ k\ args}{(k, args, \rho) \rightarrow_{app} k\ args} \qquad \frac{(\rho(f), args, \rho) \rightarrow_{app} s \wedge \mathsf{ok\_name}\ f}{(f, args, \rho) \rightarrow_{app} s}$$

$$\frac{(e, \rho[args/vars]) \rightarrow_{eval} s}{(\lambda[[vars]; e], args, \rho) \rightarrow_{app} s} \qquad \frac{(fn, args, \rho[fn/x]) \rightarrow_{app} s}{(label[[x]; fn], args, \rho) \rightarrow_{app} s}$$

$$\frac{}{([\,], \rho) \rightarrow_{eval\_list} [\,]} \qquad \frac{(e, \rho) \rightarrow_{eval} s \wedge ([el], \rho) \rightarrow_{eval\_list} sl}{([e; el], \rho) \rightarrow_{eval\_list} [s; sl]}$$

We have proved that whenever the relation for LISP 1.5 evaluation $\rightarrow_{eval}$ relates expression s under environment ρ to expression r, then lisp_eval will do the same. Here t and u are translation functions, from one form of s-expressions to another. Let $\mathsf{nil} = \mathsf{Sym}\ \texttt{"nil"}$ and $\mathsf{fst}\ (x, y, \ldots) = x$.

$$\forall s\ \rho\ r.\quad (s, \rho) \rightarrow_{eval} r \quad \Rightarrow \quad \mathsf{fst}\ (\mathsf{lisp\_eval}\ (t\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, u\ \rho, \mathsf{nil}, l)) = t\ r$$

7 Verified parser and printer

Sections 4 and 5 explained how machine code was generated and proved to implement a function called lisp_eval. The precondition of the certificate theorem requires the initial state to satisfy a complex heap invariant lisp. How do we know

that this precondition is not accidentally equivalent to false, making the theorem vacuously true? To remedy this shortcoming, we have verified machine code that will set-up an appropriate state from scratch.

The set-up and tear-down code includes a parser and printer that will, respectively, read in an input s-expression and print out the resulting s-expression. The development of the parser and printer started by first defining a function sexp2string which lays down how s-expressions are to be represented in string form (Appendix D). Then a function string2sexp was defined for which we proved:

$$\forall s.\ \ \mathsf{sexp\_ok}\ s\ \Rightarrow\ \mathsf{string2sexp}\ (\mathsf{sexp2string}\ s) = s$$

Here sexp_ok s makes sure that s does not contain symbols that print ambiguously, e.g. Sym "", Sym "(" and Sym "2". The parsing function was defined as a composition of a lexer sexp_lex and a token parser sexp_parse (Appendix D).

$$\mathsf{string2sexp}\ str = \mathsf{car}\ (\mathsf{sexp\_parse}\ (\mathsf{reverse}\ (\mathsf{sexp\_lex}\ str))\ (\mathsf{Sym\ "nil"})\ [])$$

Machine code was written and verified based on the high-level functions sexp_lex, sexp_parse and sexp2string. Writing these high-level definitions first was a great help when constructing the machine code (using the compiler from [16]).

The overall theorems about our LISP implementations are of the following form. If $\rightarrow_{eval}$ relates s with r under the empty environment (i.e. $(s, []) \rightarrow_{eval} r$), no illegal symbols are used (i.e. sexp_ok $(t\ s)$), running lisp_eval on $t\ s$ will not run out of memory (i.e. lisp_eval_pre$(t\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, l)$), the string representation of $t\ s$ is in memory (i.e. string a (sexp2string $(t\ s)$)), and there is enough space to parse $t\ s$ and set up a heap of size l (i.e. enough_space $(t\ s)\ l$), then the code will execute successfully and terminate with the string representation of $t\ r$ stored in memory (i.e. string a (sexp2string $(t\ r)$)). The ARM code expects the address of the input string to be in register 3, i.e. r3 a.

