Journal of Automated Reasoning — June 2015
Author's Pre-Print Version. Official version: doi:10.1007/s10817-015-9324-6

The reflective Milawa theorem prover is sound (down to the machine code that runs it)

Jared Davis · Magnus O. Myreen

Received 14 November 2014 / Accepted 3 March 2015 / Published June 2015

Abstract This paper presents, we believe, the most comprehensive evidence of a theorem prover's soundness to date. Our subject is the Milawa theorem prover. We present evidence of its soundness down to the machine code.

Milawa is a theorem prover styled after NQTHM and ACL2. It is based on an idealised version of ACL2's computational logic and provides the user with high-level tactics similar to ACL2's. In contrast to NQTHM and ACL2, Milawa has a small kernel that is somewhat like an LCF-style system.

We explain how the Milawa theorem prover is constructed as a sequence of reflective extensions from its kernel. The kernel establishes the soundness of these extensions during Milawa's bootstrapping process.

Going deeper, we explain how we have shown that the Milawa kernel is sound using the HOL4 theorem prover. In HOL4, we have formalized its logic, proved the logic sound, and proved that the source code for the Milawa kernel (1,700 lines of Lisp) faithfully implements this logic.

Going even further, we have combined these results with the x86 machine-code level verification of the Lisp runtime Jitawa. Our top-level theorem states that Milawa can never claim to prove anything that is false when it is run on this Lisp runtime.

Dedicated to John McCarthy (1927–2011)

The second author was partially supported by
the Royal Society UK and the Swedish Research Council.

Jared Davis
Centaur Technology, Inc., Austin TX, USA

Magnus O. Myreen
CSE Department, Chalmers University of Technology, Sweden
Computer Laboratory, University of Cambridge, UK

1 Introduction

The introduction to Hoare's seminal *An Axiomatic Basis for Computer Programming* [1] begins as follows:

> "Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."

This inspiring statement is, of course, not entirely true. Computer programs are run on electronic hardware. To truly understand the consequences of executing programs on these machines we would need theories from physics, an *inexact* science. Nevertheless, Hoare's view is still valuable as it splits the problem of trusting a program into two separate parts: the inexact question of whether the physical machine properly implements its instruction set, and the exact question of whether the program would, were it to be correctly executed, carry out the desired computation.

In this paper, we explain how we have taken the exact question to the extreme for an interactive theorem prover. Theorem provers are used to verify critical hardware and software systems, so their trustworthiness is a concern. Some popular examples of theorem provers are ACL2 [2], Coq [3], HOL4 [4], and Isabelle/HOL [5].

Our work targets the Milawa [6] theorem prover. Milawa is styled after Boyer-Moore systems like NQTHM [7] and ACL2, but unlike these programs it has a small logical kernel, somewhat like an LCF-style [8] system. Notably, its kernel includes a mechanism for performing reflection, an operation that modifies the kernel. Using this mechanism, Milawa's high-level tactics (like term rewriting) are proven sound and added into the kernel.

What would it mean for a theorem prover to be trustworthy? Roughly, we would like to know that the program will never claim to prove anything that isn't true. Any theorem prover is meant to mechanise reasoning in some particular mathematical logic; it is written in a programming language like ML, OCaml, or Lisp, and is executed by a runtime system like Poly/ML, the OCaml system, or Clozure Common Lisp. Accordingly, we should like to establish that:

A. the logic is sound and consistent,
B. the theorem prover's source code is faithful to its logic, and
C. the runtime executes the source code correctly.

We have used the HOL4 theorem prover to prove these properties about the Milawa theorem prover. We believe this work is the most comprehensive evidence of a theorem prover's soundness to date. This isn't to say you shouldn't trust other theorem provers—far from it! Many theorem provers are based on well-studied logics, effectively settling *A*, and LCF-style systems can make strong claims toward *B*. In fact, Harrison [9] has even mechanically proved *A* and a simplified *B* for HOL Light. (We discuss these and other approaches to developing trustworthy theorem provers in Section 2.) But our work is the first to also address property *C*, and really *C* is quite hard: how can we establish that a runtime for a language like ML or Lisp will properly execute programs? These runtime systems typically have many megabytes of source code and deeply depend on the operating system, C libraries, and so forth.

To avoid this complexity, we have developed our own Lisp runtime, Jitawa [10], for the explicit purpose of running Milawa. Jitawa features a just-in-time compiler down to x86-64 machine code, a copying garbage collector, efficient parsing, and support for large memory spaces. It is designed for minimal interaction with the operating system. It consists of 8,200 lines of HOL4-verified machine code, and a 200-line unverified C wrapper program that allocates memory before handing control to the verified code.

Our proofs of properties A through C are the key lemmas in a single, top-level HOL4 theorem: the kernel of the Milawa theorem prover, when run on our verified Lisp runtime Jitawa, 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. It relates the semantics of the logic (not just syntactic provability) all the way down to the concrete x86 machine code.

2 Current Approaches to Developing Trustworthy Provers

How can we develop reliable theorem provers? The approach we describe in this paper is to develop evidence, in the form of mechanically proved theorems, that indicates the theorem prover is sound. But there are many other techniques for making theorem provers trustworthy.

Property A, logical soundness, is perhaps the easiest part of the problem. Some theorem provers are based on well-known logics whose soundness is widely accepted, for instance first- or higher-order logic or ZF set theory. Soundness proofs are usually routine and are often short enough to write down and validate like ordinary mathematical proofs. Some soundness proofs have even been mechanically checked [11] by theorem provers.

Meanwhile, property B, the faithfulness of the prover's source code to its logic, is really not any different from any other software verification problem. That is, we may as well ask: how can we make sure that a compiler builds executables that implement their source code? Or, how can we be sure that an operating system will stop applications from tampering with each others' memory?

Well, there are plenty of *informal* ways to make programs more reliable. We can use languages with features that prevent certain kinds of bugs (strong type systems, garbage collection, arbitrary-precision arithmetic, etc.). We can write documentation, develop test suites, conduct code reviews, and apply static checking tools like linters. None of this will guarantee that the program is correct, but we can certainly have more confidence in a well-reviewed, well-tested program.

For theorem provers like ACL2 and PVS, and also for many reasoning tools like SAT solvers and SMT solvers, these informal approaches are the main ways of pursuing correctness. Usually these tools are pretty reliable, but sometimes [12, 13, 6, 14, 15] they have *soundness bugs*—bugs that can allow them to claim to have proven formulas that are not theorems.

Pragmatically, this is not so bad. Theorem provers have to cope with large problems like proofs about hardware and software. To make this feasible, a lot of work goes into making it easy to develop formal models and into making the reasoning algorithms powerful, efficient, and easy to use. With this in mind, we probably shouldn't obsess over correctness. If we are careful to review and test

our code then most problems will be found, and even if our program has a few esoteric bugs it will probably still be a significantly more reliable than human proof checking, which is ill-suited to the very long and detailed proofs that arise in, e.g., hardware verification.

This pragmatic view has many merits. But, of course, it would be nice to have a better story about why our theorem provers can be trusted. Fortunately, there are other ways to write theorem provers that can give us such a story.

One approach is to program the theorem prover's algorithms in such a way that they can justify their claims, and then to separately check that all of the steps in these justifications are valid. This reduces the question of trusting the theorem prover (a large program) to trusting the checking mechanism (a small one); if the theorem prover has a bug and tries to make some invalid inference, the checking program will fail to accept the justification, revealing the bug. When a theorem prover can produce output that a simple program can check, it is said to satisfy the *de Bruijn criteria.* [16]

Today, many reasoning tools, including SAT solvers [17], QBF solvers [18], SMT solvers [19], and resolution provers [20], can emit proof objects for other programs to check. A great feature of this approach is that since the reliability of the solving program does not really matter, we can freely use tools that are efficient but possibly unreliable. For instance, our solver could be a giant multithreaded C program with inline assembly that interfaces with GPUs and runs on a cluster. None of this poses a problem if the checking program is simple. In some cases, work has even gone into demonstrating the correctness of the checking programs. [20, 21, 22, 23]

For general-purpose, interactive theorem provers, there is an elegant way to embed the proof-checking system into the theorem prover itself, so that a separate program is not needed. This approach was pioneered by the LCF [8] system and today is used by HOL family of provers (HOL4, HOL Light, Isabelle, etc.). The idea is to encode the rules of inference for the logic as the only constructors for an abstract datatype, `thm`. The type system of the programming language should not allow `thm` objects to be built except by these constructors. This way, we can be sure any `thm` has been constructed by following the rules of inference.

The theorem prover with the best soundness story is HOL Light [9]. It is a *pure* LCF system, meaning that its `thm` type is very simple and does not provide unnecessary constructors, even though additional constructors could make it more efficient. Its kernel is especially small (about 400 lines of Objective Caml) and Harrison has modeled this core and mechanically proved [11] soundness properties about it.

It is worth noting that these LCF-style systems can be easily extended with logging mechanisms so the steps they have taken can be checked by other theorem provers. Using these mechanisms, it is possible to use HOL, HOL Light, and Isabelle/HOL to double check each others work, e.g., using the OpenTheory [24] framework.[1] However, these techniques currently struggle to scale to large proofs—proofs on the order of tens of thousands of lines of proof script.

All of these approaches leave open the question of Property C, whether the theorem proving program will be executed correctly. Virtually all interactive the-

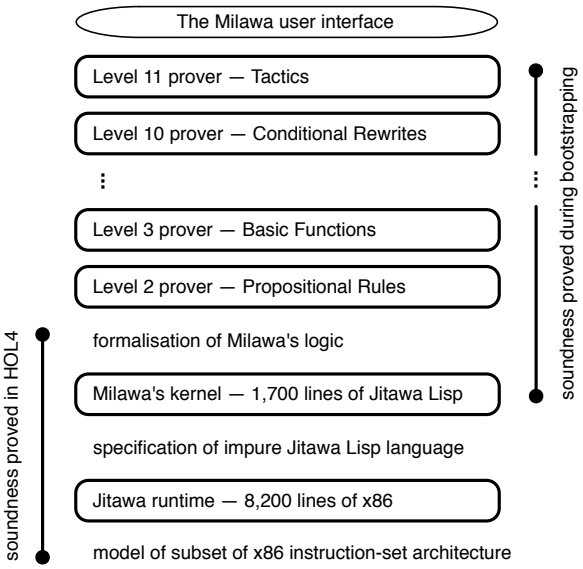
[1] Isabelle/HOL has a slightly more expressive logic than the other HOL provers and, as a result, it can import proofs but not always export to the other provers.

orem provers are implemented in functional programming languages. They rely heavily on the correctness of the abstraction layers provided by these languages and their implementations (called runtimes). In practice, these runtimes are simply just assumed to do the right thing, even though their implementations are often significantly more complex than the soundness-critical code in theorem provers! 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 need C compilers and various libraries. In contrast, LCF-style provers have at most a few thousand lines of soundness-critical code.

3 Our Approach — The Milawa Stack

We begin this paper with a brief introduction to the Milawa theorem prover (Section 4) which gives a flavor for how the system can be used to define new functions and prove theorems about them. Beyond this introduction, the rest of this paper is the story of why you should trust the Milawa theorem prover. The unusual part of our story is that we have mechanically proved theorems suggesting that Milawa is trustworthy. Our theorems start with the semantics of the logic and reach down to a detailed model of a subset of the 64-bit x86 instruction set architecture.

Here is a rough picture of the structure of the Milawa theorem prover and our evidence to suggest that it is sound.



Milawa's kernel sits at the centre of the stack. Extensions to the kernel (the Level 2-11 provers) sit on top of the kernel and below it lies our Lisp runtime, Jitawa. Note that Milawa's user interface is not part of our soundness story.

