A verified runtime for a verified theorem prover

Magnus O. Myreen[1] and Jared Davis[2]

[1] Computer Laboratory, University of Cambridge, UK
[2] Centaur Technology, Inc., Austin TX, USA

Abstract. Theorem provers, such as ACL2, HOL, Isabelle and Coq, rely on the correctness of runtime systems for programming languages like ML, OCaml or Common Lisp. These runtime systems are complex and critical to the integrity of the theorem provers.

In this paper, we present a new Lisp runtime which has been formally verified and can run the Milawa theorem prover. Our runtime consists of 7,500 lines of machine code and is able to complete a 4 gigabyte Milawa proof effort. When our runtime is used to carry out Milawa proofs, less unverified code must be trusted than with any other theorem prover.

Our runtime includes a just-in-time compiler, a copying garbage collector, a parser and a printer, all of which are HOL4-verified down to the concrete x86 code. We make heavy use of our previously developed tools for machine-code verification. This work demonstrates that our approach to machine-code verification scales to non-trivial applications.

1 Introduction

We can never be sure [6] a computer has executed a theorem prover (or any other program) correctly. Even if we could prove a processor design implements its instruction set, we have no way to ensure it will be manufactured correctly and will not be interfered with as it runs. But can we develop a theorem prover for which there are no other reasonable doubts?

Any theorem prover is based on a formal mathematical logic. Logical soundness is well-studied. It is usually established with social proofs, but some soundness proofs [20, 10] have even been checked by computers. If we accept the logic is sound, the question boils down to whether the theorem prover is *faithful* to its logic: does it only claim to prove formulas that are indeed theorems?

In many theorem provers, the trusted core—the code that must be right to ensure faithfulness—is quite small. As examples, HOL Light [12] is an LCF-style system whose trusted core is 400 lines of Objective Caml, and Milawa [5] is a Boyer-Moore style prover whose trusted core is 2,000 lines of Common Lisp. These cores are so simple we may be able to prove their faithfulness socially, or perhaps even mechanically as Harrison [11] did for HOL Light.

On the other hand, to actually use these theorem provers we need a runtime environment that can parse source code, infer types, compile functions, collect garbage, and so forth. These runtimes are far more complicated than simple

theorem-prover cores. For a rough perspective, source-code distributions of Objective Caml and Common Lisp systems seem to range from 15 MB to 50 MB on disk, and also require C compilers and various libraries.

In this paper, we present *Jitawa*, the first mechanically verified runtime designed to run a general-purpose theorem prover.

- We target the Milawa theorem prover, so we begin with a brief description of this system and explain how using a verified runtime increases our level of trust in Milawa proofs. (Section 2)
- To motivate the design of our runtime, we examine Milawa's computational and I/O needs. To meet these needs, Jitawa features efficient parsing, just-in-time compilation to 64-bit x86 machine code, garbage collection, expression printing, and an "abort with error message" capability. (Section 3)
- We consider what it means for Jitawa to be correct. We develop a formal HOL4 [21] specification (400 lines) of how the runtime should operate. This covers expression evaluation, parsing, and printing. (Section 4)
- We explain how Jitawa is implemented and verified. We build heavily on our previous tools for machine-code synthesis and verification, so in this paper we focus on how our compiler is designed and specified and also on how I/O is handled. We present the top-level correctness theorem that shows Jitawa's machine code implements its specification. (Section 5)
- We describe the relationship between Milawa and Jitawa. We have used Jitawa to carry out a 4 GB proof effort in Milawa, demonstrating the good capacity of the runtime. We explain the informal nature of this connection, and how we hope it may be formalized in future work. (Section 6)

We still need *some* unverified code. We have not tried to avoid using an operating system, and we use a C wrapper-program to interact with it. This C program is quite modest: it uses `malloc` to allocate memory and invokes our runtime. Jitawa also performs I/O by making calls to C functions for reading and writing standard input and output (Section 5.3).

2 The Milawa system

Milawa [5] is a theorem prover styled after systems like NQTHM [1] and ACL2 [13]. The Milawa logic has three kinds of objects: natural numbers, symbols, and conses. It also has twelve primitive functions like if, equal, cons, and +, and eleven macros like list, and, let*, and cond. Starting from these primitives, one may introduce the definitions of first-order, total, untyped, recursive functions as axioms. For instance, a list-length function might be introduced as

$$\forall \texttt{x}.\ \texttt{(len x)} = \texttt{(if (consp x) (+ '1 (len (cdr x))) '0)}.$$

Almost all of Milawa's source code is written as functions in its logic. We can easily run these functions on a Common Lisp system.

2.1 The trusted core

Milawa's original trusted core is a 2,000 line Common Lisp program that checks a file of *events*. Most commonly,

- `Define` events are used to introduce recursive functions, and include the name of a file that should contain a proof of termination, and
- `Verify` events are used to admit formulas as theorems, and include the name of a file that should contain a proof of the formula.