$$\begin{array}{l}
\forall s\ r\ l\ p.\\
\quad (s, []) \rightarrow_{eval} r \wedge \mathsf{sexp_ok}\ (t\ s) \wedge \mathsf{lisp_eval_pre}(t\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, l) \Rightarrow\\
\quad \{\ \exists a.\ \mathsf{r3}\ a * \mathsf{string}\ a\ (\mathsf{sexp2string}\ (t\ s)) * \mathsf{enough_space}\ (t\ s)\ l * \mathsf{pc}\ p\ \}\\
\quad p : \ldots \text{ code not shown } \ldots\\
\quad \{\ \exists a.\ \mathsf{r3}\ a * \mathsf{string}\ a\ (\mathsf{sexp2string}\ (t\ r)) * \mathsf{enough_space'}\ (t\ s)\ l * \mathsf{pc}\ (p{+}10404)\ \}
\end{array}$$

The input needs to be in register 3 for PowerPC and the eax register for x86.

8 Quantitative data

The idea for this project first arose approximately two years ago. Since then a decompiler [15] and compiler [16] have been developed to aid this project, which produced in total some 4,580 lines of proof automation and 16,130 lines of interactive proofs and definitions, excluding the definitions of the instruction set models [5, 9, 18]. Running through all of the proofs takes approximately 2.5 hours in HOL4 using PolyML.

The verified LISP implementations seem to have reasonable execution times: the `pascal-triangle` example, from Section 1, executes on a 2.4 GHz x86 processor in less than 1 millisecond and on a 67 MHz ARM processor in approximately 90 milliseconds. The PowerPC implementations have not yet been tested on real hardware. The ARM implementation is 2,601 instructions long (10,404 bytes), x86 is 3,135 instructions (9,054 bytes) and the PowerPC implementation consists of 2,929 instructions (11,716 bytes).

9 Discussion of related work

This project has produced trustworthy implementations of LISP. The VLISP project by Guttman et al. [7] shared our goal, but differed in many other aspects. For example, the VLISP project implemented a larger LISP dialect, namely Scheme, and emphasised rigour, not full formality:

> "The verification was intended to be rigorous, but not completely formal, much in the style of ordinary mathematical discourse. Our goal was to verify the algorithms and data types used in the implementation, not their embodiment in the code."

The VLISP project developed an implementation which translates Scheme programs into byte code that is then run on a rigorously verified interpreter. Much like our project, the VLISP project developed their interpreter in a subset of the source language: for them PreScheme, and for us, the input language of our augmented compiler, Section 4.

Work that aims to implement functional languages, in a formally verified manner, include Pike et al. [17] on a certifying compiler from Cryptol (a dialect of Haskell) to AAMP7 code; Dargaye and Leroy [4] on a certified compiler from mini-ML to PowerPC assembly; Li and Slind's work [10] on a certifying compiler from a subset of HOL4 to ARM assembly; and also Chlipala's certified compiler [3] from the lambda calculus to an invented assembly language. The above work either assumes that the environment implements run-time memory management correctly [3, 4] or restricts the input language to a degree where no run-time memory management is needed [10, 17]. It seems that none of the above have made use of (the now large number of) verified garbage collectors (e.g. McCreight et al. [13] have been performing correctness proofs for increasingly sophisticated garbage collectors).

The parser and printer proofs, in Section 7, involved verifying implementations of string-copy, -length, -compare etc., bearing some resemblance to pioneering work by Boyer and Yu [1] on verification of machine code. They verified Motorola MC68020 code implementing a library of string functions.

Acknowledgements. We thank Anthony Fox, Xavier Leroy and Susmit Sarkar et al. for allowing us to use their processor models for this work [5, 9, 18]. We also thank Thomas Tuerk, Joe Hurd, Konrad Slind and John Matthews for comments and discussions. We are grateful for funding from EPSRC, UK.