Our soundness story has two phases. The components above the kernel are installed during a *bootstrapping* phase. When the Milawa theorem prover is started, it runs through a bootstrapping sequence that replaces the initial kernel's proof

checker with more powerful checkers that add support for new operations (like rewriting). Each time the kernel is extended, the kernel's previous proof checker ensures the soundness of the new one before admitting it. When we write 'the Milawa theorem prover' we mean the result of running through this boostrapping process—that is, a program that can directly use high-level proof steps. We describe the kernel and this bootstrapping process in Sections 5 and 6.

The kernel itself and the components below it are verified in a separate *implementation* phase. For our story of trust, we assume that the Milawa theorem prover is run on top of our verified Lisp implementation called Jitawa (Section 7). As evidence of soundness, we have proved, using the HOL4 theorem prover, that (*A*) Milawa's logic is sound, (*B*) Milawa's kernel is faithful to the logic and (*C*) Jitawa executes Milawa's kernel correctly. We cover this work in Sections 8–13.

4 The User Interface

Before diving into our soundness story, we would like to begin by painting a picture of how the Milawa theorem prover is actually used. The human finds proofs with the help of a user interface (UI). In this section, we show how you could use this interface to define a simple list membership function (Section 4.2) and prove a basic theorem about it (Section 4.3). Using Milawa is so similar to using ACL2 that we also include a brief comparison between the two systems (Section 4.4).

To some degree, Milawa's interface mimics its kernel (which we will cover in Section 5). For instance, it records the functions you have defined and the theorems you have proven, prints messages saying it has accepted proofs, etc. But the interface is **not** part of our verified stack. In fact, for your convenience it is entirely unsound: you can skip proofs at any time, add arbitrary axioms, and so on. After you have used the interface to complete your proof, it can construct proof objects and save them into files that can be checked with the verified stack.

4.1 The Logic as a Programming Language

Milawa implements a computational logic of untyped recursive functions. By *computational*, we mean that functions in the logic can also be thought of as programs in a pure Lisp dialect.

The Milawa logic has three kinds of objects. The only numbers are the *naturals*, i.e., `0`, `1`, `2`, etc. Next we have *symbols* like `len`, `app`, `*` and `x`, which you can just think of as character strings. Together the naturals and symbols are called the *atoms*. Finally, we have *conses*, which are ordered pairs of objects. Every object is finite, i.e., there are no "circular" cons structures.

The symbol `nil` is special. It represents the empty list. It is also the only object that is regarded as false in Boolean contexts like `if` tests. Any other object is treated as true, but the symbol `t` is often used as the "canonical" true value. (This is similar to truth values in C programs, where 0 is considered false, other numbers are treated as true, and 1 is often used as the canonical true value.)

We write terms in the logic using a typical S-expression syntax derived from Lisp. We use symbols other than `t` and `nil` as variables. We write constants using the usual *quotation* mechanism from Lisp: this lets us refer to any Milawa object

as a constant literal by putting a quotation mark before it. For instance, while `foo` is a variable, `'foo` is a constant literal whose value, in any environment, is the symbol `foo`. Since there is no confusion between variables and numbers, we typically omit the quote and write, e.g., `'3` simply as `3`; similarly we may omit the quote for `t` and `nil`. We write the cons of a and b as `(`a `.` b`)`, and adopt the Lisp conventions for abbreviating lists, e.g., `(`a b c`)` means `(`a `.` `(`b `.` `(`c `.` `nil)))`. This way of writing conses works with quotation, so for instance `'(foo . bar)` is a constant whose value is the cons of the symbols `foo` and `bar`. Finally, we write function applications in a Lisp-like infix syntax, i.e., we write `(`f a b`)` instead of the more conventional $f(a, b)$.

The logic has a dozen primitive functions. Two key functions are:

`(equal x y)`	check whether `x` = `y`
`(if x y z)`	if `x` is true (non-`nil`) then `y`, else `z`

For working with conses we have:

`(consp x)`	check whether `x` is a cons
`(cons x y)`	construct the ordered pair `(x . y)`
`(car x)`	first component of an ordered pair
`(cdr x)`	second component of an ordered pair

For working with natural numbers, we have:

`(natp x)`	check whether `x` is a natural
`(< x y)`	check whether `x` is less than `y`
`(+ x y)`	add `x` and `y`
`(- x y)`	subtract `y` from `x`, or 0 when `y` $\geq$ `x`

Finally, for working with symbols, we have:

`(symbolp x)`	check whether `x` is a symbol
`(symbol-< x y)`	lexicographic ordering on symbols

An unusual feature of these primitives is that they are well-defined for all inputs, even inputs that might seem like they are ill-typed. For instance, the arithmetic functions `<`, `+`, and `-` treat non-numeric inputs as zero, so weird terms like `(+ 'foo '(a . b))` are allowed. Similarly, `symbol-<` treats non-symbols as `nil`, and `car` and `cdr` return `nil` when given atoms.

Aside from these primitives and a few *macros* for more concisely writing certain terms, everything else is defined. Usually we define *recursive* functions like `in`, below. We can also define *witness* functions to emulate quantifiers. We won't use witness functions in this introduction, but we will describe them more thoroughly in Section 5.4. We use the word "recursive" even when a function does not call itself, to distinguish between these "conventional" functions and witness functions.

4.2 Example: Defining a list-membership function

The Milawa user interface is a command-line program. When we invoke it, we are presented with a prompt:

```
MILAWA !>
```

Here we can type in commands to define functions, propose theorems, etc. Let's start by defining a list-membership function. Aside from the `:measure` part, this looks like a standard definition in Lisp or Scheme.

```
MILAWA !> (%defun in (a x)
            (if (consp x)
                (or (equal a (car x))
                    (in a (cdr x)))
              nil)
            :measure (len x))
```

Before a recursive function's definition can be accepted, we must prove that it always terminates for any possible input. This restriction lets us avoid certain logical problems that can arise from non-terminating functions. For example, a theorem in Milawa is that $a \neq$ `(+ 1 a)` for any a. If we could define a non-terminating function like `(f x)` = `(+ 1 (f x))`, then from its definition we could derive, for instance, `(f 0)` = `(+ 1 (f 0))`, contradicting our theorem.

When we propose a function definition with `%defun`, we specify a *measure* term that should be used for the termination proof. We are obliged to prove that (1) the measure always produces an ordinal, and (2) this measure decreases with each recursive call. Milawa's ordinals go up to ϵ_0, enough to admit many functions.

For the definition of `in`, we gave the measure `(len x)`. The `len` function is built into Milawa and it just computes the length of a list. Since there is only one recursive call in `in`, there are two goals to prove when we submit its definition:

```
Two goals remain.

  1. (EQUAL (ORDP (LEN X)) 'T)

  2. (IF (EQUAL (ORD< (LEN (CDR X)) (LEN X)) 'T)
         'T
         (IF (EQUAL (CONSP X) 'NIL)
             'T
             (EQUAL (NOT (EQUAL A (CAR X))) 'NIL)))
```

What is going on here? The `ordp` function checks whether its argument is a well-formed ordinal, so the first goal is to prove that the measure, `(len x)`, always produces an ordinal. Meanwhile, `ord<` is the less-than comparison for ordinal numbers. The second goal is hard to read because of the ugly `if` expressions, but boils down to showing that `(len (cdr x))` is smaller than `(len x)` when `in` recurs.

We can prove these goals by calling upon Milawa's *tactics*. There are many different tactics we can use. The `%split` tactic is good at cleaning up `if` expressions, and gives us a more readable second goal:

```
MILAWA !>(%split)
; Splitting clause 2.
; Splitting clause 1.
Two goals remain.

  1. (EQUAL (ORDP (LEN X)) 'T)
```

```
2. (IMPLIES (AND (NOT (EQUAL A (CAR X)))
                 (CONSP X))
            (EQUAL (ORD< (LEN (CDR X)) (LEN X)) 'T))
```

To prove these simplified goals, we can use a powerful tactic called `%crewrite`. This tactic uses a rewriter that makes and tracks assumptions, applies conditional rewrite rules, and evaluates ground terms. Milawa has thousands of rewrite rules about its built-in functions, and some of these rules are about `ordp`, `ord<` and `len`. Rules are organized into groups called *theories*; the usual theory is named `default`.

```
MILAWA !>(%crewrite default)
; Rewrote clause #2 in 0.001999 seconds (proved), [...]
; Rewrote clause #1 in 0.038994 seconds (proved), [...]
; Rewrote 2 clauses; 0 (+ 0 forced) remain.
All goals have been proven.
```

Rewriting with user-supplied rules is the main engine in Boyer-Moore style theorem provers like NQTHM, ACL2, and Milawa. When rewriting doesn't prove a goal outright, it often simplifies terms to leave us with a new goal that is easier to prove. It is often possible to prove the new goal by first augmenting the rewriter with additional rules. When Kaufmann, et al. wrote the main textbook about ACL2 [2], they named this approach *The Method*.

With the termination obligations out of the way, we can `%admit` the definition of `in`. When we do this, the UI can compile, check, and save Milawa proof objects for its termination obligations. It also adds the definition of `in` as an axiom so that we can start proving theorems about `in`:

```
MILAWA !>(%admit)
; Compiling worlds for IN...
; Compiling proofs for IN... [...]
;; Preparing to admit IN.
;; Proof sizes total: 3,409,472 conses [...]
; Checking the proofs... [...]
; Proof-checking completed.
;; Proofs accepted.  Saving as user/admit-in.proofs [...]
New rule: IN
```

To save time, we usually skip the proof building steps when we are initially developing proofs. But after our proof development is complete, we save the proofs into files and assemble them into an input that can be checked with the verified stack.

4.3 Example: Proving a theorem about `in`

Let's try to prove a simple theorem: no object is ever in itself. Intuitively, this is true because for a list to contain an element it must be "bigger" than that element. To make this argument precise, we use Milawa's built-in `rank` function, which counts the conses in an object:

```
(rank x) = (if (consp x)
               (+ 1 (+ (rank (car x))
                       (rank (cdr x))))
           0)
```

We start by proving a lemma toward our goal. Note that all variables in a Milawa theorem are implicitly universally quantified, so this lemma says: whenever any element `a` is in any list `x`, `x` is strictly larger than `a` in the sense of `rank`.

```
MILAWA !>(%defthm rank-when-in
           (implies (in a x)
                    (< (rank a) (rank x))))
One goal remains.

  1. (IMPLIES (AND (IN A X))
              (IFF (< (RANK A) (RANK X)) 'T))
```

To prove this lemma, we will induct according to `%cdr-induction`, an induction scheme that might normally be called structural induction on lists. To avoid too much detail, we follow up our induction scheme with a call of the `%auto` tactic, which applies a sequence of clause splitting, rewriting, and destructor elimination to simplify goals until no further progress is made. This leads us to the base case and inductive case that you might expect:

```
MILAWA !>(%cdr-induction x)
[... produces five subgoals ...]
MILAWA !>(%auto)
[... various progress messages ...]
Two goals remain.

  1. (IMPLIES (AND (NOT (CONSP X)))
              (NOT (IN A X)))

  2. (IMPLIES (AND (IN A (CONS X1 X2))
                   (NOT (< (RANK A)
                           (+ '1 (+ (RANK X1) (RANK X2))))))
              (IN A X2))
```

Each goal suggests a rewrite rule. We could solve the base case (1) by rewriting `(in a x)` to `nil` since `x` is not a cons. For the inductive case (2) we could rewrite `(in a (cons x1 x2))` to `(or (equal a x1) (in a x2))`, and the goal would follow by arithmetic.