The user generates these events and proof files ahead of time, with the help of an interactive interface that need not be trusted.

A large part of the trusted core is just a straightforward definition of formulas and proofs in the Milawa logic. A key function is proofp ("proof predicate"), which determines if its argument is a valid Milawa-logic proof; this function only accepts full proofs made up of primitive inferences, and it is defined in the Milawa logic so we can reason about provability.

When the trusted core is first started, proofp is used to check the proofs for each event. But, eventually, the core can be reflectively extended. The steps are:

1. `Define` a new proof-checking function. This function is typically a proper extension of proofp: it still accepts all the primitive proof steps, but it also permits new, non-primitive proof steps.
2. `Verify` that the new function only accepts theorems. That is, whenever the new proof checker accepts a proof of some formula ϕ, there must exist a proof of ϕ that is accepted by proofp.
3. Use the special `Switch` event to instruct the trusted core to begin using the new, now-verified function, instead of proofp, to check proofs.

After such an extension, the proofs for `Define` and `Verify` events may make use of the new kinds of proof steps. These higher-level proofs are usually much shorter than full proofp-style proofs, and can be checked more quickly.

2.2 The verified theorem prover

Milawa's trusted core has no automation for finding proofs. But separately from its core, Milawa includes a Boyer-Moore style theorem prover that can carry out a goal-directed proof search using algorithms like lemma-driven conditional rewriting, calculation, case-splitting into subgoals, and so on.

All of these algorithms are implemented as functions in the Milawa logic. Because of this, we can reason about their behavior using the trusted core. In Milawa's "self-verification" process, the trusted core is used to `Define` each of the functions making up the theorem prover and `Verify` lemmas about their behavior. This process culminates in the definition of a verified proof checker that can apply any sequence of Milawa's tactics as a single proof step. Once we `Switch` to this new proof checker, the trusted core can essentially check proofs by directly running the theorem prover.

2.3 The role of a verified runtime

Through its self-verification process, the Milawa theorem prover is mechanically verified by its trusted core. For this verification to be believed—indeed, for any theorems proven by Milawa to be believed—one must trust that

1. the Milawa logic is sound,
2. the trusted core of Milawa is faithful to its logic, and
3. the computing platform used to carry out the self-verification process has correctly executed Milawa's trusted core.

The first two points are addressed in previous work [5] in a social way. Our verified runtime does not directly bolster these arguments, but may eventually serve as groundwork for a mechanical proof of these claims (Section 6).

The third point requires trusting some computer hardware and a Common Lisp implementation. Unfortunately, these runtimes are always elaborate and are never formally verified. Using Jitawa as our runtime greatly reduces the amount of unverified code that must be trusted.

3 Requirements and design decisions

On the face of it, Milawa is quite modest in what it requires of the underlying Lisp runtime. Most of the code for its trusted core and all of the code for its theorem prover are written as functions in the Milawa logic. These functions operate on just a few predefined data types (natural numbers, symbols, and conses), and involve a handful of primitive functions and macros like `car`, `+`, `list`, and `cond`. To run these functions we just need a basic functional programming language that implements these primitives.

Beyond this, Milawa's original trusted core also includes some Common Lisp code that is outside of the logic. As some examples:

- It destructively updates global variables that store its arity table, list of axioms, list of definitions, and so forth.
- It prints some status messages and timing information so the user can evaluate its progress and performance.
- It can use the underlying Lisp system's checkpointing system to save the program's current state as a new executable.

It was straightforward to develop a new version of the Milawa core that does away with the features mentioned above: we avoid destructive updates by adopting a more functional "state-tuple" style, and simply abandon checkpointing and timing reports since, while convenient, they are not essential.

On the other hand, some other Common Lisp code is not so easy to deal with. In particular:

- It instructs the Common Lisp system to compile user-supplied functions as they are `Defined`, which is important for running new proof checkers.

– It dynamically calls either proofp or whichever proof checker has been most recently installed via `Switch` to check proofs.
– It aborts with a runtime error when invalid events or proofs are encountered, or if an attempt is made to run a Skolem function.

We did not see a good way to avoid any of this. Accordingly, Jitawa must also provide on-the-fly compilation of user-defined functions, dynamic function invocation, and some way to cause runtime errors.

3.1 I/O requirements

In Milawa's original trusted core, each `Define` and `Verify` event includes the name of a file that should contain the necessary proof, and these files are read on demand as each event is processed. For a rough sense of scale, the proof of self-verification is a pretty demanding effort; it includes over 15,000 proof files with a total size of 8 GB.