References

1. Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.
2. C. J. Cheney. A non-recursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.
3. Adam J. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Programming Language Design and Implementation (PLDI)*, pages 54–65. ACM, 2007.
4. Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 211–225. Springer, 2007.
5. Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *LNCS*. Springer, 2003.
6. Mike Gordon. Defining a LISP interpreter in a logic of total functions. In *the ACL2 Theorem Prover and Its Applications (ACL2)*, 2007.
7. Joshua Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.
8. Matt Kaufmann and J. Strother Moore. An ACL2 tutorial. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 17–21. Springer, 2008.
9. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.
10. 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.
11. Panagiotis Manolios and J. Strother Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.
12. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, 1966.
13. Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 468–479. ACM, 2007.
14. Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.
15. 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)*. IEEE, 2008.
16. Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In *Compiler Construction (CC)*, LNCS. Springer, 2009.
17. Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In Panagiotis Manolios and Matthew Wilding, editors, *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*. HappyJack Books, 2006.
18. Susmit Sarkar, Pater Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC

multiprocessor machine code. In *Principles of Programming Languages (POPL)*. ACM, 2009.
19. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 28–32. Springer, 2008.

A Definition of `lisp_inv` in HOL4

The definition of the main invariant of the LISP state.

```
ALIGNED a = (a && 3w = 0w)

string_mem "" (a,m,dm) = T
string_mem (STRING c s) (a,m,df) = a ∈ dm ∧
                                   (m a = n2w (ORD c)) ∧ string_mem s (a+1w,m,dm)

symbol_table [] x (a,dm,m,dg,g) = (m a = 0w) ∧ a ∈ dm ∧ (x = {})
symbol_table (s::xs) x (a,dm,m,dg,g) = (s ≠ "") ∧ ¬ MEM s xs ∧
  (m a = n2w (string_size s)) ∧ {a; a+4w} ⊆ dm ∧ ((a,s) ∈ x) ∧
    let a' = a + n2w (8 + (string_size s + 3) DIV 4 * 4) in
      a < a' ∧ (m (a+4w) = a') ∧ string_mem s (a+8w,g,dg) ∧
      symbol_table xs (x - {(a,s)}) (a',dm,m,dg,g)

builtin =
  ["nil"; "t"; "quote"; "+"; "-"; "*"; "div"; "mod"; "<"; "car"; "cdr";
   "cons"; "equal"; "cond"; "atomp"; "consp"; "numberp"; "symbolp"; "lambda"]

lisp_symbol_table sym (a,dm,m,dg,g) =
  ∃syms. symbol_table (builtin ++ syms) { (b,s) | (b-a,s) ∈ sym } (a,dm,m,dg,g)

lisp_x (Num k) (a,dm,m) sym = (a = n2w (k * 4 + 2)) ∧ k < 2 ** 30
lisp_x (Sym s) (a,dm,m) sym = ALIGNED (a - 3w) ∧ (a - 3w,s) ∈ sym
lisp_x (Dot x y) (a,dm,m) sym = lisp_x x (m a,dm,m) sym ∧ a ∈ dm ∧ ALIGNED a ∧
                                lisp_x y (m (a+4w),dm,m) sym

ref_set a f = {a + 4w * n2w i | i < 2 * f + 4} ∪ {a - 4w * n2w i | i ≤ 8}

ch_active_set (a,i,e) = { a + 8w * n2w j | i ≤ j ∧ j < e }

ok_data w d = if ALIGNED w then w ∈ d else ¬(ALIGNED (w - 1w))

lisp_inv (t1,t2,t3,t4,t5,t6,l) (w1,w2,w3,w4,w5,w6,a,(dm,m),sym,(dh,h),(dg,g)) =
  ∃i u.
    let v = if u then 1 + l else 1 in
    let d = ch_active_set (a,v,i) in
      32 ≤ w2n a ∧ w2n a + 2 * 8 * l + 20 < 2 ** 32 ∧ l ≠ 0 ∧
      (m a = a + n2w (8 * i)) ∧ ALIGNED a ∧ v ≤ i ∧ i ≤ v + l ∧
      (m (a + 4w) = a + n2w (8 * (v + l))) ∧
      (m (a - 28w) = if u then 0w else 1w) ∧
      (m (a - 32w) = n2w (8 * l)) ∧ (dm = ref_set a (l + l + 1)) ∧
      lisp_symbol_table sym (a + 16w * n2w l + 24w,dh,h,dg,g) ∧
      lisp_x t1 (w1,d,m) sym ∧ lisp_x t2 (w2,d,m) sym ∧ lisp_x t3 (w3,d,m) sym ∧
      lisp_x t4 (w4,d,m) sym ∧ lisp_x t5 (w5,d,m) sym ∧ lisp_x t6 (w6,d,m) sym ∧
      ∀w. w ∈ d ⇒ ok_data (m w) d ∧ ok_data (m (w + 4w)) d
```