We undo the current proof attempt, prove these rules, and try again. With the new rules in place, induction and `%auto` are enough to prove the theorem. We then tell the system to save the proof with `%qed`. As before, the interface constructs a proof object, checks it, saves it into a file, and adds the new rule.

```
MILAWA !>(%qed)
; Compiling worlds for RANK-WHEN-IN... [...]
; Preparing to check RANK-WHEN-IN.
;; Proof size: 4,712,680 conses.
; Checking the proof. [...]
;; Proof accepted.  Saving as user/thm-rank-when-in.proof
New rule: RANK-WHEN-IN
```

With our lemma established, we are ready to prove our goal:

```
MILAWA !>(%defthm not-in-self
           (not (in a a)))
One goal remains.

  1. (EQUAL (IN A A) 'NIL)
```

To prove this, we need to instantiate our lemma with $\{\texttt{x} \leftarrow \texttt{a}\}$:

```
MILAWA !>(%use (%instance (%thm rank-when-in) (x a)))
[ ... one goal with messy ifs ... ]
MILAWA !>(%split) ;; to clean it up
One goal remains.

  1. (IMPLIES (AND (IFF (< (RANK A) (RANK A)) 'T))
              (NOT (IN A A)))
```

This is true because the hypothesis is false: it is never the case that `(< a a)`. Tactics like `%crewrite` or `%auto` will notice this and prove the goal:

```
MILAWA !>(%crewrite default)
; Rewrote clause #1 in 0.001 seconds (proved), [...]
; Rewrote 1 clauses; 0 (+ 0 forced) remain.
All goals have been proven.
MILAWA !>(%qed)
; Compiling worlds for NOT-IN-SELF... [...]
;; Proof accepted.  Saving as user/thm-not-in-self.proof
New rule: NOT-IN-SELF
```

4.4 Comparison with ACL2

With a few changes these examples could have been about ACL2. Milawa is essentially a simplified reimplementation of ACL2. The two are quite similar in terms of their logics, the style of interaction, and the general approach to proving theorems by building up libraries of rewrite rules. Here are some specific differences:

- The ACL2 logic [2] has some objects that are not in Milawa, like characters, strings, and numbers besides the naturals. However, it would be straightforward to encode all "good" ACL2 objects as Milawa objects. ACL2 also has `encapsulate` [25] and functional instantiation, which aren't in Milawa.
- The ACL2 system has a much richer connection to its Lisp runtime than Milawa. ACL2 functions can interact with files [26], use arrays, and use type declarations [27] to avoid runtime type checks and bignum arithmetic. With experimental extensions, ACL2 functions can use parallelism [28] or hash-consing and memoization [29]. Milawa only has pure functions.
- Milawa does not reimplement parts of ACL2's reasoning engine such as its `type-set` and forward-chaining algorithms. ACL2's rewriter is also more efficient, has an integrated arithmetic procedure [30], and supports meta rules [31] and generalized equivalences [32]. ACL2 also has features such as trusted clause processors [33] that allow it to trust external tools like SAT solvers.

– ACL2 has richer symbols and user-defined macros. Milawa has only a small set of built-in macros: `first`, ..., `fifth`, `and`, `or`, `list`, `let`, `let*`, and `cond`.

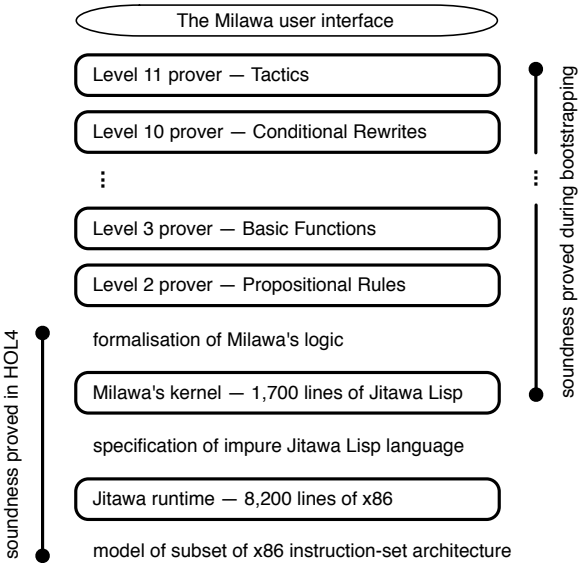
With all of this missing, it is worth noting that Milawa *does* reimplement much of ACL2. Its rewriter [6] makes assumptions, canonicalizes `equal` and `iff`-equivalent terms, mimicks ACL2's *ancestors check* for avoiding rewriting loops, handles rules with free variables and syntactic restrictions [31], and can forcibly assume hypotheses. It has tactics for destructor elimination, generalization, and fertilization. Its `%waterfall` tactic does depth-first rewriting and case-splitting. Its verified tactics are largely similar to verified clause processors [33] in ACL2.

Another difference between the two systems is that ACL2 is a single program. This program is both the user interface (with unsound features like skipping proofs, adding axioms, and redefining functions) and also the certifier. Once a file of proofs has been developed, ACL2 can be asked to certify it, and during this certification only *embeddable events*, which are meant to be sound, are accepted. In contrast, Milawa's UI is entirely separate from its certification program, its kernel.

5 The Kernel

We have now seen how Milawa's user interface can be used to develop proofs. But this UI isn't verified, has features like skipping proofs that are clearly unsound, and is really just a tool for constructing inputs to the verified stack.

The rest of this paper is about our verified stack. Recall the picture of the Milawa Stack from Section 3:



We begin, in this section, by focusing on the *kernel*, which provides a critical layer of abstraction in our verification. Everything above the kernel—the higher level provers and all of Milawa's proof-finding algorithms—is verified by running the kernel to check proofs in the Milawa logic. Everything below—the kernel's

source code and the Jitawa runtime that executes this code—is separately verified with HOL4.

The kernel is a command-line program. It is implemented in a simple Lisp dialect (Section 5.1) that is closely related to the Milawa logic. The first part of this program is a simple proof checker (Section 5.2) that accepts only fully expansive, formal proofs in the Milawa logic. On top of this, there is a command loop (Section 5.3) that processes definitions and theorems, occasionally calling upon the proof checker to handle proof obligations.

Most of this is straightforward, but one command deserves special attention, namely, `switch`. The `switch` command (Section 5.4) implements Milawa's reflection mechanism by allowing the kernel to switch to user-defined proof checking functions. We explain how this command works, its relation to the programming language, and what the kernel requires for soundness.

5.1 The Programming Language and its Relation to the Logic

Milawa's kernel is a text file with 1,700 lines of Lisp source code. This source code is written in the Lisp dialect of Jitawa, our verified Lisp runtime (Section 7). Alternately, it can be run on an (unverified) Common Lisp system; this requires an additional 300 lines of Common Lisp code that set up a package, define the Jitawa primitives, and reconfigure the Lisp parser. This extra Common Lisp code isn't verified and we don't think of it as part of the kernel.

Throughout the kernel, we assume that all Lisp objects are made of natural numbers, symbols, and non-circular conses. When we run the kernel with Jitawa, this assumption automatically holds since these are the only kinds of objects that Jitawa implements. But Common Lisp runtimes can have other kinds of objects (e.g., negative numbers), so our extra Common Lisp code explicitly checks that inputs are acceptable before passing them to kernel functions.

It is no accident that these acceptable Lisp objects correspond exactly the objects in the Milawa logic. Jitawa also has primitive functions (`equal`, `if`, `cons`, ...) and macros (`and`, `let`, ...) that match those of the Milawa logic (Section 4.1).

This leads to a kind of duality between Milawa logic functions and Jitawa functions. Now, there are a few special Jitawa routines that aren't in the logic (`define`, `print`, `funcall`, etc.), and the Milawa logic has witness functions that aren't like programs. But aside from these, Milawa logic functions have Jitawa equivalents. Similarly, while we think of the kernel as a Jitawa Lisp program, most of its functions have corresponding definitions in the logic.

5.2 The Proof Checker

The first 800 lines of Milawa's kernel define a proof-checking function called `logic.proofp`. The "`logic.`" part of the name is just a prefix we use for functions that deal with the logic, and "`proofp`" is short for *proof predicate*. This function checks whether a Lisp object represents a valid proof in the Milawa logic.

Like any mathematical logic, Milawa's logic has a syntactic definition of formulas, some formulas that are axioms, and some rules of inference for proving formulas (Section 8). This makes it easy to write a proof checker. We start with

functions to see if an object is a well-formed formula. We represent a proof as a tree of steps, where each step has the formula being derived and a justification. For each inference rule, we write a function to see if a proof step obeys the rule. Finally, `logic.proofp` walks over an alleged proof to check if each step is valid.

None of this is anything novel. Even before electronic computers, Gödel wrote a proof checker in the proof of his incompleteness theorem. So in this section we will only sketch how the proof checker works; for a full description see Chapter 3 of Davis' dissertation [6].

Terms and Formulas. Milawa's logic has two kinds of syntactic entities. The *terms* can be constants, variables, function applications, and λ-applications that are really just `let` expressions. The *formulas* are equalities between terms, or negations or disjunctions of other formulas. The kernel has functions to recognize terms and formulas:

`(logic.termp x)`	is `x` a syntactically valid term?
`(logic.formulap x)`	is `x` a syntactically valid formula?

These functions are *almost* everything we need to recognize well-formed syntax. Since we use an S-expression format we don't need a scheme for determining operator precedence. Since our terms and formulas are untyped we don't need a type context to determine well-formedness. But for proper syntax we *do* require all function applications to invoke known functions on the right number of arguments. To check this, the kernel has separate functions that take an *arity table* (usually called `atbl`) that has the names and arities of known functions:

`(logic.term-atblp x atbl)`	are arities ok within term `x`?
`(logic.formula-atblp x atbl)`	are arities ok within formula `x`?

Proof Syntax. We represent proofs as trees of proof steps. We call each proof step an *appeal* since it represents an appeal to some rule of inference. Each appeal has four components:

method	a symbol that names the rule being used,
conclusion	the formula that is being proved,
subproofs	proofs that should establish the rule's premises,
extras	any additional information, e.g., the substitution list for an instantiation rule

The kernel has various functions for working with these appeal structures. For instance, the recognizer `(logic.appealp x)` determines if `x` has the proper syntax for an appeal. There are also accessor functions for these components, e.g., `(logic.method x)` gets the method from an appeal structure `x`.

Proof-Step Validity. For each rule of inference, the kernel has a function that checks whether an appeal is a valid use of that inference rule. A few examples are shown in Figure 1 and discussed below.

The kernel uses `logic.axiom-okp` to check an appeal to an axiom. Here, `axioms` is assumed to be the current list of axioms (in practice it will contain the built-in axioms and also a *definitional* axiom for each function that has been defined).

```
(define logic.axiom-okp (x axioms atbl)
  (let ((method     (logic.method x))
        (conclusion (logic.conclusion x))
        (subproofs  (logic.subproofs x))
        (extras     (logic.extras x)))
    (and (equal method 'axiom)
         (equal subproofs nil)
         (equal extras nil)
         (memberp conclusion axioms)
         (logic.formula-atblp conclusion atbl))))

(define logic.theorem-okp (x thms atbl)
  (let ((method     (logic.method x))
        (conclusion (logic.conclusion x))
        (subproofs  (logic.subproofs x))
        (extras     (logic.extras x)))
    (and (equal method 'theorem)
         (equal subproofs nil)
         (equal extras nil)
         (memberp conclusion thms)
         (logic.formula-atblp conclusion atbl))))

(define logic.expansion-okp (x atbl)
  (let ((method     (logic.method x))
        (conclusion (logic.conclusion x))
        (subproofs  (logic.subproofs x))
        (extras     (logic.extras x)))
    (and (equal method 'expansion)
         (equal extras nil)
         (tuplep 1 subproofs)
         ;; Does the conclusion have the form "A or B,"
         ;; where B is the conclusion of the subproof?
         (let ((b (logic.conclusion (first subproofs))))
           (and (equal (logic.fmtype conclusion) 'por*)
                (equal (logic.vrhs conclusion) b)
                ;; Also, is A a well-formed formula?
                (logic.formula-atblp (logic.vlhs conclusion) atbl))))))

(define logic.appeal-step-okp (x axioms thms atbl)
  (let ((how (logic.method x)))
    (cond ((equal how 'axiom)
           (logic.axiom-okp x axioms atbl))
          ((equal how 'theorem)
           (logic.theorem-okp x thms atbl))
          ((equal how 'expansion)
           (logic.expansion-okp x atbl))
          ...)))
```

Fig. 1 Examples of Proof Checking Functions

As a practical matter, the kernel records each theorem that it has accepted and, via `logic.theorem-okp`, it allows previously proven theorems to be used as proofs. Here, `thms` is assumed to be the list of all current theorems.