The proofs in these files—especially the lowest-level proofs that proofp checks—can be very large and repetitive. As a simple but crucial optimization, an abbreviation mechanism [2] lets us reuse parts of formulas and proofs. For instance,

```
(append (cons (cons a b) c)
        (cons (cons a b) c))
```

could be more compactly written using an abbreviation as

```
(append #1=(cons (cons a b) c)
        #1#).
```

We cannot entirely avoid file input since, at some point, we must at least tell the program what we want it to verify. But we would prefer to minimize interaction with the operating system. Accordingly, in our new version of the Milawa core, we do not keep proofs in separate files. Instead, each event directly contains the necessary proof, so we only need to read a single file. This approach exposes additional opportunities for structure sharing. While the original, individual proof files for the bootstrapping process are 8 GB, the new events file is only 4 GB. It has 525 million abbreviations.

At any rate, Jitawa needs to be able to parse input files that are gigabytes in size and involve hundreds of millions of abbreviations.

3.2 Designing for performance and scalability

The real challenge in constructing a practical runtime for Milawa (or any other theorem prover) is that performance and scalability cannot be ignored. Our previously verified Lisp interpreter [18] is hopelessly inadequate: its direct interpreter approach is too slow, and it also has inherent memory limitations that prevent it from handling the large objects the theorem prover must process.

For Jitawa, we started from scratch and made sure the central design decisions allowed our implementation to scale. For instance:

– To improve execution times, functions are just-in-time compiled to native x86 machine code.
– To support large computations, we target 64-bit x86. Jitawa can handle up to 2^{31} live cons cells, i.e., up to 16 GB of conses at 8 bytes per cons.
– Parsing and printing are carefully coded not to use excessive amounts of memory. In particular, lexing is merged with parsing into what is called a scanner-less parser, and abbreviations are supported efficiently.
– Since running out of heap space or stack space is a real concern, we ensure graceful exits in all circumstances and provide helpful error messages when limits are reached.

4 The Jitawa specification

Jitawa implements a read-eval-print loop. Here is an example run, where lines starting with `>` are user input and the others are the output.

```
> '3
3
> (cons '5 '(6 7))
(5 6 7)
> (define 'increment '(n) '(+ n '1))
NIL
> (increment '5)
6
```

What does it mean for Jitawa to be correct? Intuitively, we need to show the input characters are parsed as expected, the parsed terms are evaluated according to our intended semantics, and the results of evaluation are printed as the correct character sequences.

To carry out a proof of correctness, we first need to formalize how parsing, evaluation, and printing are supposed to occur. In this section, we describe our formal, HOL specification of how Jitawa is to operate. This involves defining a term representation and evaluation semantics (Sections 4.1 and 4.2), and specifying how parsing and printing (Section 4.3) are to be done. We combine these pieces into a top-level specification (Section 4.4) for Jitawa.

Altogether, our specification takes 400 lines of HOL code. It is quite abstract: it has nothing to do with the x86 model, compilation, garbage collection, and so on. We eventually (Section 5.4) prove Jitawa's machine code implements this specification, and we regard this as a proof of "Jitawa is correct."

4.1 Syntax

Milawa uses a typical s-expression [15] syntax. While Jitawa's parser has to deal with these expressions at the level of individual characters, it is easier to model

these expressions as a HOL datatype,

$$
\begin{array}{rcll}
\mathit{sexp} & ::= & \mathsf{Val}\ \mathit{num} & \text{(natural numbers)} \\
& | & \mathsf{Sym}\ \mathit{string} & \text{(symbols)} \\
& | & \mathsf{Dot}\ \mathit{sexp}\ \mathit{sexp} & \text{(cons pairs).}
\end{array}
$$

We use the name Dot instead of Cons to distinguish it from the name of the function called Cons which produces this structure; the name Dot is from the syntax `(1 . 2)`. As an example, the *sexp* representation of `(+ n '1)` is

```
Dot (Sym "+")
    (Dot (Sym "N")
         (Dot (Dot (Sym "QUOTE") (Dot (Val 1) (Sym "NIL")))
              (Sym "NIL"))).
```

Our specification also deals with well-formed s-expression, i.e. s-expressions that can be evaluated. We represent these expressions with a separate datatype, called *term*. The *term* representation of `(+ n '1)` is

$$\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Add})\ [\mathsf{Var}\ \texttt{"N"}, \mathsf{Const}\ (\mathsf{Val}\ 1)].$$

The definition of *term* is shown below. Some constructors are marked as macros, meaning they expand into other terms in our semantics and in the compiler, e.g., Cond expands into If (if-then-else) statements. These are the same primitives and macros as in the Milawa theorem prover.