B Sample verification proof of 'car' primitive

The verification proofs of the primitive LISP operations build on lemmas about lisp_inv. The following lemma is used in the proof of the theorem about car

described in Section 3.1. This lemma can be read as saying that, if lisp_inv relates x_1 to Dot-pair v_1, then x_1 is a word-aligned address into memory segment m, and an assignment of car v_1 to v_2 corresponds to replacing x_2 with the value of memory m at address x_1, i.e. $m(x_1)$.

$$
\begin{array}{l}
(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1) \wedge \\
\mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m, m_2, m_3)\ \Rightarrow \\
(x_1\ \&\ 3 = 0) \wedge x_1 \in \mathsf{domain}\ m \wedge \\
\mathsf{lisp\_inv}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l)\ (x_1, m(x_1), x_3, x_4, x_5, x_6, a, m, m_2, m_3)
\end{array}
$$

One of our tools derives the following Hoare triple theorem for the ARM instruction that is to be verified: `ldr r4,[r3]` (encoded as `E5934000`).

$$
\begin{array}{l}
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{m}\ m * \mathsf{pc}\ p * \langle (r_3\ \&\ 3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle\ \} \\
\ p : \texttt{E5934000} \\
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ m(r_3) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4)\ \}
\end{array}
$$

Application of the frame rule (shown in Appendix C) produces:

$$
\begin{array}{l}
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{m}\ m * \mathsf{pc}\ p * \langle (r_3\ \&\ 3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle * \\
\ \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\ \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (r_3, r_4, x_3, x_4, x_5, x_6, a, m, m_2, m_3) \rangle \} \\
\ p : \texttt{E5934000} \\
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ m(r_3) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) * \langle (r_3\ \&\ 3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle * \\
\ \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\ \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (r_3, r_4, x_3, x_4, x_5, x_6, a, m, m_2, m_3) \rangle \}
\end{array}
$$

Now the postcondition can be weakened to the desired expression:

$$
\begin{array}{l}
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{m}\ m * \mathsf{pc}\ p * \langle (r_3\ \&\ 3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle * \\
\ \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\ \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (r_3, r_4, x_3, x_4, x_5, x_6, a, m, m_2, m_3) \rangle \} \\
\ p : \texttt{E5934000} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}
\end{array}
$$

Since variables r_3, r_4, x_3, x_4, x_5, x_6, m, m_2, m_3 do not appear in the postcondition, they can be existentially quantified in the precondition, which then strengthens as follows:

$$
\begin{array}{l}
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \langle \exists x\ y.\ \mathsf{Dot}\ x\ y = v_1 \rangle\ \} \\
\ p : \texttt{E5934000} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}
\end{array}
$$

The specification for car follows by moving the boolean condition:

$$
\begin{array}{l}
(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \Rightarrow \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\ p : \texttt{E5934000} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}
\end{array}
$$

All of the primitive LISP operations were verified in the same manner. For the HOL4 implementation, a 50-line ML program was written to automate these proofs given the appropriate lemmas about lisp_inv.