The function `logic.expansion-okp` is used to check appeals to the *expansion* rule of inference. This rule is simply:

$$\frac{B}{A \vee B} \quad \text{(for any formulas } A \text{ and } B\text{)}$$

The function just checks whether the conclusion of the appeal has the form $A \vee B$ where B is the conclusion of the sub-proof, and makes sure that A is a well-formed formula.

There are similar functions for the other rules of inference. These functions are then combined to form `logic.appeal-step-okp`, which can check whether an arbitrary appeal is valid by just looking at its *method* and then invoking the appropriate function.

The Proof Checker. The top-level proof checking function is

```
(logic.proofp x axioms thms atbl),
```

and it simply calls `logic.appeal-step-okp` on every step of a proof. All of the functions we have discussed so far, including `logic.proofp`, can be regarded as functions in the logic. We will soon see (Section 5.4) how this is used to allow the kernel to be reflectively extended.

5.3 The Command Loop

Beyond the proof-checker, the kernel has a command-processing loop that lets you define functions and check alleged proofs. This part of the kernel makes use of special Lisp primitives like `define`, `funcall`, and `error` that are not part of the Milawa logic, so unlike `logic.proofp` it does not have corresponding Milawa logic definitions.

There are four kinds of commands:

`(verify name formula proof)`
: Prove a theorem.

`(def name formals body measure proofs)`
: Define a recursive function and prove it terminates.

`(witness name bound-var free-vars body)`
: Define a witness function (to emulate a quantifier).

`(switch name)`
: Use a new proof-checking function (Section 5.4).

The command-processing loop tries to accept a list of these commands. A command can only be accepted when it meets certain requirements. For instance, a `verify` command must have a well-formed `formula` and a valid `proof` whose conclusion is `formula`. When requirements aren't met, the kernel calls `error` to abort with an error.

Most of these requirements depend on the current *state* of the system. For instance, whether or not a formula is well-formed depends on what functions have been defined. The kernel maintains an explicit `state` object that has the following components:

axioms	the current list of axioms (formulas)
thms	the current list of theorems (formulas)
atbl	the current arity table, which governs well-formed syntax
checker	the current proof checker (initially `logic.proofp`)
ftbl	the function table of definitions given to the Lisp system

When commands are accepted the state is updated, e.g., after a `verify` command has been accepted, its `formula` will be added to *thms*. Similarly, a successful `def` or `witness` command results in a new formula, called a *definitional axiom*, to be added to the *axioms* and a new entry to be added to the *atbl*.

5.4 Reflection and the `Switch` Command

The most interesting part of Milawa's kernel is its `switch` command. This command lets us replace `logic.proofp` with a user-defined proof-checking function. That is, after a `switch`, the kernel will use the new function instead of `logic.proofp` to check the proofs of theorems in `verify` commands, and also to check the termination proofs for `def` commands.

How is this useful? The basic idea is that a new proof checker can be "more powerful" than `logic.proofp`. The proofs that `logic.proofp` accepts must be carried out in full and are very long. In practice, this limits our ability to construct and check proofs. New proof checkers, in contrast, can accept derived rules of inference, and hence can permit shorter proofs. These new rules can be very powerful. For instance, in Section 6, we describe extended proof checkers that can directly use rewriting, case splitting, etc.

How can this be sound? Roughly speaking, before the kernel will accept a command like `(switch new-proofp)`, the user must first prove that `new-proofp` can only accept theorems. As a basic syntactic criteria, the kernel insists that `new-proofp` is the name of a defined function which, like `logic.proofp`, has four arguments, e.g.,

```
(new-proofp x axioms thms atbl).
```

Then, to establish that `new-proofp` is semantically acceptable, i.e., it does not accept proofs of non-theorems, the user must have already proved a formula called the *fidelity claim*. This formula is roughly:

$$
\begin{array}{l}
\forall x, \mathit{axioms}, \mathit{thms}, \mathit{atbl} : \\
\quad \texttt{(new-proofp } x \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\
\qquad \Longrightarrow \\
\left(\begin{array}{l}
\exists p : \texttt{(logic.appealp } p\texttt{)} \\
\qquad\quad \wedge \texttt{(logic.proofp } p \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\
\qquad\quad \wedge \texttt{(logic.conclusion } p\texttt{)} = \texttt{(logic.conclusion } x\texttt{)}
\end{array} \right)
\end{array}
$$

Informally, this means that whenever the new proof checker accepts some appeal x as a proof, there really does exist some fully detailed proof p of the same formula that is accepted by `logic.proofp`. When a `switch` command meets these criteria, the kernel updates its state so that the *checker* becomes `new-proofp`.

The real fidelity claim is slightly more complicated than the above because the Milawa logic does not have explicit quantifiers. Instead, all free variables in a Milawa formula are implicitly universally quantified, so we can't directly write the above because of the nested $\exists p$.

Instead, we emulate the quantifier using a witness (Skolem) function. Section 8.3 has detailed semantics, but intuitively a witness function just chooses an element a that satisfies a predicate $P(a)$ when such an a exists. In this case, we use

the witness function

$$\texttt{(logic.provable-witness } \phi \texttt{ axioms thms atbl)},$$

which chooses a valid proof of formula ϕ with respect to these particular `axioms`, `thms`, and `atbl` when such a proof exists. More formally, the definitional axiom for `logic.provable-witness` in the Milawa logic can be understood as follows:

$$\begin{array}{l}
\forall \phi, \mathit{proof}, \mathit{axioms}, \mathit{thms}, \mathit{atbl}: \\
\quad \textsf{let } \mathit{wit} = \texttt{(logic.provable-witness } \phi \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\
\quad \textsf{in} \\
\quad\quad \left(\begin{array}{l} \texttt{(logic.appealp } \mathit{proof}\texttt{)} \\ \quad \wedge \texttt{(logic.proofp } \mathit{proof} \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\ \quad \wedge \texttt{(logic.conclusion } \mathit{proof}\texttt{)} = \phi \end{array} \right) \\
\quad\quad\quad \Longrightarrow \\
\quad\quad \left(\begin{array}{l} \texttt{(logic.appealp } \mathit{wit}\texttt{)} \\ \quad \wedge \texttt{(logic.proofp } \mathit{wit} \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\ \quad \wedge \texttt{(logic.conclusion } \mathit{wit}\texttt{)} = \phi \end{array} \right)
\end{array}$$

To read this axiom, notice that we quantify over all possible objects *proof*. If there is *any* such object that is a valid proof of ϕ (the hypothesis), then we may conclude that *wit*, the object returned by `logic.provable-witness`, is also a valid proof of ϕ. On the other hand, if there is no way to prove ϕ, then this implication is vacuous and tells us nothing about *wit*.

With this witness in place, it is straightforward to formally define provability as an ordinary, recursive function:

$$\begin{array}{l}
\forall \phi, \mathit{axioms}, \mathit{thms}, \mathit{atbl}: \\
\quad \texttt{(logic.provablep } \phi \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\
\quad\quad = \\
\quad \textsf{let } \mathit{wit} = \texttt{(logic.provable-witness } \phi \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\
\quad \textsf{in} \\
\quad\quad \left(\begin{array}{l} \texttt{(logic.appealp } \mathit{wit}\texttt{)} \\ \quad \wedge \texttt{(logic.proofp } \mathit{wit} \ \mathit{axioms} \ \mathit{thms} \ \mathit{atbl}\texttt{)} \\ \quad \wedge \texttt{(logic.conclusion } \mathit{wit}\texttt{)} = \phi \end{array} \right)
\end{array}$$

How does this work? Suppose ϕ really is a provable formula. Then *wit* must be a valid proof of ϕ, so it will satisfy the conjuncts. Otherwise, since ϕ is not provable, we don't know anything about *wit*. But whatever it happens to be, it can't satisfy these conjuncts: if it did, then it would be a valid proof of ϕ, which we have assumed does not exist.

Connecting Lisp and the Logic. To support reflection, care is taken so that every function defined in the Milawa logic has a corresponding definition in the Lisp runtime.

In the case of an ordinary `def` command, the kernel executes a Jitawa Lisp `define` command to instruct the Lisp system to introduce a new executable function with the given name, formals, and body. The *ftbl* records each definition that was given to the Lisp, and also ensures that the user cannot redefine kernel functions or use names that are Jitawa primitives like `error`.

The Lisp definitions for `witness` commands are more subtle. Witness functions allow you to emulate quantifiers by choosing an element a that satisfies a predicate $P(x)$ when such an a exists. Take, for instance, `logic.provablep-witness`, introduced just above. From a logical perspective, the definitional axiom for this function lets you choose some proof of any provable formula ϕ. Unfortunately, this axiom doesn't suggest any algorithm that we could implement as a program that returns a proof for any provable ϕ.

Although we can't magically synthesize a Jitawa analogue for a witness function that computes real witnesses, we do still introduce a Jitawa analogue that simply calls the `error` primitive to say it cannot be executed because it is a witness function. Why do we even bother? One reason is that nothing in Milawa prohibits ordinary recursive functions in Milawa from calling witness functions—for instance, `logic.provablep` calls `logic.provable-witness`. This error-causing definition ensures that if we ever attempt to execute a function like `logic.provablep`, a sensible error will be produced. (ACL2 follows this same approach.)

Altogether this connection ensures that any successful executions of the Jitawa analogues of Milawa logic functions do, in fact, accurately reflect the logical definitions. This is paramount to the correctness of Milawa's reflection mechanism. Consider:

- Before the `switch` command will allow some `new-proofp` function to be installed, it requires that the fidelity claim for `new-proofp` has been established using the current proof checker.
- This fidelity claim is, like any formula, a logical statement; it pertains to the *logical definition* of `new-proofp`.
- Based on this fidelity claim, the kernel will subsequently begin executing the *Jitawa analogue* of `new-proofp` to check proofs.

In other words, this connection between the Lisp and the logic plays a fundamental role in ensuring that the fidelity claim is actually meaningful, and allows the kernel to be extended only with verified code.

5.5 Relation to LCF-style systems

Milawa's kernel has many similarities to an LCF-style [8] theorem prover such as HOL or HOL Light. Theorems in these systems are instances of the `thm` datatype, and can only be constructed by using a handful of constructor functions. In a *pure* LCF-style system, the only constructors implement the primitive rules of inference; an *impure* system might include additional constructors, e.g., to allow proofs to be constructed more efficiently.

Milawa's initial proof checker, `logic.proofp`, is quite like the `thm` type. Both of these are meant to ensure valid reasoning by requiring all proofs to be carried out in full. In practice, this means developing *fully expansive* proof-finding tactics that can justify their claims by building acceptable proofs or `thm`s.

Of course, there are also many differences.

LCF-style systems have an efficiency advantage in that intermediate proof steps can be garbage collected once they are no longer needed, whereas Milawa's style of proof objects must exist in full in the proof files to be checked. Proofs in higher order logics are also likely to be much shorter than the proofs `logic.proofp`

accepts: natural-deduction style logics are quite convenient in comparison with Milawa's sentence-style logic; they also support more powerful theorems [34] than a first-order system like Milawa, and these theorems can shorten proofs.

On the other hand, Milawa's very simple untyped logic has a much more direct connection to the programming language. It also allows us to quote arbitrary terms in a very direct and easy way, without having to deal with types. This kind of a logic seems like a natural starting point when exploring the development of theorem provers based on reflection.

6 The Bootstrapping Process

We have now seen two complementary systems. Milawa's user interface (Section 4) allows you to develop proofs using tactics like `%use` and `%auto` that are similar to Boyer-Moore style proof automation. In contrast, its kernel (Section 5) requires you to give it a proof of every alleged theorem, and just checks these proofs.