$$
\begin{array}{rcll}
\mathit{term} & ::= & \mathsf{Const}\ \mathit{sexp} & \\
& | & \mathsf{Var}\ \mathit{string} & \\
& | & \mathsf{App}\ \mathit{func}\ (\mathit{term}\ \mathsf{list}) & \\
& | & \mathsf{If}\ \mathit{term}\ \mathit{term}\ \mathit{term} & \\
& | & \mathsf{LambdaApp}\ (\mathit{string}\ \mathsf{list})\ \mathit{term}\ (\mathit{term}\ \mathsf{list}) & \\
& | & \mathsf{Or}\ (\mathit{term}\ \mathsf{list}) & \\
& | & \mathsf{And}\ (\mathit{term}\ \mathsf{list}) & \text{(macro)} \\
& | & \mathsf{List}\ (\mathit{term}\ \mathsf{list}) & \text{(macro)} \\
& | & \mathsf{Let}\ ((\mathit{string} \times \mathit{term})\ \mathsf{list})\ \mathit{term} & \text{(macro)} \\
& | & \mathsf{LetStar}\ ((\mathit{string} \times \mathit{term})\ \mathsf{list})\ \mathit{term} & \text{(macro)} \\
& | & \mathsf{Cond}\ ((\mathit{term} \times \mathit{term})\ \mathsf{list}) & \text{(macro)} \\
& | & \mathsf{First}\ \mathit{term} \mid \mathsf{Second}\ \mathit{term} \mid \mathsf{Third}\ \mathit{term} & \text{(macro)} \\
& | & \mathsf{Fourth}\ \mathit{term} \mid \mathsf{Fifth}\ \mathit{term} & \text{(macro)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{func} & ::= & \mathsf{Define} \mid \mathsf{Print} \mid \mathsf{Error} \mid \mathsf{Funcall} \\
& | & \mathsf{PrimitiveFun}\ \mathit{primitive} \mid \mathsf{Fun}\ \mathit{string}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{primitive} & ::= & \mathsf{Equal} \mid \mathsf{Symbolp} \mid \mathsf{SymbolLess} \\
& | & \mathsf{Consp} \mid \mathsf{Cons} \mid \mathsf{Car} \mid \mathsf{Cdr} \mid \\
& | & \mathsf{Natp} \mid \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Less}
\end{array}
$$

4.2 Evaluation semantics

We define the semantics of expressions as a relation $\xrightarrow{\mathsf{ev}}$ that explains how objects of type *term* evaluate. Following Common Lisp, we separate the store k for functions from the environment *env* for local variables. We model the I/O streams *io* as a pair of strings, one for characters produced as output, and one for characters yet to be read as input. Our evaluation relation $\xrightarrow{\mathsf{ev}}$ explains how terms may be evaluated with respect to some particular k, *env*, and *io* to produce a resulting *sexp* and an updated k' and io'.

As an example, the following rule shows how Var terms are evaluated. We only permit the evaluation of bound variables, i.e. $x \in \mathsf{domain}\ env$.

$$\frac{x \in \mathsf{domain}\ env}{(\mathsf{Var}\ x, env, k, io) \xrightarrow{\mathsf{ev}} (env(x), k, io)}$$

Our evaluation relation is defined inductively with auxilliary relations $\xrightarrow{\mathsf{evl}}$ for evaluating a list of terms and $\xrightarrow{\mathsf{ap}}$ for applying functions. For instance, the following rule explains how a function (i.e., something of type *func*) is applied: first the arguments are evaluated using $\xrightarrow{\mathsf{evl}}$, then the apply relation $\xrightarrow{\mathsf{ap}}$ determines the result of the application.

$$\frac{(args, env, k, io) \xrightarrow{\mathsf{evl}} (vals, k', io') \ \wedge\ (f, vals, env, k', io') \xrightarrow{\mathsf{ap}} (ans, k'', io'')}{(\mathsf{App}\ f\ \ args, env, k, io) \xrightarrow{\mathsf{ev}} (ans, k'', io'')}$$

With regards to just-in-time compilation, an interesting case for the apply relation $\xrightarrow{\mathsf{ap}}$ is the application of user-defined functions. In our semantics, a user-defined function *name* can be applied when it is defined in store k with the right number of parameters.

$$\frac{\begin{array}{c}k(name) = (params, body) \ \wedge\ (\mathsf{length}\ vals = \mathsf{length}\ params) \wedge \\ (body, [params \leftarrow vals], k, io) \xrightarrow{\mathsf{ev}} (ans, k', io') \ \wedge\ name \notin \mathsf{reserved\_names}\end{array}}{(\mathsf{Fun}\ name, vals, env, k, io) \xrightarrow{\mathsf{ap}} (ans, k', io')}$$

Another interesting case is how user-defined functions are introduced. New definitions can be added to k by evaluation of the Define function. We disallow overwriting existing definitions, i.e. $name \notin \mathsf{domain}\ k$.

$$\frac{name \notin \mathsf{domain}\ k}{(\mathsf{Define}, [name, params, body], env, k, io) \xrightarrow{\mathsf{ap}} (\mathsf{nil}, k[name \mapsto (params, body)], io)}$$

In Jitawa's implementation, an application of Define compiles the expression *body* into machine code. Notice how nothing in the above rule requires that it should be possible to evaluate the expression *body* at this stage. In particular, the functions mentioned inside *body* might not even be defined yet. This means that how we compile function calls within *body* depends on the compile-time state: if the function to be called is already defined we can use a direct jump/call to its code, but otherwise we use a slower, dynamic jump/call.