C Proof performed by compiler

Internally the compiler runs through a short proof when constructing the theorem presented in Section 4. This proof makes use of the following five proof rules derived from the definition of our machine-code Hoare triple, developed in previous work [15]. Formal definitions and detailed explanations are given in the first author's PhD thesis [14]. Here $\cup$ is simply set union.

frame:	$\{p\}\ c\ \{q\}\ \Rightarrow\ \forall r.\ \{p * r\}\ c\ \{q * r\}$
code extension:	$\{p\}\ c\ \{q\}\ \Rightarrow\ \forall d.\ \{p\}\ c \cup d\ \{q\}$
composition:	$\{p\}\ c\ \{q\}\ \wedge\ \{q\}\ d\ \{r\}\ \Rightarrow\ \{p\}\ c \cup d\ \{r\}$
move pure:	$\{p * \langle b \rangle\}\ c\ \{q\}\ =\ (b \Rightarrow \{p\}\ c\ \{q\})$
tail recursion:	$(\forall x.\ P(x) \wedge G(x) \Rightarrow \{p(x)\}\ c\ \{p(F(x))\}) \wedge$ $(\forall x.\ P(x) \wedge \neg G(x) \Rightarrow \{p(x)\}\ c\ \{q(D(x))\})\ \Rightarrow$ $(\forall x.\ \mathsf{pre}(G, F, P)(x) \Rightarrow \{p(x)\}\ c\ \{q(\mathsf{tailrec}(G, F, D)(x))\})$

The last rule mentions tailrec and pre, which are functions that satisfy:

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

$$\forall x.\quad \mathsf{pre}(G, F, P)(x) = \mathsf{if}\ G(x)\ \mathsf{then}\ \mathsf{pre}(G, F, P)(F(x)) \wedge P(x)\ \mathsf{else}\ P(x)$$

Note that any tail-recursive function can be defined as an instance of tailrec, introduced using a trick by Manolios and Moore [11]. Another noteworthy feature: if $\mathsf{pre}(G, F, P)(x)$ is true then $\mathsf{tailrec}(G, F, D)$ terminates for input x.

The compiler starts its proof from the following theorems describing the test $v_1 = \mathsf{Sym}\ \texttt{"nil"}$ as well as operations car, cdr and plus.

1. $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   $p : \texttt{E3330003}$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4) * \mathsf{sz}\ (v_1 = \mathsf{Sym}\ \texttt{"nil"}) *$
   $\exists n\ c\ v.\ \mathsf{sn}\ n * \mathsf{sc}\ c * \mathsf{sv}\ v\ \}$
2. $(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   $p : \texttt{E5935000}$
   $\{\ \mathsf{lisp}\ (v_1, v_2, \mathsf{car}\ v_1, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}$
3. $(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   $p : \texttt{E5933004}$
   $\{\ \mathsf{lisp}\ (\mathsf{cdr}\ v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}$
4. $(\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = v_3 \wedge m{+}n < 2^{30})\ \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   $p : \texttt{E0844005 E2444002}$
   $\{\ \mathsf{lisp}\ (v_1, \mathsf{plus}\ v_2\ v_3, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}$

The compiler next generates two branches to glue the code together; the branch instructions have the following specifications:

5. $\{\ \mathsf{pc}\ p * \mathsf{sz}\ z * \langle z \rangle\ \}\ p : \texttt{0A000004}\ \{\ \mathsf{pc}\ (p+24) * \mathsf{sz}\ z\ \}$
6. $\{\ \mathsf{pc}\ p * \mathsf{sz}\ z * \langle \neg z \rangle\ \}\ p : \texttt{0A000004}\ \{\ \mathsf{pc}\ (p+4) * \mathsf{sz}\ z\ \}$
7. $\{\ \mathsf{pc}\ p\ \}\ p : \texttt{EAFFFFF8}\ \{\ \mathsf{pc}\ (p-24)\ \}$

The specifications above are collapsed into theorems describing one pass through the code by composing 1,5 and 1,6,2,3,4,7, which results in:

8. $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s} * \langle v_1 = \mathsf{Sym}\ \texttt{"nil"} \rangle\ \}$
   $p : \texttt{E3330003 0A000004}$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+28) * \mathsf{s}\ \}$
9. $(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1) \wedge$
   $(\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = \mathsf{car}\ v_1 \wedge m{+}n < 2^{30}) \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s} * \langle v_1 \neq \mathsf{Sym}\ \texttt{"nil"} \rangle\ \}$
   $p : \texttt{E3330003 0A000004 E5935000 E5934004 E0844005 E2444002 EAFFFFF8}$
   $\{\ \mathsf{lisp}\ (\mathsf{cdr}\ v_1, \mathsf{plus}\ v_2\ (\mathsf{car}\ v_1, l), \mathsf{car}\ v_1, v_4, v_5, v_6) * \mathsf{pc}\ p * \mathsf{s}\ \}$

Code extension is applied to theorem 8, and then the rule for introducing a tail-recursive function is applied. The compiler produces the following total-correctness specification.

