The reflective Milawa theorem prover is sound

(down to the machine code that runs it)

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

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

Abstract. Milawa is a theorem prover styled after ACL2 but with a small kernel and a powerful reflection mechanism. We have used the HOL4 theorem prover to formalize the logic of Milawa, prove the logic sound, and prove that the source code for the Milawa kernel (2,000 lines of Lisp) is faithful to the logic. Going further, we have combined these results with our previous verification of an x86 machine-code implementation of a Lisp runtime. Our top-level HOL4 theorem states that when Milawa is run on top of our verified Lisp, it will only print theorem statements that are semantically true. We believe that this top-level theorem is the most comprehensive formal evidence of a theorem prover's soundness to date.

1 Introduction

Theorem provers like HOL4, Coq, and ACL2 are each meant to reason in some particular logic, are each written in a programming language like ML, OCaml, or Lisp, and are each executed by a runtime like Poly/ML, the OCaml system, or Clozure Common Lisp. If we want to make sure that a theorem prover can only prove true statements, we should ideally show that:

A. the logic is sound,

B. the theorem prover's source code is faithful to its logic, and

C. the runtime executes the source code correctly.

In this paper, we explain how we have used the HOL4 theorem prover to establish these three properties about the Milawa theorem prover.

Milawa [2] is a theorem prover inspired by NQTHM and ACL2. Unlike these programs it has a small kernel, somewhat like an LCF-style system. This kernel notably performs reflection and includes a mechanism that modifies the kernel at runtime. High-level tactics like (conditional) rewriting are added into the kernel through a sequence of reflective extensions.

Our proofs of A through C for the Milawa prover are the key lemmas in a single, top-level HOL4 theorem: when the kernel of the Milawa theorem is run on our verified Lisp runtime, Jitawa [13], it will only ever prove statements that are true with respect to the semantics of Milawa's logic. This theorem means, for instance, that no matter how reflection or any other operation is used, the statement 'true equals false' can never be proved. This top-level theorem relates the semantics of the logic (not just syntactic provability) all the way down to the concrete x86 machine code.

We believe this work provides the most comprehensive formal evidence of a theorem prover's soundness to date, as the combination of these three properties have, to our knowledge, never before been formally proved for any interactive theorem prover.

2 Milawa in a nutshell

Before delving into the details of our formalizations and soundness results, we start with a high-level description of the Milawa theorem prover.

ACL2-like. Milawa follows the Boyer-Moore tradition of theorem provers. Like NQTHM and ACL2, its logic is essentially a clean subset of first-order Lisp. Also like these systems, its top-level loop processes user-provided *events*. Events steer the prover process. A user can submit events which, for example, cause the prover to define a new function or prove a specific theorem. However, Milawa is simpler than ACL2 in many ways. Milawa is particularly minimalist in its user-interface and debugging output: it really just processes a list of events, aborting if any event is unacceptable.

Small kernel. The most important difference between Milawa and ACL2 is that Milawa has a small logical kernel, somewhat like an LCF-style prover. In contrast, other Boyer-Moore systems have no cordoned off area for soundness-critical code. This design means that the authors of ACL2 must program very carefully to avoid accidentally introducing soundness bugs. But it also means that ACL2 can make greater leaps in reasoning and perform well on large-scale applications; the ACL2 design avoids the LCF-bottleneck where all proofs must *at runtime* boil down to the primitive inferences of the logic.

Reflection. Milawa was designed to show that it is possible to combine the benefits of a small trusted kernel and, at the same time, avoid the LCF-bottleneck. Milawa has approximately 2000 lines of soundness-critical Lisp code. This Lisp code initializes the system and sets up the top-level event handling loop. An important part of this code is the *initial proof checker*. This initial proof checker only accepts proofs that use the primitive inferences of the logic, very much like an LCF-style kernel. In order to allow larger steps in proofs, Milawa supports a special event that replaces the prover's current proof checker with a new, user-supplied proof checker. For this *switch event* to be accepted, we must first prove that the new user-supplied proof checker (which is just a function in the logic of Milawa) can only prove statements that the initial proof checker could have proved. The initial proof checker lives within the Milawa logic. Every function defined in the logic is also defined outside in the underlying Lisp runtime.

Bootstrapping. By (repeatedly) replacing the initial proof checker with new, improved checkers that can make larger leaps in their proofs, we can build a prover that performs ACL2-style proofs where, e.g., conditional rewriting is treated as a single inference step. We call the soundness critical code—the initial 2000-line Lisp program—Milawa's kernel. The Milawa theorem prover is what this kernel morphs into after running through a long list of events (the *bootstrapping* sequence) that ultimately installs a powerful, ACL2-like proof checker. This final proof checker allows for high-level, Boyer-Moore style steps such as rewriting, case splitting, generalization, cross-fertilization, and so forth.

3 Method

To prove the soundness of Milawa in HOL4, we proceeded as follows.

A. We started by formalizing Milawa's logic, following closely the detailed prose description given in Chapter 2 of Davis [2]. We then proved the logic is sound. This part was largely a routine formalization and soundness proof (Section 4), but we did hit some surprises involving the termination obligations Milawa generates (Section 7).

B. Next we turned our attention to the implementation of Milawa's kernel. Our task here was to verify these 2,000 lines of Lisp code with respect to the behavior of Jitawa [13], our verified Lisp runtime. We proved a connection using the following steps (Section 5).