In this section, we explain the bootstrapping process that connects these two systems. This process involves using Milawa's reflective kernel to verify all of the tactics of the user interface. It culminates in the verification and installation (via the `switch` command) of a *level 11 proof checker*. This so-called "proof checker" is really a theorem prover; it can apply whole sequences of Milawa's tactics as a single proof step.

6.1 Planning the Proof

Milawa's theorem proving tactics are complex programs, so it is challenging to prove properties about their behavior. Meanwhile, the kernel only accepts fully expansive proofs, which are large and difficult to construct. We begin by developing a detailed, informal proof plan. This planning process separates the intellectual task of discovering why Milawa's tactics are sound from the engineering task of constructing a formal proof that Milawa's kernel can accept.

We develop our proof plan in ACL2. ACL2's logic is so similar to Milawa's that it is quite easy to model Milawa in ACL2. While ACL2 is normally thought of as a formal verification tool, we want to stress that we are using it only **informally** as a familiar, mature environment for sketching the proof—our final evidence of Milawa's soundness does *not* require any trust in ACL2.

What does it mean to verify a tactic? The basic goal is to show that any formula the tactic claims is true can indeed be justified using the rules of the Milawa logic. Our approach is constructive. First, we write a fully expansive version of the tactic. Then, we show that for all sensible inputs, the fully expansive version produces a valid `logic.proofp`-style proof, and this proof has "the right conclusion."

As an example, Milawa's rewriter can evaluate ground terms to constants, e.g., given `(fact 5)`, it can produce `120`. The claim being made here that the formula, "`(fact 5)` = `120`" is provable. To verify evaluation, we first write a fully expansive evaluator. Whereas our ordinary evaluator will produce `120` when given `(fact 5)`, this new function instead constructs a `logic.proofp`-style proof that concludes "`(fact 5)` = `120`".

In our proof plan, we use ACL2 to show that:

1. If the definitions used to evaluate some term, x, are valid—i.e., they involve only well-formed formulas according to the `atbl` and are all among the `axioms` given to `logic.proofp`—then our fully expansive evaluator produces a valid proof—i.e., the object it produces is accepted by `logic.proofp`.
2. If x' is the result of evaluating x, then our fully expansive evaluator proves the right formula, i.e., the conclusion of its proof is $x = x'$.

In the course of our ACL2 proof plan, we develop fully expansive versions of all of Milawa's tactics, and sketch out proofs like the above. This is some work. As groundwork, we implement many derived rules of inference as functions which we call *builders* (Section 6.2). These builders serve as very useful abstractions for verification, and allow us to develop a compositional approach to verifying builders (Section 6.3). Through considerable engineering, we follow this approach to verify all of Milawa's tactics (Section 6.4).

With the proof plan in place, all that remains is the engineering task of constructing a formal proof that Milawa's kernel can accept. We will return to that in Section 6.5, where we will put these fully expansive tactics to another use!

6.2 Implementing Derived Rules

We implement derived rules of inference using *builder* functions. A builder function takes as inputs proofs of its premises, and perhaps other various related terms or formulas. From these inputs, it carries out some sequence of proof steps that results in the desired proof.

Let's see an example. We will use two primitive rules of the Milawa logic:

$$\frac{}{\neg A \vee A}\ \text{Propositional Schema} \qquad \frac{A \vee B \quad \neg A \vee C}{B \vee C}\ \text{Cut}$$

Using these rules, we will derive a new rule:

$$\frac{A \vee B}{B \vee A}\ \text{Commutativity of Or}$$

Here is a derivation of this rule, and a corresponding builder function:

1. $A \vee B$	Given	
2. $\neg A \vee A$	Prop. Schema	
3. $B \vee A$	Cut lines 1, 2	

```
(define build.commute-or (x)
  (let* ((a (logic.vlhs (logic.conclusion x)))
         (line1 x)
         (line2 (build.propositional-schema a))
         (line3 (build.cut line1 line2)))
    line3))
```

Notice that to carry out the primitive steps, `build.commute-or` calls upon `build.cut` and `build.propositional-schema`. These functions are *primitive builders* which simply `cons` together an appeal structure that is suitable for `logic.proofp`. In contrast, `build.commute-or` itself is a *derived builder*—a function that constructs its proofs by calling other builders.

When we write subsequent derived builders, we can freely use primitive or derived builders that we have already implemented. This is convenient in that it

allows us to reuse common proof patterns, but it is still a fully expansive way of building proofs. That is, the proof objects that our builders generate are carried out in full, using only the primitive steps that `logic.proofp` accepts.

Milawa's derived builders are similar to derived rules in an LCF-style system like HOL. One difference is that builders construct new proof objects for `logic.proofp` to check later, whereas derived rules in HOL construct new `thm` instances that are checked as they are built. A more important difference is that Milawa's builders are ordinary functions in the Milawa logic, so we can reason about them. In contrast, derived rules in HOL are ML programs that construct ML objects. They aren't defined inside of higher-order logic, so the prover cannot directly reason about them.

6.3 Compositional Verification of Derived Rules

This ability to reason about builders plays a key role in the verification of Milawa. We are generally not concerned with the details of *how* a particular builder function constructs its proof. But we would like to know that, under suitable input constraints, each builder creates a proof that is

- **well typed**: it meets the basic structural constraints of `logic.appealp`,
- **relevant**: it has the desired conclusion, and
- **faithful** to the logic: it only uses valid inferences accepted by `logic.proofp`.

For primitive builders like `build.propositional-schema` that simply `cons` together new appeal objects, we establish these properties by structural arguments that involve, for instance, the definitions of functions like `logic.appealp` and `logic.appeal-step-okp`. Here are what these properties look like, in our ACL2 proof plan, for the propositional schema:

(well-typed)
```
(logic.formulap a)
    ⟹ (logic.appealp (build.propositional-schema a))
```

(relevant)
```
(logic.conclusion (build.propositional-schema a))
    = (logic.por (logic.pnot a) a)                    ;; i.e., ¬A ∨ A
```

(faithful)
```
(logic.formula-atblp a atbl)
    ⟹ (logic.proofp (build.propositional-schema a)
                    axioms thms atbl)
```

Meanwhile, here are the three theorems for `build.commute-or`:

(well-typed)
```
(logic.appealp x) ∧
(logic.fmtype (logic.conclusion x)) = 'por*      ;; given A ∨ B
    ⟹
(logic.appealp (build.commute-or x))
```

(relevant)
```
(logic.conclusion (build.commute-or x))
    =
(logic.por (logic.vrhs (logic.conclusion x))     ;; derive B ∨ A
           (logic.vlhs (logic.conclusion x)))
```

(faithful)
```
(logic.proofp x axioms thms atbl) ∧
(logic.fmtype (logic.conclusion x)) = 'por*
    ⟹
(logic.proofp (build.commute-or x) axioms thms atbl)
```

While the statements of the three theorems look similar for primitive and derived builders, their proofs are very different. For derived rules, we carry out these proofs entirely by appealing to the (already proven) three theorems for each subsidiary builder. There is no need to involve the definitions of functions like `logic.appeal-step-okp` or to consider the particular `cons` structures being produced. There is also no need to consider the details of how the subsidiary builders operate.

This is important because it means the verification approach is modular and compositional: even if some subsidiary builder is large and complicated, this complexity is hidden away during later proofs about other builders that make use of it. Also, we can freely modify a builder, e.g., to use a shorter derivation, without impacting the proofs for builders that depend on it; we will soon see (Section 6.5) that it is useful for builders to produce short proofs.

Following this basic approach, in our proof plan, we develop and verify a collection of efficient builders to carry out basic propositional reasoning steps, to reason about equality, and to reason about core functions like `if` and `equal`. Beyond these simple derivations, we also develop other rules with inductive derivations. For instance, the Subset Rule is:

$$\frac{A_1 \vee \cdots \vee A_n}{B_1 \vee \cdots \vee B_m}, \text{ where } \{A_1, \ldots, A_n\} \subseteq \{B_1, \ldots, B_m\}$$

We implement inductive derivations as recursive builders, and verify them using inductive proofs. Some other useful inductive derivations include the Tautology Rule, equality substitution throughout terms, and evaluation of ground terms using a McCarthy [35] style evaluator and a fully expansive function that can build corresponding proofs of evaluations.

Students in introductory logic courses are often asked to find such derivations. When developing our basic builders, we were able to draw upon substantial previous work. Milawa's rules of propositional logic are taken essentially from Shoenfield [36], and many of our builders follow derivations used by Shankar [37] in his proof of Gödel's theorem, by Boyer and Moore [38] in their work on NQTHM, and by Kaufmann, Manolios, and Moore [2] in their description of the ACL2 logic.

6.4 From Derived Rules to Theorem Proving Tactics

The builders we have described so far allow us to carry out "forward" directed reasoning. By this we mean that they allow us to prove new formulas by combining together other formulas that we already have proven. However, to implement effective proof automation, we really would like to be able to work "backward" from a goal. For instance, in Section 4 we saw how the user interface allows us to submit goals and then simplify these goals, using tactics like `%split` and `%crewrite` to create new subgoals. This kind of backward proof search is common throughout all interactive theorem provers.

To support backward-directed reasoning, we develop functions for working with clauses. A *clause* in Milawa is a disjunction of literals. Each *literal* is technically a formula of the form $term \neq \texttt{nil}$, but informally we think of each literal simply as a term. Since disjunction is associative and commutative, there isn't an explicit

notion of the *hypotheses* or *conclusion* of a clause as there are in sequents. However, in the user interface clauses are displayed as implications by arbitrarily taking the last literal as the conclusion, and the (negated) previous literals as the hypotheses. Any Milawa formula can be "compiled" into an equivalent clause by propositional reasoning and the axioms about `equal`, and `if`.

All of Milawa's high-level *tactics* operate on clauses. In the LCF [8] system, a tactic t is a function that takes a goal to prove, g, and produces (1) a list of subgoals, $g_1, \ldots, g_n$, which should together imply g, and (2) a function, v, called a *validation*, which given proofs of $g_1, \ldots, g_n$ should construct a proof of g. Tactics in Milawa are similar, but since Milawa is first order we cannot implement validations as higher-order functions. Instead, each Milawa tactic is a pair of functions: one which like t reduces a goal to some subgoals, and one which like v justifies this reduction by proving g when given proofs of $g_1, \ldots, g_n$. Like builders, tactics are ordinary functions in the Milawa logic; we can reason about them and establish their validity, i.e., that the validation function really can prove the original goals when it is given proofs of the subgoals.

The general idea, then, is as follows. If we want to prove some arbitrary formula A, we begin by compiling it into a goal clause. We then apply a sequence of tactics, e.g., `%crewrite`, `%split`, etc., which may split the goal into new subgoals, simplify those subgoals, etc. Once we have eliminated all of the goals, we chain together the validation functions for the tactics, in the reverse order, to construct proofs. Eventually this produces a proof of our goal clause, which we can translate into a proof of our goal formula using the builder function for our formula compiler. The user interface manages this process, recording the order of tactic application and the arguments to tactics in a *proof skeleton* structure.

Many of Milawa's tactics are very simple. For instance, the `%use` tactic simply extends a goal clause with an extra literal that is an instance of a previously proven theorem. This may seem silly: to prove A, we will instead prove $\mathit{true} \vee A$. But actually `:use` hints are very commonly used in ACL2 proofs as a way to guide the prover to the "right" instantiations of theorems; `%use` is similarly useful in Milawa, for instance see the proof in Section 4.3. Milawa's `%generalize` tactics, named after ACL2's *generalization* procedures, simply replace an arbitrary subterm with a fresh variable everywhere throughout a clause. The `%fertilize` tactic, which is something like ACL2's *cross fertilization* procedure, just eliminates a top-level "hypothesis" of the form `(equal x y)` by replacing all occurrences of x with y throughout the clause. Milawa's `%elim` and `%conditional-eqsubst` tactics, which are like ACL2's *destructor elimination* procedure, are only slightly more complex.