10. $\mathsf{sumlist\_pre}(v_1, v_2, v_3) \Rightarrow$
    $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
    $p : \texttt{E3330003 0A000004 E5935000 E5934004 E0844005 E2444002 EAFFFFF8}$
    $\{\ \mathsf{let}\ (v_1, v_2, v_3) = \mathsf{sumlist}(v_1, v_2, v_3)\ \mathsf{in}$
      $\mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+28) * \mathsf{s}\ \}$

Here sumlist is defined as an instance of tailrec, and sumlist_pre is an instance of pre. The compiler exports sumlist_pre as the following recursive function which collects all of the side conditions that must hold for proper execution of the code:

$\mathsf{sumlist\_pre}(v_1, v_2, v_3) =$
  $\mathsf{if}\ v_1 = \mathsf{Sym}\ \texttt{"nil"}\ \mathsf{then\ true\ else}$
    $\mathsf{let}\ \mathit{cond} = (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \mathsf{in}$
    $\mathsf{let}\ v_3 = \mathsf{car}\ v_1\ \mathsf{in}$
    $\mathsf{let}\ \mathit{cond} = \mathit{cond} \wedge (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \mathsf{in}$
    $\mathsf{let}\ v_1 = \mathsf{cdr}\ v_1\ \mathsf{in}$
    $\mathsf{let}\ \mathit{cond} = \mathit{cond} \wedge (\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = v_3 \wedge m{+}n < 2^{30})\ \mathsf{in}$
    $\mathsf{let}\ v_2 = \mathsf{plus}\ v_2\ v_3\ \mathsf{in}$
      $\mathsf{sumlist\_pre}(v_1, v_2, v_3) \wedge \mathit{cond}$

When the loop rule is applied above, its parameters are assigned values:

$$
\begin{array}{rcl}
p & = & \lambda(v_1,v_2,v_3).\ \mathsf{lisp}\ (v_1,v_2,v_3,v_4,v_5,v_6,l) * \mathsf{pc}\ p * \mathsf{s} \\
q & = & \lambda(v_1,v_2,v_3).\ \mathsf{lisp}\ (v_1,v_2,v_3,v_4,v_5,v_6,l) * \mathsf{pc}\ (p+28) * \mathsf{s} \\
G & = & \lambda(v_1,v_2,v_3).\ v_1 \neq \mathsf{Sym}\ \texttt{"nil"} \\
F & = & \lambda(v_1,v_2,v_3).\ (\mathsf{cdr}\ v_1, \mathsf{plus}\ v_2\ (\mathsf{car}\ v_1, l), \mathsf{car}\ v_1) \\
D & = & \lambda(v_1,v_2,v_3).\ (v_1,v_2,v_3) \\
P & = & \lambda(v_1,v_2,v_3).\ (v_1 \neq \mathsf{Sym}\ \texttt{"nil"}) \Rightarrow \\
 & & \quad (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1) \wedge \\
 & & \quad (\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = \mathsf{car}\ v_1 \wedge m{+}n < 2^{30})
\end{array}
$$

D Definition of s-expression printing and parsing

Our machine code for printing LISP s-expressions implements sexp2string.