   1. Jitawa's correctness theorem is stated in terms of a read-eval-print loop which reads ASCII input. Using rewriting, we evaluated its parser on the ASCII definition of Milawa's kernel.
   2. Once the ASCII input had been turned into appropriate abstract syntax, we ran a proof-producing tool [12] to translate deeply embedded Lisp programs into their 'obvious' shallowly embedded counterparts.
   3. Given the convenient shallow embeddings, we proved that Milawa's main loop maintains an invariant that implies that all proved theorems are true w.r.t. our semantics of Milawa's logic.

C. We had already verified our Lisp runtime, Jitawa, as described in a previous paper [13]. What remained was to connect the results from *A* and *B* to Jitawa's top-level correctness theorem (Section 6).

The result of combining *A*, *B* and *C* is a top-level theorem (Section 6) that relates logical soundness all the way down to machine-code execution. We found mistakes in Milawa's implementation, but no soundness bugs (Section 7).

4 Milawa's logic

We start with a formalization and soundness proof of Milawa's logic. Milawa targets a first-order logic of recursive functions with induction up to ϵ_0, similar to the logics of NQTHM and ACL2. The objects of the logic are the natural numbers, symbols, and conses (ordered pairs) of other objects; we call these objects S-expressions. The logic has primitive functions for working with S-expressions like equality checking, addition, cons, car, cdr, etc., whose behavior is given with axioms. Starting from these primitives, we can define recursive functions that look like Lisp programs. An introduction to the logic can be found in Chapter 2 of Davis [2].

The Milawa logic is considerably weaker than popular higher-order logics. Thanks to this, its soundness can be established using higher-order logic as the meta-logic. In this section, we explain how we have used the HOL4 system to formalize the syntax (Section 4.1), semantics (4.3) and rules of inference (4.4) of the Milawa logic, and to mechanically prove the soundness of its inference rules (4.5) and definition principle (4.6). In later sections we connect these soundness proofs to the theorem prover's implementation.

4.1 Syntax of terms and formulas

We formalize the syntax of the Milawa logic as the following datatype:

sexp	::=	Val *num* | Sym *string* | Dot *sexp* *sexp*	S-expression
prim	::=	If | Equal | Not | Symbolp | Symbol_less	
	|	Natp | Add | Sub | Less | Consp | Cons	
	|	Car | Cdr | Rank | Ord_less | Ordp	
func	::=	PrimitiveFun *prim*	primitive functions
	|	Fun *string*	user-defined
term	::=	Const *sexp*	constant S-expression
	|	Var *string*	variable
	|	App *func* (*term* list)	function application
	|	LamApp (*string* list) *term* (*term* list)	λ formals body actuals
formula	::=	$\neg$*formula*	negation
	|	*formula* $\vee$ *formula*	disjunction
	|	*term* = *term*	term equality

These type definitions are not quite enough to capture correct Milawa syntax. We write separate well-formedness predicates called term_ok and formula_ok to formalize the additional requirements. In particular,

– every function application must have correct arity and refer to a known function with respect to the context (see below), and
– every lambda application must have the same number of formal and actual parameters, must have distinct formal parameters, and its body must not refer to variables other than its formal parameters; these requirements make substitution straightforward.

The $\mathsf{term\_ok}$ and $\mathsf{formula\_ok}$ well-formedness predicates depend on a *logical context*, π, which will be explained below.

4.2 Context

The definitions of the syntax, semantics and inference rules all depend on information regarding user-defined functions. To keep the formalization simple, we chose to combine all of this information into a single mapping, which we call the *logical context.* We model the logical context as a finite partial map π from function names, of type *string*, to elements of type:

$$\mathit{string}\ \mathsf{list} \times \mathit{func\_body} \times (\mathit{sexp}\ \mathsf{list} \to \mathit{sexp})$$

The first component, *string* list, names the formal parameters for the function. The second component, *func_body*, gives the syntactic definition for the function. This *func_body* is usually either (1) the right-hand side of a definition, for an ordinary function defined by an equation, or (2) a variable name and property, for a witness (Skolem) function. For reasons that will be explained in Section 4.6, we also allow the omission of the function body, i.e., a None alternative.

func_body ::=	Body *term*	concrete term (e.g. recursive function)
|	Witness *term string*	property, element name
|	None	no function body given

Finally, the *sexp* list $\to$ *sexp* component is an *interpretation function*, which is used in the definition of the semantics. These interpretation functions specify what meaning the semantics is to assign to applications of user-defined functions. In the next section, we will see a well-formedness criteria that relates the interpretation functions with the syntax in *func_body*.

4.3 Semantics

Next, we define a semantics of Milawa's formulas. We present these semantics in a top-down order. Our topmost definition is validity: a Milawa formula p is *valid*, written $\models_\pi p$, if and only if (1) p is syntactically correct w.r.t. the logical context π and (2) p evaluates to true in π for all variable instantiations i.

$$(\models_\pi p) \quad = \quad \mathsf{formula\_ok}_\pi\ p\ \wedge\ \forall i.\ \mathsf{eval\_formula}\ i\ \pi\ p$$

We define the evaluation of a formula with respect to a particular variable instantiation i. Our formula evaluator, $\mathsf{eval\_formula}\ i\ \pi$, is built on top of a term evaluator, $\mathsf{eval\_term}\ i\ \pi$, as follows. The syntax overloading can be confusing in the following definition. On the left-hand side $\neg$, $\vee$ and $\mathtt{=}$ are the constructors for the *formula* type, while on the right-hand side $\neg$ and $\vee$ are the usual Boolean connectives and $=$ is HOL's equality predicate.