More sophisticated tactics do much more to automatically simplify clauses. For instance, the `%cleanup` tactic:

- Standardizes "not-variants" like `(equal x nil)` and `(if x nil t)` to `(not x)`.
- Eliminates double negations like `(not (not x))`.
- Removes (proves) any clauses with "obviously true" literals like `5` $\neq$ `nil`.
- Removes (proves) any clauses with complementary literals like `a` and `(not a)`.
- Removes any "absurd" literals like `nil` or `(not 5)`, i.e., simplifies $A \vee \mathit{false} \rightsquigarrow A$.
- Removes any repeated literals, i.e., simplifies $A \vee A \rightsquigarrow A$.
- Removes any "subsumed" clauses (supersets of other clauses).

Meanwhile, the `%split` tactic extends these cleaning steps with a case splitting mechanism. That is, it can lift the conditions of `if` expressions to the top level.

This is a basic mechanism that lets us reduce goals like $P \rightarrow$ `(if a b c)` into separate subgoals like $P, a \rightarrow b$ and $P, \neg a \rightarrow c$.

Milawa's most complex tactics are its rewriters. The conditional rewriter, `%crewrite`, walks over a clause, $L_1 \vee ... \vee L_n$, rewriting each literal L_i in turn. When it rewrites L_1, it can assume that all of the other literals are false. Similarly, when it encounters a term of the form `(if a b c)`, it can assume a is true while rewriting b, and that a is false while rewriting c.

The rewriter tracks these assumptions using an *assumptions structure*, which has disjoined-set style data structures to track `equal` and `iff`-equivalent terms. That is, if we first assume `(equal a b)` and then assume `(equal b c)`, the assumptions structure will subsequently know that `a` and `c` are also equal. The assumptions system helps the rewriter to canonicalize equal and equivalent terms to their simplest syntactic forms. For instance, if the rewriter encounters `c`, the assumptions system could reduce it to `a`.

To verify the assumptions system, we develop a "slow" version of it that records all of the explicit assumptions that were made, and that builds *equivalence traces* that capture, at a somewhat high level, how conclusions like `(equal a c)` were reached. These traces can then be compiled into fully expansive proofs. In contrast, the "fast" version of the assumptions structure does not record how it arrived at its conclusions. It is verified by showing that it always reaches the same conclusions as the slow version.

The rewriter itself is quite complex. It can make assumptions, evaluate ground terms, and use previously proven rules. If these rules have free variables, the rewriter will try to intelligently instantiate them using suitable terms it has made assumptions about. Rules can have *backchain limits* to prevent the rewriter from spending excessive time relieving their hypotheses, or can have *forced* hypotheses which will be turned into new subgoals when the rewriter cannot prove them. The application of rules can be syntactically restricted so that rules like `(+ a b)` = `(+ b a)` do not loop. There is also a sophisticated *ancestors checking* heuristic that can avoid subtle backchaining loops.

The verification of the rewriter is very much like the verification of the assumptions system. We develop a "slow" version of the rewriter that builds *rewrite traces*. These traces explain, in high level but detailed steps, how it has arrived at its conclusion, and they contain enough information to be compiled into formal proofs. In contrast, the "fast" version of the rewriter does not record how it has carried out its work, and is verified by showing it computes the same results as the slow rewriter.

For a full treatment of Milawa's rewriter and other tactics, see Chapters 7-10 of Davis's dissertation [6].

6.5 Formalizing the Proof Plan

With the intellectual task of developing our proof plan completed, we are ready to turn our attention to the engineering task of constructing a formal proof.

How can we do this? Consider that Milawa is styled after ACL2 and that, during the course of our ACL2 proof plan, we have implemented fully expansive versions of all of Milawa's tactics. So here is the idea. First, we will redo the ACL2 proof plan using Milawa's (untrustworthy) user interface. Then, since we already

have fully expansive versions of Milawa's tactics, we'll use them to emit a formal version of the Milawa proof, which we can check with the kernel.

This idea almost works.

Following the ACL2 proof plan with Milawa was sometimes tricky, but ultimately successful. Milawa is meant to be like ACL2, and we developed tools for keeping our Milawa proof in sync with the ACL2 proof. But ACL2 really has quite a lot of features. To be able to follow the ACL2 proof sketch, we sometimes modified Milawa's tactics to make them more like ACL2, e.g., we extended Milawa's rewriter with ancestors checking, free variable matching, and forcing. We also sometimes reworked our ACL2 proofs to avoid features that would be hard to reimplement, such as its arithmetic procedure and type reasoning system. In the end, our Milawa proof matches our ACL2 proof almost lemma for lemma.

However, the idea of using the fully-expansive versions of Milawa's tactics to produce `logic.proofp`-style proofs turned out to be impractical. Even despite work to make Milawa's builders efficient, the proofs become overwhelmingly large.

Fortunately, we can avoid needing to create fully expansive proofs by taking advantage of the kernel's reflection mechanism. That is, instead of trying to verify Milawa's tactics directly with `logic.proofp` style proofs, we first introduce and verify a sequence of increasingly capable proof checkers. We use the word *levels* to describe this sequence—that is, `logic.proofp` is the Level 1 proof checker; the objects it accepts are Level 1 proofs and may use only Level 1 proof steps; the Level 1 steps correspond to the primitive rules of the logic.

At each new level in the sequence, we allow new kinds of proof steps to be used. For instance, the Level 2 proof checker can carry out many derived rules of inference, such as the *Commutativity of Or*, as a single step. Because of our existing work to verify our builders and tactics, we can easily to develop an ACL2 proof sketch of the *fidelity claim* (Section 5.4) that is necessary to install the new proof checker. After porting this proof sketch to Milawa, we can emit a Level 1 proof of the fidelity claim for Level 2. Even though this is a fully expansive proof, it is small enough to practically construct and check. (Section 6.6)

This is progress. By taking advantage of these new steps, Level 2 proofs can be written more concisely than Level 1 proofs. For instance, *Right Associativity* is a derived rule of inference that takes eight level 1 steps to carry out. But Level 2 proofs can use this rule directly, as a single step. This savings is realized for every use of the new rule. This means it is practical to build and check more difficult proofs in Level 2 than in Level 1. A Level 2 proof of Milawa's fidelity is still too large to construct, but it is possible to introduce additional intermediate proof checkers, each more capable than the last. (Section 6.7).

6.6 Defining and Verifying New Proof Checkers

The Level 2 proof checker accepts all of the primitive rules of inference, and also accepts 26 new derived rules of inference, each of which is a simple propositional manipulation. These rules include the Commutativity of Or rule (Section 6.2), Modus Ponens, and a few critical rules that are heavily used in tactics like `%split`. Even this modest selection of rules makes proofs for Level 2 much shorter than proofs for `logic.proofp`.

```
(define build.commute-or-okp (x)
  (let ((method     (logic.method x))
        (conclusion (logic.conclusion x))
        (subproofs  (logic.subproofs x))
        (extras     (logic.extras x)))
    (and (equal method 'build.commute-or)
         (equal extras nil)
         (equal (len subproofs) 1)
         (let ((subconc (logic.conclusion (car subproofs))))
           (and (equal (logic.fmtype conclusion) 'por*)
                (equal (logic.fmtype subconc) 'por*)
                (equal (logic.vlhs conclusion) (logic.vrhs subconc))
                (equal (logic.vrhs conclusion) (logic.vlhs subconc)))))))

(define level2.step-okp (x axioms thms atbl)
  (let ((method (logic.method x)))
    (cond ((equal method 'build.commute-or)
           (build.commute-or-okp x))
          ((equal method 'build.right-expansion)
           (build.right-expansion-okp x atbl))
          ;; ... cases for all the other new rules ...
          (t
           ;; otherwise it is not a new step, but we will still
           ;; accept all Level 1 steps
           (logic.appeal-step-okp x axioms thms atbl)))))
```

Fig. 2 Core of the Level 2 Proof Checker

The definition of the Level 2 proof checker is very similar to `logic.proofp`. In particular, recall from Section 5.2, and Figure 1 the definitions of

- individual step checking functions like `logic.expansion-okp`, and
- the wrapper function for arbitrary Level 1 proof steps, `logic.appeal-step-okp`.

For Level 2, we write:

- new individual step-checking functions like `build.commute-or-okp`, and
- a new wrapper that recognizes any Level 2 step, `level2.step-okp`.

The functions are shown in Figure 2. The level 2 proof checker, `level2.proofp`, simply applies `level2.step-okp` everywhere throughout a proof.

Notice that, besides accepting the new Level 2 steps, the Level 2 proof checker also still accepts all Level 1 steps. This is not strictly necessary. It would be perfectly safe, though not useful, to install a new proof checker that doesn't accept any proofs at all! However, as a general convenience, we prefer to arrange our higher-level provers to also accept lower-level steps.

In Milawa's kernel, before we can use the `switch` command to begin using `level2.proofp` to check proofs, we must first show that it only accepts provable formulas. This boils down to showing that each new individual `step-okp` function satisfies the following property: whenever `*-step-okp` accepts an appeal x, and all of the subproofs of x are provable, then the conclusion of x is also provable. This, in turn, is an easy corollary of the three theorems (Section 6.3) for each corresponding builder.

The successful translation of the fidelity claim for the Level 2 proof checker was an important landmark in our bootstrapping process. Early in the project, we

had serious concerns about whether we could practically emit a fully expansive proof showing the fidelity of a more powerful proof checker. In total, the proof involves 421 definitions and 4188 theorems, but most of these definitions and theorems (about 80%) are from Milawa's core utilities and logic libraries and are not specifically about the new Level 2 rules and builders. The proofs are 1.7 GB on disk.

6.7 Summary of Proof Checkers

The bootstrapping process introduces several new proof checkers. Here is a brief summary of the new rules we add at each level:

Level 2	Basic propositional rules
Level 3	Rules about equality and basic functions like `equal`, `if`, etc.
Level 4	Miscellaneous groundwork, mostly clause support
Level 5	Equivalence traces (rewrite assumptions), clause updating
Level 6	If-lifting, core clause splitting algorithm
Level 7	Full `%split` tactic (with if-lifting and clause cleaning)
Level 8	Rewrite traces (the major steps in slow rewriting)
Level 9	Unconditional rewriting (fast, trace-free version)
Level 10	Conditional rewriting (fast, trace-free version)
Level 11	All remaining tactics

The selection of rules to include into each level is guided by pragmatics. At each level, we are looking to add new rules that will make the new proof checker more powerful and able to accept much shorter proofs. However, we cannot add rules whose verification would be too complex for the current proof checker. To identifying important rules to focus on, we developed tools to profile the sizes of proofs and to explain how much of a proof was due to each kind of tactic; we then studied the control flow of these tactics to look for builders that we could approach. For a detailed description of work, see Chapter 12 of Davis [6].

Higher-level proofs are typically more concise, faster to construct, and faster to check than lower-level proofs. Because the amount of improvement realized at each level depends upon the particulars of the proof being constructed, it is not possible to make broad statements like "Level n proofs are 35% smaller and can be checked 20% more quickly than Level $n-1$ proofs." For instance, in Level 6 we verified our clause splitting algorithm. This can result in considerable improvements in proofs that make heavy use of clause splitting, but will not appreciably impact a proof that is mainly carried out by rewriting.

Even so, we can at least illustrate the impact of higher-level proof checkers on an example proof. For this example we consider the proof of faithfulness of our evaluator, which is normally a moderately difficult Level 7 proof. We instruct the user interface to construct the proof at each level. We can then compare the sizes of these proofs and how long it takes to build and check them. Attempts to construct a Level 1 proof failed with more than 25 GB allocated.