Strictly speaking, Milawa does not require that Define is to be applicable to functions that cannot be evaluated. However, we decided to allow such definitions to keep the semantics clean and simple. Another advantage of allowing compilation of calls to not-yet-defined functions is that we can immediately support mutually recursive definitions, e.g.:

```
(define 'even '(n) (if (equal n '0) 't   (odd  (- n '1))))
(define 'odd  '(n) (if (equal n '0) 'nil (even (- n '1))))
```

When the expression for `even` is compiled, the compiler knows nothing about the function `odd` and must thus insert a dynamic jump to the code for `odd`. But when `odd` is compiled, `even` is already known and the compiler can insert a direct jump to the code for `even`.

4.3 Parsing and printing

Besides evaluation, our runtime must provide parsing and printing. We begin by modeling our parsing and printing algorithms at an abstract level in HOL as two functions, sexp2string and string2sexp, which convert s-expressions into strings and vice versa. The printing function is trivial. Parsing is more complex, but we can gain some assurance our specification is correct by proving it is the inverse of the printing function, i.e.

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

Unfortunately, Jitawa's true parsing algorithm must be slightly more complicated. It must handle the `#1=`-style abbreviations described in Section 3.1. Also, the parser we verified in previous work [18] assumed the entire input string was present in memory, but since Jitawa's input may be gigabytes in size, we instead want to read the input stream incrementally. We define a function,

$$\mathsf{next\_sexp}\ :\ \mathit{string} \rightarrow \mathit{sexp} \times \mathit{string},$$

that only parses the first s-expression from an input string and returns the unread part of the string to be read later.

We can prove a similar "inverse" theorem for next_sexp via a printing function, abbrevs2string, that prints a list of s-expressions, each using some abbreviations a. That is, we show next_sexp correctly reads the first s-expression, and leaves the other expressions for later:

$$\forall s\ a\ \mathit{rest}.\ \ \mathsf{next\_sexp}\ (\mathsf{abbrevs2string}\ ((s, a) :: \mathit{rest}))\ =\ (s, \mathsf{abbrevs2string}\ \mathit{rest}).$$

4.4 Top-level specification

We give our top-level specification of what constitutes a valid Jitawa execution as an inductive relation, $\xrightarrow{\mathsf{exec}}$. Each execution terminates when the input stream ends or contains only whitespace characters.

$$\frac{\mathsf{is\_empty}\ (\mathsf{get\_input}\ io)}{(k, io) \xrightarrow{\mathsf{exec}} io}$$

Otherwise, the next s-expression is read from the input stream using next_sexp, this s-expression s is then evaluated according to $\xrightarrow{\text{ev}}$, and finally the result of evaluation, *ans*, is appended to the output stream before execution continues.

$$\frac{\begin{array}{l}\neg\text{is\_empty}\ (\text{get\_input}\ io) \wedge \\ \text{next\_sexp}\ (\text{get\_input}\ io)) = (s, rest) \wedge \\ (\text{sexp2term}\ s, [], k, \text{set\_input}\ rest\ io) \xrightarrow{\text{ev}} (ans, k', io') \wedge \\ (k', \text{append\_to\_output}\ (\text{sexp2string}\ ans)\ io') \xrightarrow{\text{exec}} io''\end{array}}{(k, io) \xrightarrow{\text{exec}} io''}$$

5 The Jitawa implementation

The verified implementation of Jitawa is 7,500 lines of x86 machine code. Most of this code was not written and verified by hand. Instead, we produced the implementation using a combination of manual verification, decompilation and proof-producing synthesis [19].

1. We started by defining a simple stack-based bytecode language into which we can easily compile Lisp programs using a simple compilation algorithm.
2. Next, we defined a heap invariant and proved that certain machine instruction "snippets" implement basic Lisp operations and maintain this invariant.
3. These snippets of verified machine code were then given to our extensible synthesis tool [19] which we used to synthesise verified x86 machine code for our compilation algorithm.
4. Next, we proved the concrete byte representation of the abstract bytecode instructions is *in itself* machine code which performs the bytecode instructions themselves. Thus jumping directly to the concrete representation of the bytecode program will correctly execute it on the x86 machine.
5. Finally, we verified code for parsing and printing of s-expressions from an input and output stream and connected these up with compilation to produce a "parse, compile, jump to compiled code, print" loop, which we have proved implements Jitawa's specification.

Steps 2 and 3 correspond very closely to how we synthesised, in previous work [18], verified machine-code for our Lisp evaluation function lisp_eval.

5.1 Compilation to bytecode

Jitawa compiles all expressions before they are executed. Our compiler targets a simple stack-based bytecode shown in Figure 1. At present, no optimizations are performed except for tail-call elimination and a simple optimization that speeds up evaluation of LambdaApp, Let and LetStar.