$$\begin{array}{lcl}
\mathsf{eval\_formula}\ i\ \pi\ (\neg p) & = & \neg(\mathsf{eval\_formula}\ i\ \pi\ p) \\
\mathsf{eval\_formula}\ i\ \pi\ (p \vee q) & = & \mathsf{eval\_formula}\ i\ \pi\ p \vee \mathsf{eval\_formula}\ i\ \pi\ q \\
\mathsf{eval\_formula}\ i\ \pi\ (x \mathtt{=} y) & = & (\mathsf{eval\_term}\ i\ \pi\ x = \mathsf{eval\_term}\ i\ \pi\ y)
\end{array}$$

We define term evaluation with respect to a variable instantiation i. Here $[[v_0, \ldots, v_n] \mapsto [x_0, \ldots, x_n]]$ is a function which maps v_i to x_i, for $0 \le i \le n$, and all other variable names to NIL. Below map is a function such that map $f\ [t_0, t_1, \ldots, t_n] = [f\ t_0, f\ t_1, \ldots, f\ t_n]$.

$$\begin{array}{lcl}
\mathsf{eval\_term}\ i\ \pi\ (\mathsf{Const}\ c) & = & c \\
\mathsf{eval\_term}\ i\ \pi\ (\mathsf{Var}\ v) & = & i(v) \\
\mathsf{eval\_term}\ i\ \pi\ (\mathsf{App}\ f\ xs) & = & \mathsf{eval\_app}\ (f, \mathsf{map}\ (\mathsf{eval\_term}\ i\ \pi)\ xs, \pi) \\
\mathsf{eval\_term}\ i\ \pi\ (\mathsf{LambdaApp}\ vs\ x\ xs) & = & \mathsf{let}\ ys = \mathsf{map}\ (\mathsf{eval\_term}\ i\ \pi)\ xs\ \mathsf{in} \\
 & & \quad \mathsf{eval\_term}\ [vs \mapsto ys]\ \pi\ x
\end{array}$$

Application of a function to a list of concrete arguments, a list of type *sexp* list, is evaluated according the following eval_app function. This function evaluates primitive functions according to eval_primitive and user-defined functions according to the interpretation function *interp* stored in the logical context. The interpretation functions will be explained further below.

$$\begin{array}{lcl}
\mathsf{eval\_app}\ (\mathsf{PrimitiveFun}\ p, args, \pi) & = & \mathsf{eval\_primitive}\ p\ args \\
\mathsf{eval\_app}\ (\mathsf{Fun}\ name, args, \pi) & = & \mathsf{let}\ (\_, \_, interp) = \pi(name)\ \mathsf{in} \\
 & & \quad interp(args)
\end{array}$$

We omit the definition of eval_primitive, which is lengthy and straightforward, but note that it is a total function. A few example evaluations:

$$\begin{array}{ll}
\mathsf{eval\_primitive\ Add}\ [\mathsf{Val}\ 2, \mathsf{Val}\ 3] & = \mathsf{Val}\ 5 \\
\mathsf{eval\_primitive\ Add}\ [\mathsf{Val}\ 2, \mathsf{Sym}\ \texttt{"a"}] & = \mathsf{Val}\ 2 \\
\mathsf{eval\_primitive\ Cons}\ [\mathsf{Val}\ 2, \mathsf{Sym}\ \texttt{"a"}] & = \mathsf{Dot}\ (\mathsf{Val}\ 2)\ (\mathsf{Sym}\ \texttt{"a"})
\end{array}$$

The definitions above constitute the semantics of Milawa. Clearly, this semantics is intimately dependent on the interpretation functions stored inside the context π. In order to make sure that these interpretation functions are 'the right ones', i.e., correspond to the syntactic definitions of the user-defined functions, we require that the context is well-formed, i.e., satisfies a predicate we will call context_ok.

For a context to be well-formed, any user-defined functions with an entry of the following form in the logical context π,

$$\pi(name) \ = \ (formals, \mathsf{Body}\ body, interp)$$

must have the *interp* function return the same value as an evaluation of *body* with appropriate instantiations of the formal parameters, i.e., the following *defining equation* must be true:

$$\forall i.\ \ interp(\mathsf{map}\ i\ formals) \ = \ \mathsf{eval\_term}\ i\ \pi\ body$$

Note that this is a non-trivial equation since eval_term, which appears on the right-hand side of the equation, can refer to *interp* via eval_app. Indeed, proving soundness of the definition principle requires proving that the termination obligations generated by Milawa imply that our interpetation is total (Section 4.6).

A similar condition applies to witness functions. If,

$$\pi(\mathit{name}) \;=\; (\mathit{formals}, \mathsf{Witness}\ \mathit{prop}\ \mathit{var}, \mathit{interp})$$

is true then the following implication must hold. This implication states that, if there exists some value v such that property *prop* is true when variable names $\mathit{var} :: \mathit{formals}$ are substituted for values $v :: \mathit{args}$ in *prop*, then $\mathit{interp}(\mathit{args})$ returns some such value v. Here the test for 'is true' is Lisp's truth test, i.e., 'not equal to `NIL`'. These witness functions are explained in Davis [2].

$$\begin{array}{l}
\forall \mathit{args}.\\
\quad (\exists v.\ \mathsf{eval\_term}\ [\mathit{var} :: \mathit{formals} \mapsto v :: \mathit{args}]\ \pi\ \mathit{prop} \neq \mathtt{NIL}) \Longrightarrow\\
\quad \mathsf{eval\_term}\ [\mathit{var} :: \mathit{formals} \mapsto \mathit{interp}(\mathit{args}) :: \mathit{args}]\ \pi\ \mathit{prop} \neq \mathtt{NIL}
\end{array}$$

The well-formedness criteria for contexts puts no restrictions on the *interp* function if the function body is None.