Level	Build Time (sec)	Size (million conses)	Check Time (sec)
1	∅	∅	∅
2	6,238	8,289	31,451
3	2,879	4,310	5,323
4	2,279	1,117	2,816
5	2,157	1,049	3,120
6	1,482	426	2,737
7	768	222	1,874
8	691	171	1,430
9	167	129	440
10	65	58	457
11	8	27	163

The full bootstrapping proofs end up being quite large. Taken all together, there are 2,081 definitions and 13,563 theorems. The collected proof files come to 8.4 GB of disk space.

Milawa's bootstrapping process reduces the question of trusting an interactive theorem prover to that of trusting its kernel. In the remainder of the paper, we turn our attention to the lower levels, e.g., the correctness of the kernel and the runtime beneath it.

7 The Runtime

The Milawa theorem prover was originally programmed to run on top of Common Lisp implementations [6]. However, in order to extend Milawa's soundness story down into the programming language layers, we have implemented a simpler Lisp runtime called Jitawa [10]. Jitawa was designed to be simple enough to be verified, yet sufficiently realistic to be able to host Milawa's kernel and run through its boostrapping process.

In this section, we describe Jitawa and the requirements the Milawa theorem prover puts on the underlying programming language and runtime. In Section 9, we present Jitawa's formal specification and, in Section 12, we describe how Jitawa's x86 implementation was constructed and verified.

7.1 Language Requirements

On the face of it, Milawa is quite modest in what it requires of the underlying Lisp runtime. Most of the code for its kernel 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 datatypes (natural numbers, symbols, and conses), and involve a handful of primitive functions and macros like `car`, `+`, `list`, and `cond` corresponding to the primitives and macros of Milawa's logic. To run these functions we just need a basic functional programming language that implements these primitives and macros.

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:

– Milawa requires that the underlying Lisp system compiles user-supplied functions as they are defined, which is important for running new proof checkers.
– Milawa dynamically calls either `logic.proofp` or whichever proof checker has been most recently installed via `switch` to check proofs.
– Milawa aborts with a runtime error when invalid events or proofs are encountered, or if an attempt is made to run a witness function.

We did not see a good way to avoid any of this. Accordingly, the hosting Lisp system must support at least: on-the-fly compilation of user-defined functions, dynamic function invocation, and some way of causing runtime errors.

7.2 Jitawa Lisp

The Lisp language Jitawa implements is described formally in Section 9, however, we summarize its main features here. Jitawa implements the following primitve Lisp functions and macros that correspond directly to the Milawa theorem prover's primitives and macros, which were described in Section 4.1.

```
equal if consp cons car cdr natp < + - symbolp symbol-<
let let* cond or and list first second third fourth fifth
```

Jitawa also features a few primitive functions that are not present in Milawa's logic, namely:

`(define name vars body)`	defines a new function in Jitawa
`(print x1 x2 ... xn)`	prints `x1`...`xn` as a line of output
`(error x1 x2 ... xn)`	same as print but also exits Jitawa
`(funcall fname x1 ... xn)`	calls function `fname` with `x1`...`xn`

Note that all of these functions evaluate their arguments. In particular, this means that definition of most functions use quotes to avoid unwanted evaluation:

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

Jitawa includes a macro called `defun` which expands to a call to `define` with a quote inserted for each argument. The line above could equally well have been written:

```
(defun increment (n) (+ n '1))
```

Dynamic function calls are performed using the `funcall` primitive, e.g. `(funcall exp n)` returns the same as `(increment n)` if `exp` evaluates to the symbol `increment`.

7.3 Designed 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. A previously verified Lisp interpreter [39] was hopelessly inadequate: its direct interpreter approach was too slow, and it also had inherent memory limitations (due to the 32-bit architectures) 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 performance, functions are dynamically compiled to 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 quantities of memory. In particular, lexing is merged with parsing into what is called a scanner-less parser, and abbreviations are supported efficiently.
- Running out of heap space or stack space is a real concern; we ensure graceful exits in all circumstances and helpful error messages if limits are reached.

We believe these design decisions were necessary for our Lisp implementation to be able to scale to the computationally heavy task of running through Milawa's boostrapping process.

7.4 I/O Requirements and Interaction

In Milawa's original trusted core, each `def` 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 bootstrapping process 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—can be very large and repetitive. As a simple but crucial optimization, an abbreviation mechanism [29] 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 latest 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, the underlying Lisp runtime needs to be able to parse input files that are gigabytes in size and involve hundreds of millions of abbreviations.

For Jitawa, we decided to read all input from `stdin` and write all output to `stdout`. Jitawa provides a read-eval-print loop. Here is an example run. Lines starting with `>` are user input and the other lines are output from Jitawa.

```
> '3
3
> (cons '5 '(6 7))
(5 6 7)
> (defun increment (n) (+ n '1))
NIL
> (increment '5)
6
> (funcall 'increment '5)
6
> '(#1=(a b c) #1#)
((a b c) (a b c))
```

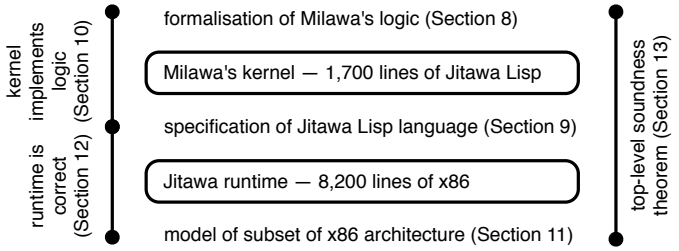
Jitawa's verified implementation makes calls to (unverified) external C functions which our proofs assume perform a few basic I/O operations correctly, e.g. to read a line of input and output a string.

7.5 Jitawa Implementation in Numbers

Jitawa's implementation is setup as a small C file (200 lines of C and 200 lines of assembly) which allocates the necessary heap space and then starts the verified x86 machine code (8,200 x86 instructions). A full run through Milawa's bootstrapping process takes 5 days, which is approximately eight times slower than doing the same with CCL — a state-of-the-art Common Lisp implementation.

8 The Logic the Kernel Implements

Now that we have explained the role of each component in the Milawa stack (user interface, theorem prover, kernel and runtime), we turn our attention to the formal specification and verification that build up to our top-level HOL theorem about the stack's trustworthiness (Section 13). We structure the next few section as a top-down descent through the layers.



We begin, in this section, with the specification of the Milawa logic and its semantics. The Milawa logic is a first-order logic of *untyped* recursive functions with induction up to ϵ_0, similar to the logics of NQTHM and ACL2. We have used HOL to formalize the syntax (Section 8.1), semantics (8.3), and rules of inference (8.4) of the Milawa logic, and to mechanically prove the soundness of its inference rules (8.5) and definition principle (8.6).

8.1 Syntax

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

sexp ::=	Val *num*	$\mathbb{N}$ numbers
	| Sym *string*	symbols
	| Dot *sexp sexp*	cons pairs
prim ::=	If | Equal | Not | Symbolp | Symbol_less	
	| Natp | Add | Sub | Less | Consp | Cons	
	| Car | Cdr | Rank | Ord_less | Ordp	
func ::=	PrimitiveFun *prim*	primitives
	| Fun *string*	user-defined
term ::=	Const *sexp*	constant
	| Var *string*	variable
	| App *func* (*term* list)	function app.
	| LambdaApp (*string* list) *term* (*term* list)	(λformals. 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 may not refer to variables besides its formal parameters. This requirement makes substitution straightforward.

The term_ok and formula_ok well-formedness predicates depend on a *logical context*, π, which will be explained below.

8.2 Context

The definition of the syntax, semantics and inference rules all depend on information regarding the user-defined functions. To keep the formalisation 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*, holds the syntax of the definition for the function. This *func_body* holds the right-hand side of a definition in case this is a conventional function definition based on an equation, or a variable name and property if this is the definition of a witness function (these witness functions are similar to Hilbert's choice in HOL). For reasons that will be explained in Section 8.6, 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.

The end of the next section defines a well-formedness criteria which relates the interpretation functions with the syntax in *func_body*.

8.3 Semantics

Next, we define a semantics of Milawa's formulas. In this section we start with the top-level definition before explaining the auxiliary definitions. The top most definition defines when a Milawa formula is valid: a Milawa formula p is *valid*, written $\models_\pi p$, if and only if p is syntactically correct w.r.t. the context π and p evaluates to true in context 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 evaluation of formulas, i.e. eval_formula $i\ \pi$, as follows using a function for evaluation of terms, i.e. eval_term $i\ \pi$. The syntax overloading can be confusing in the following definition. On the left-hand side $\neg$, $\vee$ and = are the constructors for the *formula* type, while on the right-hand side $\neg$ and $\vee$ are the usual Boolean connectives and $=$ is the equality predicate (over the *sexp* type, in this case).

$$\begin{array}{lcl}
\mathsf{eval_formula}\ i\ \pi\ (\neg p) & = & \neg(\mathsf{eval_formula}\ i\ \pi\ p) \\
\mathsf{eval_formula}\ i\ \pi\ (p \vee q) & = & \mathsf{eval_formula}\ i\ \pi\ p \vee \mathsf{eval_formula}\ i\ \pi\ q \\
\mathsf{eval_formula}\ i\ \pi\ (x = y) & = & (\mathsf{eval_term}\ i\ \pi\ x = \mathsf{eval_term}\ i\ \pi\ y)
\end{array}$$

We define term evaluation w.r.t. a specific variable instantiation i next. Here $[[v_0, \ldots, v_n] \mapsto [x_0, \ldots, x_n]]$ is a function which maps v_i to x_i, for $0 \leq i < n$, and all

other variable names to `NIL`. Below map is a function such that map $f\ [t_0, \ldots, t_n] = [f\ t_0, \ldots, f\ t_n]$.

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

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:

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

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:

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

Note that this is a non-trivial equation since term, which appears on the right-hand side of the equation, can refer to *interp* via app. Indeed, proving soundness of the definition principle boils down to showing that the termination obligations always imply that such an interpretation function exists (Section 8.6).

A similar condition applies to witness functions. If

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

is true then the following implication must hold. This implication states that if there exists some value v that makes *prop* true when variable names $var :: formals$

are substituted for values $v :: args$ in $prop$, then $interp(args)$ returns some such v. Here the test for 'is true' is Lisp's truth test, i.e. 'not equal to NIL'.

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

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.

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

8.4 Inference Rules

Our top-level soundness theorem states that the kernel's implementation is sound w.r.t. the semantics of the Milawa logic, as defined above. In this section, we formalise the syntactic inference rules that Milawa's implementation uses for proof checking. We also explain how we have proved that these inferences rules are sound w.r.t. the semantics.

We use HOL to formlise the inferences rules as an inductively defined relation $\vdash_\pi$. Judgments are of the form $\vdash_\pi formula$, where π is the logical context. Our formalization of the logic has 13 inference rules. The simplest six are listed below. Here formula_ok is used to ensure that all derivable judgments are syntactically correct.

$$\frac{\vdash_\pi a \vee (b \vee c)}{\vdash_\pi (a \vee b) \vee c}\ \text{(associativity)} \qquad \frac{\vdash_\pi a \vee b \qquad \vdash_\pi \neg a \vee c}{\vdash_\pi b \vee c}\ \text{(cut)}$$

$$\frac{\vdash_\pi a \vee a}{\vdash_\pi a}\ \text{(contraction)} \qquad \frac{\mathsf{formula\_ok}_\pi\ a \qquad \vdash_\pi b}{\vdash_\pi a \vee b}\ \text{(expansion)}$$

$$\frac{\mathsf{formula\_ok}_\pi\ a}{\vdash_\pi \neg a \vee a}\ \text{(prop. schema)} \qquad \frac{a \in \mathsf{milawa\_axioms}}{\vdash_\pi a}\ \text{(basic axiom)}$$

Above, milawa_axioms is a set consisting of the 56 axioms from Davis [6]. 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.