We model our compilation algorithm as a HOL function that takes the name, parameters, and body of the new function, and also a system state s. It returns

bytecode	::=	Pop	pop one stack element
	|	PopN *num*	pop n stack elements below top element
	|	PushVal *num*	push a constant number
	|	PushSym *string*	push a constant symbol
	|	LookupConst *num*	push the nth constant from system state
	|	Load *num*	push the nth stack element
	|	Store *num*	overwrite the nth stack element
	|	DataOp *primitive*	add, subtract, car, cons, ...
	|	Jump *num*	jump to program point n
	|	JumpIfNil *num*	conditionally jump to n
	|	DynamicJump	jump to location given by stack top
	|	Call *num*	static function call (faster)
	|	DynamicCall	dynamic function call (slower)
	|	Return	return to calling function
	|	Fail	signal a runtime error
	|	Print	print an object to stdout
	|	Compile	compile a function definition

Fig. 1. Abstract syntax of our bytecode.

a new system state, s', where the compiled code for body has been installed and other minor updates have been made.

$$\mathsf{compile}\ (name, params, body, s)\ =\ s'$$

We model the execution of bytecode using an operational semantics based on a next-state relation $\xrightarrow{\mathsf{next}}$. For simplicity and efficiency, we separate the value stack xs from the return-address stack rs; the relation also updates a program counter p and the system state s. The simplest example of $\xrightarrow{\mathsf{next}}$ is the Pop instruction, which just pops an element off the expression stack and advances the program counter to the next instruction.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Pop}])}{(top :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (xs, rs, p + \mathsf{length}(\mathsf{Pop}), s)}$$

Call *pos* is not much more complicated: we change the program counter to *pos* and push a return address onto the return stack.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Call}\ pos])}{(xs, rs, p, s) \xrightarrow{\mathsf{next}} (xs, (p + \mathsf{length}(\mathsf{Call}\ pos)) :: rs, pos, s)}$$

A DynamicCall is similar, but reads the name and expected arity n of the function to call from the stack, then searches in the current state to locate the position *pos* where the compiled code for this function begins.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{DynamicCall}])\ \wedge\ \mathsf{find\_func}\ (fn, s) = \mathsf{some}\ (n, pos)}{(\mathsf{Sym}\ fn :: \mathsf{Val}\ n :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (xs, (p + \mathsf{length}(\mathsf{DynamicCall})) :: rs, pos, s)}$$

The Print instruction is slightly more exotic: it appends the string representation of the top stack element, given by sexp2string (Section 4.3), onto the output stream, which is part of the system state s. It leaves the stack unchanged.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Print}])\ \wedge\ \mathsf{append\_to\_output}\ (\mathsf{sexp2string}\ top, s) = s'}{(top :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (top :: xs, rs, p + \mathsf{length}(\mathsf{Print}), s')}$$

The most interesting bytecode instruction is, of course, Compile. This instruction reads the name, parameter list, and body of the new function from the stack and updates the system state using the compile function.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Compile}])\ \wedge\ \mathsf{compile}\ (name, params, body, s) = s'}{(body :: params :: name :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (\mathsf{nil} :: xs, rs, p + \mathsf{length}(\mathsf{Compile}), s')}$$

At first sight, this definition might seem circular since the compile function operates over bytecode instructions. It is not circular: we first define the syntax of bytecode instructions, then the compile function which generates bytecode, and only then define the semantics of evaluating bytecode instructions, $\xrightarrow{\mathsf{next}}$.

Compile instructions are generated when we encounter an application of Define. For instance, when the compiler sees an expression like

`(define 'increment '(n) '(+ n '1))`,

it generates the following bytecode instructions (for some specific k):

PushSym `"INCREMENT"`	pushes symbol `increment` onto the stack
LookupConst k	pushes expression `(n)` onto the stack
LookupConst $(k{+}1)$	pushes expression `(+ n '1)` onto the stack
Compile	compiles the above expression

5.2 From bytecode to machine code

Most compilers use some intermediate language before producing concrete machine code. However, our compiler goes directly from source to concrete machine code by representing bytecode instructions as a string of bytes that *are* machine code. For example, the Compile instruction is represented by bytes

```
48 FF 52 88
```

which happens to be 64-bit x86 machine code for `call [rdx-120]`, i.e. an instruction which makes a procedure call to a code pointer stored at memory address `rdx-120`.