The full definition of the well-formedness criteria for contexts, context_ok, is given below. Here free_vars is a function that computes the list of free variables of a term, and list_to_set converts a list to a set.

$$\begin{array}{l}
\mathsf{context\_ok}\ \pi \;=\\
\quad (\forall \mathit{name}\ \mathit{formals}\ \mathit{body}\ \mathit{interp}.\\
\quad\quad (\pi(\mathit{name}) = (\mathit{formals}, \mathsf{Body}\ \mathit{body}, \mathit{interp})) \Longrightarrow\\
\quad\quad\quad \mathsf{term\_ok}_\pi\ \mathit{body} \wedge \mathsf{all\_distinct}\ \mathit{formals}\ \wedge\\
\quad\quad\quad \mathsf{list\_to\_set}\ (\mathsf{free\_vars}\ \mathit{body}) \subseteq \mathsf{list\_to\_set}\ \mathit{formals}\ \wedge\\
\quad\quad\quad \forall i.\ \mathit{interp}(\mathsf{map}\ i\ \mathit{formals}) = \mathsf{eval\_term}\ i\ \pi\ \mathit{body}) \wedge\\
\quad (\forall \mathit{name}\ \mathit{formals}\ \mathit{prop}\ \mathit{var}\ \mathit{interp}.\\
\quad\quad (\pi(\mathit{name}) = (\mathit{formals}, \mathsf{Witness}\ \mathit{prop}\ \mathit{var}, \mathit{interp})) \Longrightarrow\\
\quad\quad\quad \mathsf{term\_ok}_\pi\ \mathit{prop} \wedge \mathsf{all\_distinct}\ (\mathit{var} :: \mathit{formals})\ \wedge\\
\quad\quad\quad \mathsf{list\_to\_set}\ (\mathsf{free\_vars}\ \mathit{prop}) \subseteq \mathsf{list\_to\_set}\ (\mathit{var} :: \mathit{formals})\ \wedge\\
\quad\quad\quad \forall \mathit{args}.\\
\quad\quad\quad\quad (\exists v.\ \mathsf{eval\_term}\ [\mathit{var} :: \mathit{formals} \mapsto v :: \mathit{args}]\ \pi\ \mathit{prop} \neq \mathtt{NIL}) \Longrightarrow\\
\quad\quad\quad\quad \mathsf{eval\_term}\ [\mathit{var} :: \mathit{formals} \mapsto \mathit{interp}(\mathit{args}) :: \mathit{args}]\ \pi\ \mathit{prop} \neq \mathtt{NIL})
\end{array}$$

4.4 Inference Rules

Due to space constraints, this section will only sketch a few of Milawa's 13 inference rules. Two of the simplest are shown below. Here milawa_axioms is a set consisting of the 56 axioms from Davis [2]. Most of these are basic facts about the primitive functions, e.g. term equality is reflective, symmetric and transitive; the Less primitive is anti-reflective and transitive, etc.

$$\frac{\vdash_\pi a \vee (b \vee c)}{\vdash_\pi (a \vee b) \vee c}\ \text{(associativity)} \qquad\qquad \frac{a \in \mathsf{milawa\_axioms}}{\vdash_\pi a}\ \text{(basic axiom)}$$

The most complicated inference rule allows induction according to a user-defined measure over the ordinals up to ε_0. We omit the presentation of that lengthy inference rule, which Chapter 6 of Kaufmann et al. [6] explains in detail.

Apart from the normal inference rules, we also include rules that allow function definitions to be looked up from the logical context, e.g.

$$\frac{\pi(\mathit{name}) = (\mathit{formals}, \mathsf{Body}\ \mathit{body}, \mathit{interp})}{\vdash_\pi \mathsf{App}\ (\mathsf{Fun}\ \mathit{name})\ (\mathsf{map}\ \mathsf{Var}\ \mathit{formals})\ \texttt{=}\ \mathit{body}}$$

4.5 Soundness and Consistency

We state the soundness theorem for Milawa's inference rules as follows:

$$\forall \pi\ p.\ \ \mathsf{context\_ok}\ \pi \land (\vdash_\pi p) \Longrightarrow (\models_\pi p)$$

We have proved this statement by induction over the inference rules $\vdash_\pi$. Proving soundness of the induction rule was the most interesting case: this proof required induction over the ordinals up to ε_0, for which we need to know that less-than over these ordinals is well-founded. Fortunately, Kaufmann and Slind [7] had already formalized this result in HOL4. The soundness of the induction rule follows almost directly from their result.

The soundness theorem from above lets us immediately prove many reassuring corollaries. For instance, since $\models_\pi$ `T = NIL` is false and $\vdash_\pi$ `T = T` is true we know that Milawa's inference rules are consistent.

4.6 Soundness Preserved by Function Definitions

As part of our verification of Milawa's kernel (Section 5.4), we have proved that the kernel maintains an invariant which states that (1) the current logical context π is well-formed, $\mathsf{context\_ok}\ \pi$, and (2) that all theorems the Milawa theorem prover has accepted are provable using the inference rules based on that current context π, i.e., for any formula p accepted by the kernel, we have $\vdash_\pi p$. However, when new definitions are made the logical context is extended. In order to maintain our invariant, we must hence show that properties (1) and (2) carry across context extensions.

Proving that property (1) carries across is straightforward since the syntactic inference rules only make tests for inclusion in the context.