$$
\begin{array}{rcl}
\mathsf{sexp2string}\ x & = & \mathsf{aux}\ (x, \mathsf{T}) \\
\mathsf{aux}\ (\mathsf{Num}\ n, b) & = & \mathsf{num2str}\ n \\
\mathsf{aux}\ (\mathsf{Sym}\ s, b) & = & s \\
\mathsf{aux}\ (\mathsf{Dot}\ x\ y, b) & = & \mathsf{if}\ \mathsf{isQuote}\ (\mathsf{Dot}\ x\ y) \wedge b\ \mathsf{then}\ \texttt{"'"} +\!\!+ \mathsf{aux}\ (\mathsf{car}\ y, \mathsf{T})\ \mathsf{else} \\
 & & \quad \mathsf{let}\ (a,e) = (\mathsf{if}\ b\ \mathsf{then}\ (\texttt{"("}, \texttt{")"})\ \mathsf{else}\ (\texttt{""}, \texttt{""}))\ \mathsf{in} \\
 & & \quad\ \mathsf{if}\ y = \mathsf{Sym}\ \texttt{"nil"}\ \mathsf{then}\ a +\!\!+ \mathsf{aux}\ (x, \mathsf{T}) +\!\!+ e\ \mathsf{else} \\
 & & \quad\ \mathsf{if}\ \mathsf{isDot}\ y\ \mathsf{then}\ a +\!\!+ \mathsf{aux}\ (x, \mathsf{T}) +\!\!+ \texttt{" "} +\!\!+ \mathsf{aux}\ (y, \mathsf{F}) +\!\!+ e \\
 & & \quad\quad\ \mathsf{else}\ a +\!\!+ \mathsf{aux}\ (x, \mathsf{T}) +\!\!+ \texttt{" . "} +\!\!+ \mathsf{aux}\ (y, \mathsf{F}) +\!\!+ e \\
\mathsf{isDot}\ x & = & \exists y\ z.\ x = \mathsf{Dot}\ y\ z \\
\mathsf{isQuote}\ x & = & \exists y.\ x = \mathsf{Dot}\ (\mathsf{Sym}\ \texttt{"quote"})\ (\mathsf{Dot}\ y\ (\mathsf{Sym}\ \texttt{"nil"}))
\end{array}
$$

Parsing is defined as the follows. Here reverse is normal list reversal.

$$\mathsf{string2sexp}\ str = \mathsf{car}\ (\mathsf{sexp\_parse}\ (\mathsf{reverse}\ (\mathsf{sexp\_lex}\ str))\ (\mathsf{Sym}\ \texttt{"nil"})\ [])$$

The lexing function sexp_lex splits a string into a list of strings, e.g.

$$\mathsf{sexp\_lex}\ \texttt{"(car ('23 . y))"} = [\texttt{"("}, \texttt{"car"}, \texttt{"("}, \texttt{"'"}, \texttt{"23"}, \texttt{"."}, \texttt{"y"}, \texttt{")"}, \texttt{")"}]$$

Token parsing is defined as:

$$
\begin{array}{rcl}
\mathsf{sexp\_parse}\ []\ exp\ stack & = & exp \\
\mathsf{sexp\_parse}\ (\texttt{")"} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Sym}\ \texttt{"nil"})\ (exp :: stack) \\
\mathsf{sexp\_parse}\ (\texttt{"("} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Dot}\ exp\ (\mathsf{head}\ stack))\ (\mathsf{tail}\ stack) \\
\mathsf{sexp\_parse}\ (\texttt{"."} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{car}\ exp)\ stack \\
\mathsf{sexp\_parse}\ (\texttt{"'"} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Dot}\ (\mathsf{Dot}\ (\mathsf{Sym}\ \texttt{"quote"}) \\
 & & \quad (\mathsf{Dot}\ (\mathsf{car}\ exp)\ (\mathsf{Sym}\ \texttt{"nil"})))\ (\mathsf{cdr}\ exp))\ stack \\
\mathsf{sexp\_parse}\ (t :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Dot}\ (\mathsf{if}\ \mathsf{is\_num}\ t\ \mathsf{then} \\
 & & \quad \mathsf{Num}\ (\mathsf{str2num}\ t)\ \mathsf{else}\ \mathsf{Sym}\ t)\ exp)\ stack
\end{array}
$$