For each of these byte sequences, we prove a *machine-code Hoare triple* [17] which states that it correctly implements the intended behaviour of the bytecode instruction in question with respect to a heap invariant lisp_bytecode_inv.

$$\begin{array}{l}\mathsf{compile}\ (name, params, body, s) = s'\ \Longrightarrow \\ \{\, \mathsf{lisp\_bytecode\_inv}\ (body :: params :: name :: xs, rs, s) * \mathsf{pc}\ p \,\} \\ \ p : \texttt{48 FF 52 D8} \\ \{\, \mathsf{lisp\_bytecode\_inv}\ (\mathsf{nil} :: xs, rs, s') * \mathsf{pc}\ (p + 4)\ \vee\ \mathsf{error} \,\}\end{array}$$

At this stage you might wonder: but doesn't that Hoare triple rely on more code than just those four bytes? The answer is yes: it requires machine code which implements the compile function from above. We have produced such machine code by a method described in previous work [19]; essentially, we teach the theorem prover about basic operations w.r.t. to a heap invariant and then have the theorem prover synthesize machine code that implements the high-level algorithm for compilation. The code we synthesized in this way is part of lisp invariant lisp_bytecode_inv shown above. Thus when the above x86 instruction (i.e. `call [rdx-120]`) is executed, control just jumps to the synthesised code which when complete executes a return instruction that brings control back to the end of those four bytes.

The Hoare triples used here [17] do not require code to be at the centre of the Hoare triple as the following "code is data" theorem shows:

$$\forall p\ c\ q.\quad \{p\}\ c\ \{q\}\ =\ \{p * \mathsf{code}\ c\}\ \emptyset\ \{q * \mathsf{code}\ c\}$$

A detailed explanation of this rule, and a few others that are handy when dealing with self-modifying code, can be found in our previous paper [17].

5.3 I/O

Jitawa calls upon the external C routines `fgets` and `fputs` to carry out I/O. These external calls require assumptions in our proof. For instance, we assume that calling the routine at a certain location x—which our unverified C program initializes to the location of `fgets` before invoking the runtime—will:

1. produce a pointer z to a null-terminated string that contains the first n characters of the input stream, for some n, and
2. remove these first n characters from the input stream.

We further assume that the returned string is only empty if the input stream was empty. The machine-code Hoare triple representing this assumption is:

$$\begin{array}{l}\{\,\mathsf{rax}\ x * \mathsf{rbx}\ y * \mathsf{memory}\ m * \mathsf{io}\ (x, in, out) * \mathsf{pc}\ p\,\}\\ \quad p : \texttt{call rax}\\ \{\,\exists z\ n.\ \mathsf{rax}\ x * \mathsf{rbx}\ z * \mathsf{memory}\ m' * \mathsf{io}\ (x, \mathsf{drop}\ n\ in, out) * \mathsf{pc}\ (p+3)\ *\\ \qquad \langle \mathsf{string\_in\_mem\_at}\ (z, m', \mathsf{take}\ n\ in) \wedge (n = 0 \implies in = \texttt{""})\rangle\,\}\end{array}$$

The fact that Jitawa implements an interactive read-eval-print loop is not apparent from our top-level correctness statement: it is just a consequence of reading lazily—our next_sexp style parser reads only the first s-expression, and `fgets` reads through at most the first newline—and printing eagerly.

5.4 Top-level correctness theorem

The top-level correctness theorem is stated as the following machine-code Hoare triple. If the Jitawa implementation is started in a state where enough memory

is allocated (init_state) and the input stream holds s-expressions for which an execution of Jitawa terminates, then either a final state described by $\xrightarrow{\text{exec}}$ is reached or an error message is produced.

$$\{\, \mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io \rangle \,\}$$
$$p : \mathsf{code\_for\_entire\_jitawa\_implementation}$$
$$\{\, \mathsf{error\_message} \vee \exists io'.\ \langle ([], io) \xrightarrow{\text{exec}} io' \rangle * \mathsf{final\_state}\ io' \,\}$$

This specification allows us to resort to an error message even if the evaluated s-expressions would have a meaning in terms of the $\xrightarrow{\text{exec}}$ relation. This lets us avoid specifying at what point implementation-level resource limits are hit. The implementation resorts to an error message when Jitawa runs into an arithmetic overflow, attempts to parse a too long symbol (more than 254 characters long), or runs out of room on the heap, stack, symbol table or code heap.

6 The combination of Milawa and Jitawa

Jitawa runs fast enough and manages its memory well enough to successfully complete the proof of self-verification for Milawa. This is a demanding proof that many Common Lisp systems cannot successfully complete. The full input file is 4 gigabytes and contains 520 million unique conses. On our computer, Clozure Common Lisp—an excellent, state-of-the-art Common Lisp implementation—takes 16 hours to finish the proof; this is with all optimization enabled, garbage collection tuning, and inlining hints that provide significant benefits.

Jitawa is currently about 8x slower for the full proof. While this is considerably slower, it may be adequate for some proof efforts. We are also investigating how performance may be improved. Jitawa is only 20% slower than CCL on the first 4,500 events (about 1.5 hours of computation), so it seems that competitive performance may be possible.