Proving that well-formedness of the context, i.e., property (2), carries across context extensions is less straightforward. The main complication is that we need to find an interpretation for the new function that agrees with the syntax of the new definition. Using HOL's choice operator, we define a function $\mathsf{new\_interp}$ (definition omitted) which constructs such an interpretation if such an interpretation exists. This reduces the goal to proving that an interpretation exists. For witness functions, this proof is almost immediate. For conventional functions, this proof required showing that a $\vdash_\pi$-proof of the generated termination obligations is sufficient to imply that a suitable interpretation exists. Below, $\mathsf{definition\_ok}$ (definition omitted) requires that certain syntactic conditions are true and that the termination obligations can be proved.

$$\begin{array}{l}\forall \pi\ \mathit{name}\ \mathit{formals}\ \mathit{body}.\\ \quad \mathsf{context\_ok}\ \pi \land \mathsf{definition\_ok}\ (\mathit{name}, \mathit{formals}, \mathit{body}, \pi) \Longrightarrow\\ \quad \mathsf{context\_ok}\ (\pi[\mathit{name} \mapsto (\mathit{formals}, \mathit{body}, \mathsf{new\_interp}\ \pi\ \mathit{name}\ \mathit{formals}\ \mathit{body})])\end{array}$$

5 Correctness of Milawa's implementation

With logical soundness out of the way, our next goal was to show that the source code of the Milawa kernel respects the logic's inference rules.

First, some background: in previous work [13], we introduced the Jitawa Lisp runtime. Jitawa is able to *host* the Milawa theorem prover. By this, we mean that it is able to execute Milawa's kernel all the way through its *bootstrapping process* [2], a long sequence of definitions, proofs and reflective extensions which ultimately extend the kernel with many high-level proof procedures like those of NQTHM and ACL2. As part of the Jitawa work, we developed an operational semantics for the Lisp dialect that Jitawa executes, and proved that the x86 machine code for Jitawa implements this semantics.

Milawa's kernel is about 2,000 lines of Lisp code. In this section, we explain how we have proved that this Lisp code is faithful to Milawa's inference rules w.r.t. the operational semantics that Jitawa has been proved to implement.

5.1 From ASCII characters to a shallow embedding in HOL4

The top-level Jitawa semantics describes how S-expressions are to be parsed from an input stream of ASCII characters and then evaluated. One of the simplest functions in Milawa's kernel is shown below. This function will be used as a running example of how we lift Lisp functions into HOL to make interactive verification manageable.

```
(defun lookup-safe (a x)
  (if (consp x)
      (if (equal a (car (car x)))
          (if (consp (car x))
              (car x)
            (cons (car (car x)) (cdr (car x))))
        (lookup-safe a (cdr x)))
    nil))
```

When Jitawa reads the ASCII definition of `lookup-safe`, it parses the lines above and, as far as its operational semantics is concerned, turns them into a datatype of the form:

$$\mathsf{App\ Define\ [Const\ (Sym\ \texttt{"LOOKUP-SAFE"}), Const\ (...), Const\ (...)]}$$

We wrote a custom conversion (based mostly on rewriting) in HOL4 which parses the source code for Milawa's 2000-line kernel into abstract datatypes such as the expression above. The evaluation of the parser happens inside the HOL4 logic, so the result is a theorem of the form $\mathsf{string\_to\_prog\ milawa\_kernel\_lisp} = \ldots$

When Jitawa evaluates the Define expression from above, a definition for `lookup-safe` is added to its list of functions. The new entry is:

```
function name:  "LOOKUP-SAFE"
parameter list: "A", "X"
function body:  If (App (PrimitiveFun Consp) [Var "X"])
                  (If (App (PrimitiveFun Equal) [...])
                     (If (App (PrimitiveFun Consp) [...] (...) (...))
                     (App (Fun "LOOKUP-SAFE") [...]))
                  (Const (Sym "NIL"))
```

Instead of performing tedious proofs directly over deep embeddings such as that above, we developed a tool that automatically translates these deep embeddings into shallow embeddings and, in the process, proves that the shallow embeddings accurately describe evaluations of the deep embeddings. The details of this tool are the subject of a separate paper [12], but the net effect of using it on `lookup-safe` is easy to see: we get a simple HOL function,

```
lookup_safe a x  =  if consp x then
                      if a = car (car x) then
                        if consp (car x) then
                          car x
                        else cons (car (car x)) (cdr (car x))
                      else lookup_safe a (cdr x)
                    else Sym "NIL"
```

and a theorem relating the deep embedding to this shallow embedding, stated in terms of the application relation $\xrightarrow{\mathsf{ap}}$ of Jitawa's semantics:

$$\ldots \implies (\mathsf{Fun}\ \texttt{"LOOKUP-SAFE"}, [a, x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{lookup\_safe}\ a\ x, \mathit{state})$$

Here *state* is Jitawa's mutable state which has, e.g., the I/O streams and the list of function definitions. The state is not changed by lookup_safe because `lookup-safe` is a pure function. Extracted impure functions take the state as input and produce a new state as output, e.g. Milawa's admit_defun function returns a (value, new-state) pair:

$$\ldots \implies (\mathsf{Fun}\ \texttt{"ADMIT-DEFUN"}, [\mathit{cmd}, s], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{admit\_defun}\ \mathit{cmd}\ s\ \mathit{state})$$

5.2 Milawa's proof checkers and reflection

The largest and most important pure function in Milawa is its initial proof checker, proofp. This function is given an *appeal* (an alleged proof) to check. It walks through the appeal, checking that each proof step is a valid use of some inference rule. When Milawa starts, it uses proofp to check alleged proofs of theorems and termination obligations. But the kernel can later be told to start using some user-defined function, say new-proofp, to check proofs. Typically new-proofp can accept "higher level" proofs that use new inference rules beyond the "base level" rules available in proofp. The kernel will only switch to new-proofp after establishing its *fidelity claim*: whenever new-proofp accepts a high-level proof of ϕ, there must exist a base-level proof of ϕ that proofp would accept.

We prove that proofp is faithful to the inference rules of the Milawa logic. That is, whenever proofp is given well-formed inputs and it returns something other than `NIL`, the conclusion of the alleged proof is $\vdash_\pi$-provable. Here *axioms* and *thms* are lists of formulas, and *atbl* is an arity table.

$$
\begin{array}{l}
\forall \mathit{appeal}\ \mathit{axioms}\ \mathit{thms}\ \mathit{atbl}. \\
\quad \mathsf{appeal\_syntax\_ok}\ \mathit{appeal} \wedge \mathsf{atbl\_ok}\ \pi\ \mathit{atbl}\ \wedge \\
\quad \mathsf{thms\_inv}\ \pi\ \mathit{thms} \wedge \mathsf{thms\_inv}\ \pi\ \mathit{axioms}\ \wedge \\
\quad \mathsf{proofp}\ \mathit{appeal}\ \mathit{axioms}\ \mathit{thms}\ \mathit{atbl} \neq \mathsf{Sym}\ \texttt{"NIL"} \Longrightarrow\ \vdash_\pi \mathsf{conclusion\_of}\ \mathit{appeal}
\end{array}
$$

To accommodate the reflective installation of new proof checkers, the invariant we describe in the next section requires that the property above must always hold for whatever function is the current proof checker. It turns out that Milawa's checks of the fidelity claim are sufficient to show that a new-proofp may only be installed when it satisfies this property.

5.3 Milawa's invariant

As it executes, Milawa's kernel carries around state with several lists and mappings that must be kept consistent. Its program state consists of:

- a list of axioms and definitions,
- a list of proved theorems,
- an arity table for syntax checks (e.g., are all mentioned functions defined? are they called with the right number of arguments?),
- the name of the current proof checker (proofp, new-proofp, . . .), and
- a function table that lists all the definitions that have been given to the Lisp runtime, and the names of functions that must be avoided since they have a special meaning in the runtime (error, print, define, funcall, . . .).

There is also state specific to the Lisp runtime's semantics:

- its view of how functions have been defined,
- its input and output streams, and
- a special *ok* flag that records whether an error has been raised.

Finally, for our soundness proof, there is also logical (ghost) state:

- a logical context π must also be maintained.

A key part of our proof was to formalize the invariant that relates these state elements. For the most part, the dependencies and relationships between the state components were obvious, e.g. each entry in the function table must have a corresponding entity inside the runtime's function table, and since this is a reflective theorem prover each function in the logic must have an entry in the runtime's function table.

A few details were less straightforward. Each layer has its own abstraction level, e.g. the kernel and runtime allow macros but these are expanded away in the logic, and the function table uses S-expression syntax but the runtime's operational semantics only sees an abstraction of this syntax. There are also some language mismatches: the logic has primitives (e.g. `ordp` and `ord-<`) which are not primitive in the runtime, and the runtime has several primitives that are not part of the logic (e.g., `funcall`, `print`, `error`). To further complicate things, some of these components can lag behind: the function table starts off mentioning functions that have not yet been defined in the logic. Such functions can only be defined using exactly the definition given in the function table, otherwise the defining event, `admit-defun` or `admit-witness`, causes a runtime error. We will explain this invariant in more detail in forthcoming journal article and/or extensive technical report.

We proved that each event handling function, e.g. `admit-thm`, `admit-defun`, `admit-switch` etc., maintains the invariant. As a result, the kernel's top-level event-handling loop maintains the invariant.

5.4 Theorem: Milawa is faithful to its logic

Milawa's kernel reads input, processes it, and then prints output that says whether it has accepted the proofs and definitions it has been given. In order to make it clearer what Milawa claims to have proved, we extended Milawa with a new event, (`admit-print` ϕ), which causes ϕ to be printed if it has already been proved as a theorem, or else fails. For instance, this new event can print:

```
(PRINT (THEOREM (PEQUAL* (+ A B) (+ B A))))
```

We formulate the soundness of Milawa as a guarantee about the possible output: whatever the input, Milawa will only ever print `THEOREM` lines for formulas that are true w.r.t. the semantics $\models_\pi$ of the logic. More precisely, we first define what an acceptable line of output is w.r.t. a given logical context π:

$$\begin{array}{rcl}\mathsf{line\_ok}\ (\pi, l) & = & (l = \texttt{"NIL"}) \lor \\ & & (\exists n.\ (l = \texttt{"(PRINT (}n\ \texttt{...))"}) \land \mathsf{is\_number}\ n) \lor \\ & & (\exists \phi.\ (l = \texttt{"(PRINT (THEOREM }\phi\texttt{))"}) \land \mathsf{context\_ok}\ \pi \land \models_\pi \phi)\end{array}$$

We then prove that Milawa's top-level function, $\mathsf{milawa\_main}$, only produces output lines that satisfy $\mathsf{line\_ok}$, assuming that no runtime errors were raised during execution, i.e., that *ok* is true. Here $\mathsf{compute\_output}$ (definition omitted) is a high-level specification of what output lines coupled with their respective logical context the input *cmds* produces.

$$\begin{array}{l}\exists ans\ k\ output\ ok.\\ \quad \mathsf{milawa\_main}\ cmds\ \mathsf{init\_state} = (ans, (k, output, ok)) \land \\ \quad (ok \implies (ans = \mathsf{Sym}\ \texttt{"SUCCESS"}) \land \\ \qquad\qquad \mathsf{let}\ result = \mathsf{compute\_output}\ cmds\ \mathsf{in} \\ \qquad\qquad\quad \mathsf{every\_line}\ \mathsf{line\_ok}\ result \land \\ \qquad\qquad\quad output = \mathsf{output\_string}\ result)\end{array}$$

This approach works in part because Jitawa's print function, though used by Milawa's kernel, is not made available in the Milawa logic. In other words, a user-defined function can't trick us into invalidly printing `(PRINT (THEOREM ...))`.