Is there a formal connection between Jitawa and Milawa? We would eventually like to mechanically prove that, when run with Jitawa, Milawa's trusted core is faithful to the Milawa logic. We have not yet done this, but we have at least proved a weaker connection, viz. evaluation in Jitawa respects the 52 axioms [5, Ch. 2] of the Milawa logic that constrain the behavior of Lisp primitives.

For instance, the *Closed Universe Axiom* says every object must satisfy either `natp`, `symbolp`, or `consp`. In Milawa, this is written as:

```
(por* (pequal* (natp x) 't)
      (por* (pequal* (symbolp x) 't)
            (pequal* (consp x) 't)))
```

The corresponding HOL theorem is stated as:

```
valid_sexp ["x"] ("  (por* (pequal* (natp x) 't)            " ++
                  "        (por* (pequal* (symbolp x) 't)   " ++
                  "              (pequal* (consp x) 't)))   ")
```

We are able to copy the axiom statements into HOL nearly verbatim by having valid_sexp use our parser to read the string into its datatype representation.

7 Discussion and related work

Theorem provers are generally very trustworthy. The LCF [7] approach has long been used to minimize the amount of code that must be trusted. Harrison [11] has even formally proved—using an altered version of HOL Light—theorems suggesting HOL Light's LCF-style kernel is faithful to its logic. By addressing the correctness of the runtime system used to execute the prover, we further increase our confidence in these systems.

Runtime correctness may be particularly important for theorem provers that employ reflective techniques. In a separate paper [9], Harrison remarks:

> "[...] the final jump from an abstract function inside the logic to a concrete implementation in a serious programming language which *appears to correspond to it* is a glaring leap of faith."

While we have not proven a formal connection between Milawa and Jitawa, our work suggests it may be possible to verify a theorem prover's soundness down to the concrete machine code which implements it, thus reducing this "glaring leap of faith." We have kept Jitawa's top-level specification (Section 4) as simple as possible to facilitate such a proof.

Most of this paper has dealt with the question: how do we create a verified Lisp system that is usable and scales well? The most closely related work on this topic is the VLISP project [8], which produced a "comprehensively" (not formally) verified Scheme implementation. The subset of Scheme which they address is impressive: it includes strings, destructive updates and I/O. However, their formalisation and proofs did not reach as far as machine or assembly level, as we have done here and in previous work [18].

Recently, Leroy's Coq-verified C compiler [14], which targets PowerPC, ARM and 32-bit x86 assembly, has been extended with new front-ends that makes it compile MiniML [4] and a garbage-collected source language [16]. The latter extension has been connected to intermediate output from the Glasgow Haskell Compiler. Our runtime uses a verified copying garbage collector similar to the sample collector in McCreight et al. [16], but less than 10% of our proof scripts are directly concerned with verification of our garbage collector and interfacing with it; our approach to this is unchanged from our previous paper [18].

Unrelated to Leroy's C compiler, Chlipala [3] has done some interesting verification work, in Coq, on compilation of a functional language: he verified a compiler from a functional language with references and exceptions to a toy assembly language. Chlipala emphasises use of adaptive programs in Coq's tactic language to make proofs robust.

Source code. The code for Jitawa and its 30,000-line verification proof are available at `http://www.cl.cam.ac.uk/~mom22/jitawa/`. Similarly, Milawa is available at `http://www.cs.utexas.edu/~jared/milawa/Web/`.

Acknowledgements. We thank Mike Gordon, Warren Hunt, Scott Owens, Anna Slobadová and Sol Swords for commenting on drafts of this paper. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1997.
2. Robert S. Boyer and Warren A. Hunt, Jr. Function memoization and unique object representation for ACL2 functions. In *ACL2 '06*. ACM, 2006.
3. Adam J. Chlipala. A verified compiler for an impure functional language. In *POPL*. ACM, 2010.
4. Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *LPAR*. Springer, 2007.
5. Jared C. Davis. *A Self-Verifying Theorem Prover*. PhD thesis, University of Texas at Austin, December 2009.
6. James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.
7. Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS. Springer-Verlag, 1979.
8. Joshua Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.
9. John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Cambridge, UK, 1995.
10. John Harrison. Formalizing basic first order model theory. In *TPHOLs*, LNCS. Springer, 1998.
11. John Harrison. Towards self-verification of HOL Light. In *IJCAR*, LNAI. Springer, 2006.
12. John Harrison. HOL Light: An overview. In *TPHOLs*, LNCS. Springer, 2009.
13. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.
14. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*. ACM, 2006.
15. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3(4):184–195, April 1960.
16. Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*. ACM, 2010.
17. Magnus O. Myreen. Verified just-in-time compiler on x86. In *POPL*. ACM, 2010.
18. Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *TPHOLs*, LNCS. Springer, 2009.
19. Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In *Compiler Construction (CC)*, LNCS. Springer, 2009.
20. Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *TPHOLs*, LNCS. Springer, 2005.
21. Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, LNCS. Springer, 2008.