This soundness theorem can be related back to the operational semantics of Jitawa through the following theorem, which was automatically derived by our tool for lifting deep embeddings into shallow embeddings:

$$\ldots \implies (\mathsf{Fun}\ \texttt{"MILAWA-MAIN"}, [\mathit{input}], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{milawa\_main}\ \mathit{input}\ \mathit{state})$$

6 Top-level soundness theorem

Now we are ready to connect the above soundness result to the top-level correctness theorem for Jitawa, which was proved in previous work [13]. Its top-level correctness theorem is stated in terms of a machine-code Hoare triple [11], which can informally be read as saying: if Jitawa's implementation is started from a state where enough memory is allocated (init_state) and the input stream of ASCII characters holds *input* for which Jitawa terminates, then either an error message is reported or a final state described by $\xrightarrow{\mathsf{exec}}$ is reached for which *ok* is true and *output* is the final state of the output stream (final_state).

$$\begin{array}{l} \{\, \mathsf{init\_state}\ \mathit{input} * \mathsf{pc}\ \mathit{pc} * \langle \mathsf{terminates\_for}\ \mathit{input} \rangle \,\} \\ \ \mathit{pc} : \mathsf{code\_for\_entire\_jitawa\_implementation} \\ \{\, \mathsf{error\_message} \vee \exists \mathit{output}.\ \langle ([], \mathit{input}) \xrightarrow{\mathsf{exec}} (\mathit{output}, \mathsf{true}) \rangle * \mathsf{final\_state}\ \mathit{output} \,\} \end{array}$$

Roughly speaking, $\xrightarrow{\mathsf{exec}}$ involves parsing some input, evaluating it with $\xrightarrow{\mathsf{ap}}$, and printing the result. By manually unrolling $\xrightarrow{\mathsf{exec}}$ to reveal the $\xrightarrow{\mathsf{ap}}$ relation for the call of `milawa_main`, it was straightforward to prove our top-level theorem relating Milawa's soundness down to the concrete x86 machine code.

This theorem, shown below, can informally be read as follows: if the ASCII input to Jitawa is the code for Milawa's kernel followed by a call to Milawa's main function on any input *input*, then the machine-code implementation for Jitawa will either abort with an error message, or succeed and print line_ok output (according to compute_output) followed by `SUCCESS`. Here strings are lists of characters, hence the use of list append (++) for strings.

$$\begin{array}{l} \forall \mathit{input}\ \mathit{pc}. \\ \quad \{\, \mathsf{init\_state}\ (\mathsf{milawa\_implementation} \mathbin{+\!\!+} \texttt{"(milawa-main '}\mathit{input}\texttt{)"}) * \mathsf{pc}\ \mathit{pc} \,\} \\ \quad\ \mathit{pc} : \mathsf{code\_for\_entire\_jitawa\_implementation} \\ \quad \{\, \mathsf{error\_message} \vee (\mathsf{let}\ \mathit{result} = \mathsf{compute\_output}\ (\mathsf{parse}\ \mathit{input})\ \mathsf{in} \\ \qquad\qquad\qquad\qquad \langle \mathsf{every\_line}\ \mathsf{line\_ok}\ \mathit{result} \rangle * \\ \qquad\qquad\qquad\qquad \mathsf{final\_state}\ (\mathsf{output\_string}\ \mathit{result} \mathbin{+\!\!+} \texttt{"SUCCESS"})) \,\} \end{array}$$

7 Quirks, bugs and other points of interest

We ran into some surprises during the proof.

Two minor bugs. No soundness bugs were found during our proof, but two minor bugs were uncovered and fixed. One was a harmless omission in the initial function arity table. The other allowed definitions with malformed parameter lists (not ending with `nil`) to be accepted. We don't see how these bugs could be exploited to derive a false statement, but the latter could probably have lead to undefined behavior when using a Common Lisp runtime, instead of our verified Lisp runtime.

Complication with termination obligations. In its current form, Milawa will only accept user-defined functions when their termination obligations are proven. However, the termination obligations can, in some cases, mention the function that is being defined. For instance, when defining a function like:

$$f(n,k) \;=\; \mathsf{if}\ n=0\ \mathsf{then}\ k\ \mathsf{else}\ f(n-1, f(n-1,k+1))$$

Milawa will require that the following termination condition has been proved for some measure function m:

$$\begin{aligned} n \neq 0 \Longrightarrow\ & m(n-1, f(n-1,k+1)) <_{\mathsf{ord}} m(n,k)\ \wedge \\ & m(n-1,k+1) <_{\mathsf{ord}} m(n,k) \end{aligned}$$

But note that this statement mentions function f, i.e., f ought to be part of the logical context π in order for this formula to be well-formed ($\mathsf{formula\_ok}$). Milawa's kernel gets around this problem by checking the proof of such termination obligations in a half-way state, where f is acceptable syntax but the defining equation is not yet available as a theorem. Our formalization of the logic checks the termination obligations in a similar half-way state: the termination obligations are checked in a state where the function's name is available in the context but the function body is set to None (Section 4.2).

Extensions. Once we had completed the full soundness proof, we took the opportunity to step back and consider what part of the system can be made better without complicating the soundness proof.

Evaluation through reflection: The original version of Milawa only used reflection to run the user-defined proof-checkers. However, one can equally well prove theorems by evaluation in the runtime, since all function defined in the logic also have a counter-part in the runtime. We have implemented and proved sound such an event handler (`admit-eval`).

Support for non-terminating functions: Note that our formalization of Milawa's logic only requires that there must exist an interpretation in HOL for each of the functions living in Milawa's logic. This means, e.g., that tail-recursive functions can be admitted without any proof of the termination obligations, because any tail-recursive function can be defined in HOL without a termination proof. We have proved that it is sound to extend a context with any recursive function that passes a simple syntactic check which tests whether all recursive calls are in tail position. This extension has not been implemented in the Milawa kernel because, if it were there, Milawa might not terminate. The precondition in the correctness theorem for our Lisp implementation requires that Milawa terminates for all inputs (Section 6).

8 Summary and related work

Davis' dissertation [2] describes how the Milawa theorem prover is constructed using self-verification from a small trusted kernel. In this paper, we have explained how we have verified in HOL4 that this kernel is indeed trustworthy. We have proved that the implementation of the Milawa theorem prover can never prove a statement that is false when it is run on Jitawa, our verified Lisp implementation. This theorem goes from the logic all the way down to the machine code. To the best of our knowledge, this is the most comprehensive formal evidence of a theorem prover's soundness to date.

Related work. The most closely related work is that of Kumar et al. [8] which aims to verify a similar end-to-end soundness result for a version of the HOL light theorem prover. Kumar et al. have a verified machine-code implementation of ML [9] (the dialect is called CakeML) and have an implementation of the HOL light kernel which has been proved sound w.r.t. a formal semantics of higher-order logic (HOL). At the time of writing, this CakeML project has not yet composed the correctness theorem for the ML implementation with the soundness result for the verified implementation of the HOL light kernel.

Kumar et al. based their semantics of HOL on work by Harrison [5], in which Harrison formalized HOL and proved soundness of its inference rules. Harrison's formalization did not include any definition mechanisms.

A reduced version of the Calculus of Inductive Constructions (CiC), i.e., the logic implemented by the Coq proof assistant, has also been formalized. Barras [1] has given reduced CiC a formal semantics in set theory and formalized a soundness proof in Coq. Recently, Wang and Barras [15] showed that the approach is modular and applied the framework to the Calculus of Constructions plus an abstract equational theory.

Milawa's logic is a simplified variant of the ACL2 logic. The ACL2 logic has previously been modeled in HOL, most impressively by Gordon et al. [3, 4]. In this work, ACL2's S-expressions and axioms are formalized as a shallow embedding in HOL. ACL2's axioms are proven to be theorems in HOL, and a mechanism is developed in which proved statements can be transferred between HOL4 and ACL2. Our work is in many ways cleaner, e.g., Milawa's S-expressions do not contain characters, strings or complex rationals, which clutter proofs. As part of our previous work on the verified Jitawa Lisp implementation, we proved that the axioms of Milawa (milawa_axioms from Section 4.4) are compatible with Jitawa's semantics. In the current paper, we went much further: we formalized the logic, proved soundness of all of Milawa's inference rules and proved soundness of the concrete implementation of Milawa w.r.t. Jitawa's semantics.

Other theorem prover implementations have also been verified. Noteworthy verifications include Ridge and Margetson [14]'s soundness and completeness proofs for a simple first-order tableau prover that can be executed in Isabelle/HOL by rewriting, and the verification of a SAT solver with modern optimizations by Marić [10]. Marić suggests that his SAT solver can be used as an automatically Isabelle/HOL-code-generated implementation.

Source code. Milawa's soundness proof and all auxiliary files are available at `http://www.cl.cam.ac.uk/~mom22/jitawa/`, and the Milawa theorem prover is available at `http://www.cs.utexas.edu/~jared/milawa/Web/`.

Acknowledgments. We thank Thomas Sewell for commenting on drafts of this paper. The first author was funded by the Royal Society, UK. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

1. Bruno Barras. Sets in Coq, Coq in sets. *J. Formalized Reasoning*, 3(1), 2010.
2. Jared C. Davis. *A Self-Verifying Theorem Prover.* PhD thesis, University of Texas at Austin, December 2009.
3. Michael J. C. Gordon, Warren A. Hunt Jr., Matt Kaufmann, and James Reynolds. An embedding of the ACL2 logic in HOL. In *International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, pages 40–46. ACM, 2006.
4. Michael J. C. Gordon, James Reynolds, Warren A. Hunt Jr., and Matt Kaufmann. An integration of HOL and ACL2. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 153–160. IEEE Computer Society, 2006.
5. John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2009.
6. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.
7. Matt Kaufmann and Konrad Slind. Proof pearl: Wellfounded induction on the ordinals up to $epsilon_0$. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 294–301. Springer, 2007.
8. Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2014. Submitted, waiting peer review.
9. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Peter Sewell, editor, *Principles of Programming Languages (POPL).* ACM, 2014.
10. Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50), 2010.
11. Magnus O. Myreen. Verified just-in-time compiler on x86. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Principles of Programming Languages (POPL).* ACM, 2010.
12. Magnus O. Myreen. Functional programs: conversions between deep and shallow embeddings. In *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2012.
13. Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2011.
14. Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2005.
15. Qian Wang and Bruno Barras. Semantics of intensional type theory extended with decidable equational theories. In *Computer Science Logic (CSL)*, volume 23 of *LIPIcs.* Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013.