The inference rule for functional equality is defined using an auxiliary function which expands into disjunctions of inequalities:

$$\mathsf{neqs}\ [(x_1, y_1), \ldots, (x_n, y_n)]\ z = \neg(x_1 = y_1) \vee \ldots \vee \neg(x_n = y_n) \vee z$$

Functional equality is formalised as follows, using map as described earlier and fst and snd such that $\mathsf{fst}\ (x, y) = x$ and $\mathsf{snd}\ (x, y) = y$.

$$\frac{\mathsf{formula\_ok}_\pi\ (\mathsf{neqs}\ l\ (\mathsf{App}\ f\ (\mathsf{map\ fst}\ l) = \mathsf{App}\ f\ (\mathsf{map\ snd}\ l)))}{\vdash_\pi \mathsf{neqs}\ l\ (\mathsf{App}\ f\ (\mathsf{map\ fst}\ l) = \mathsf{App}\ f\ (\mathsf{map\ snd}\ l))}$$

The next inference rules require substitution through terms and formulas, $\mathsf{term\_sub}\ \sigma\ t$ and $\mathsf{formula\_sub}\ \sigma\ a$, where σ is a list of pairs (var, exp). Note that substitution does not go into the body of lambda expressions because substitution will only be applied to terms that satisfy term_ok, i.e. terms where all of the variables in the body of the lambda terms are bound by the lambda.

$$\begin{array}{lcl}
\mathsf{lookup}\ (v, [], r) & = & r \\
\mathsf{lookup}\ (v, (var, exp) :: \sigma, r) & = & \mathsf{if}\ v = var\ \mathsf{then}\ exp\ \mathsf{else\ lookup}\ (v, \sigma, r) \\
\\
\mathsf{term_sub}\ \sigma\ (\mathsf{Const}\ c) & = & \mathsf{Const}\ c \\
\mathsf{term_sub}\ \sigma\ (\mathsf{Var}\ v) & = & \mathsf{lookup}\ (v, \sigma, \mathsf{Var}\ v) \\
\mathsf{term_sub}\ \sigma\ (\mathsf{App}\ f\ xs) & = & \mathsf{App}\ f\ (\mathsf{map}\ (\mathsf{term_sub}\ \sigma)\ xs) \\
\mathsf{term_sub}\ \sigma\ (\mathsf{LambdaApp}\ vs\ x\ xs) & = & \mathsf{LambdaApp}\ vs\ x\ (\mathsf{map}\ (\mathsf{term_sub}\ \sigma)\ xs) \\
\\
\mathsf{formula_sub}\ \sigma\ (\neg p) & = & \neg(\mathsf{formula_sub}\ \sigma\ p) \\
\mathsf{formula_sub}\ \sigma\ (p \vee q) & = & \mathsf{formula_sub}\ \sigma\ p \vee \mathsf{formula_sub}\ \sigma\ q \\
\mathsf{formula_sub}\ \sigma\ (x = y) & = & \mathsf{term_sub}\ \sigma\ x = \mathsf{term_sub}\ \sigma\ y
\end{array}$$

Milawa's inference rules for instantiation and beta reduction require that no ill-formed terms are introduced; that's why they mention formula_ok.

$$\frac{\vdash_\pi a \qquad \mathsf{formula\_ok}_\pi\ (\mathsf{formula\_sub}\ \sigma\ a)}{\vdash_\pi (\mathsf{formula\_sub}\ \sigma\ a)}\ \text{(instantiation)}$$

$$\frac{\mathsf{formula\_ok}_\pi\ (\mathsf{LambdaApp}\ xs\ a\ ys = \mathsf{term\_sub}\ (\mathsf{zip}\ (xs, ys))\ a)}{\vdash_\pi \mathsf{LambdaApp}\ xs\ a\ ys = \mathsf{term\_sub}\ (\mathsf{zip}\ (xs, ys))\ a}\ \text{(beta red.)}$$

The next rule is the most exotic inference rule: base evaluation. This rule lets us apply primitive Lisp functions to constant S-expressions. Here eval_primitive is the function, mentioned in the previous section, that evaluates Lisp primitives.

$$\frac{\mathsf{arity}\ op = \mathsf{length}\ args}{\begin{array}{l}\vdash_\pi \mathsf{App}\ (\mathsf{PrimitiveFun}\ op)\ (\mathsf{map\ Const}\ args) = \\ \quad \mathsf{Const}\ (\mathsf{eval\_primitive}\ op\ args)\end{array}}\ \text{(base eval.)}$$

Milawa's most sophisticated rule enables induction w.r.t. less-than over the ordinals up to ε_0. The rule Milawa adopts is similar to the induction rule of ACL2, and Chapter 6 of Kaufmann, et. al [2] nicely introduces this rule. Here ordp and ord_less abbreviate applications of primitives Ordp and Ord_less,

$$\begin{array}{lcl}
\mathsf{ordp}\ m & = & \mathsf{App}\ (\mathsf{PrimitiveFun\ Ordp})\ [m] \\
\mathsf{ord_less}\ m\ n & = & \mathsf{App}\ (\mathsf{PrimitiveFun\ Ord_less})\ [m, n]
\end{array}$$

and or_list $[x_0, x_1, \ldots, x_n] = x_0 \vee x_1 \vee \ldots \vee x_n$. The free variables are: the conclusion formula f; the measure function m, represented as a single expression with free variables; and the list of induction hypothesis qs, each hypothesis is represented as a pair: a case expression and lists of substitutions to f (each substitution must decrease the measure m).

$$\frac{\begin{array}{l}\vdash_\pi \mathsf{ordp}\ m = \mathtt{T} \\ \vdash_\pi \mathsf{or\_list}\ (f :: \mathsf{map}\ \mathsf{fst}\ qs) \\ (\forall q\ ss.\ \mathsf{mem}\ (q, ss)\ qs \Longrightarrow \\ \qquad \vdash_\pi \mathsf{or\_list}\ (f :: \neg q :: \mathsf{map}\ (\lambda s.\ \neg(\mathsf{formula\_sub}\ s\ f))\ ss)) \\ (\forall q\ ss\ s.\ \mathsf{mem}\ (q, ss)\ qs \wedge \mathsf{mem}\ s\ ss \Longrightarrow \\ \qquad \vdash_\pi \neg q \vee (\mathsf{ord\_less}\ (\mathsf{term\_sub}\ s\ m)\ m = \mathtt{T}))\end{array}}{\vdash_\pi f}$$

The final two inference rules allow function definitions to be looked up from the logical context. The following rule lets us use functions that have a conventional defining equation.

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

Witness functions give rise to judgments that mimic the implication they represent in the semantics (as part of context_ok). The following inference rule can be read as follows: for any instantiation i of the free variables $var :: vs$ which make *body* true, i.e. not equal to `NIL`, the function called *name* returns, when applied to vs instantiated according to i, some value for *var* such that *body* is true, i.e. not equal to `NIL`.

$$\frac{\pi(name) = (vs, \mathsf{Witness}\ body\ var, interp)}{\begin{array}{l}\vdash_\pi body = \mathtt{NIL} \vee \\ \quad \neg(\mathsf{LambdaApp}\ (var :: vs)\ body \\ \qquad (\mathsf{App}\ (\mathsf{Fun}\ name)\ (\mathsf{map\ Var}\ vs) :: \mathsf{map\ Var}\ vs) = \mathtt{NIL})\end{array}}$$

8.5 Soundness and Consistency

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

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

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 [40] had already formalized this result in HOL4. The soundness of the induction rule follows almost directly from their result.

The above soundness theorem lets us immediately prove many reassuring corollaries. For instance, since $\models_\pi$ `T = NIL` is false, we know that this formula cannot be proved using any combination of Milawa's inference rules $\vdash_\pi$. Milawa's inference rules are consistent, since we can also prove that there exists true formulas: e.g. $\vdash_\pi$ `T = T`.

8.6 Soundness Preserved by Function Definitions

As part of our verification of Milawa's kernel (Section 10), we have proved that the kernel maintains an invariant which states that the current logical context π is well-formed,

$$\mathsf{context\_ok}\ \pi \tag{1}$$

and 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:

$$\vdash_\pi p \tag{2}$$

However, when new definitions are made the logical context is extended. In order to maintain our invariant we must thus show that properties (1) and (2) carry across past context extensions.

Proving that property (2) carries over when the context is extended is straighfoward since the syntactic inference rules only make simple tests for inclusion in the context. We can freely assign a meaning to a previously unused names, as the following theorem states: if π' is an extension of π with a new *name*, i.e. $\pi' = \pi[name \mapsto value]$ for some *value*, then:

$$name \notin \mathsf{domain}\ \pi \implies \forall p.\ (\vdash_\pi p) \Longrightarrow (\vdash_{\pi'} p)$$

This theorem is trivial to prove by rule induction over $\vdash_\pi$.

Proving that well-formedness of the context, i.e. property (1), carries across inclusion of new definitions is less straighforward. The main complication is that we need to find an interpretation for the new function such that this interpretation agrees with the syntax of the new definition. In fact, for our overall soundness theorem we only need to know that some such well-formed interpretation exists. Thus we state the theorem for preservation of property (1) to say: if the current context π is well-formed (context_ok) and the new definition satisfies definition_ok, which will be explained below, then there exists an interpretation *interp* for the new definition such that the updated context is well-formed.

$$\begin{array}{l}\forall \pi\ name\ formals\ body.\\ \quad \mathsf{context\_ok}\ \pi \wedge \mathsf{definition\_ok}\ (name, formals, body, \pi) \implies\\ \quad \exists interp.\ \mathsf{context\_ok}\ (\pi[name \mapsto (formals, body, interp)])\end{array} \tag{3}$$

In what follows, we will explain how we proved theorem (3), but first let's take a brief look at the definition of definition_ok. Figure 3 lists its formal definition which states that: the function *name* is unused in the current context; the list of formal parameters does not contain duplicates; the body of the definition is syntactically well-formed; and, in case the new function is recursive, its termination obligations must have been proved in an intermediate context extended with the name of the new function but not its definition. Section 8.6.2 will explain, through an example, the lengthy definition of the function which generates the termination obligations, i.e. termination_obligations.

```
([] ++ ys) = []                          flat [] = []
((x :: xs) ++ ys) = x::(xs ++ ys)        flat (x :: xs) = x ++ flat xs

callmap_sub ss zs =
  map (λ(xs, ys). (map (term_sub ss) xs, map (term_sub ss) ys)) zs

callmap name (Const c) = []
callmap name (Var v) = []
callmap name (App (PrimitiveFun If) [x0, x1, x2]) =
  callmap name x0 ++
  map (λ(x, y). (x, x0::y)) (callmap name x1) ++
  map (λ(x, y). (x, App (PrimitiveFun Not) [x0]::y)) (callmap name x2)
callmap name (App f xs) =
  if f = Fun name then (xs, [])::flat (map (callmap name) xs)
                  else flat (map (callmap name) xs)
callmap name (LambdaApp vs x xs) =
  flat (map (callmap name) xs) ++
       callmap_sub (zip (vs, xs)) (callmap name x))

progress_obligation m formals (xs, ys) =
  or_list ((ord_less (term_sub (zip (formals, xs)) m) m = T)::
           map (λy. (y = NIL)) ys)

termination_obligations name body formals measure =
  if callmap name body = [] then [] else ((ordp measure = T)::
    map (progress_obligation measure formals) (callmap name body))

definition_ok (name, formals, None, π) =
  name ∉ domain π ∧ all_distinct formals
definition_ok (name, formals, Witness body var, π) =
  name ∉ domain π ∧ all_distinct (var :: formals) ∧ term_ok_π body ∧
  list_to_set (free_vars body) ⊆ list_to_set (var :: formals)
definition_ok (name, formals, Body body, π) =
  name ∉ domain π ∧ all_distinct formals ∧
  let π' = π[name ↦ (formals, None, arbitrary)] in
    term_ok_π' body ∧ list_to_set (free_vars body) ⊆ list_to_set formals ∧
    ∃m. ∀p. mem p (termination_obligations name body formals m)
              ⟹  ⊢_π' p
```

Fig. 3 Definition of definition_ok and termination_obligations.

8.6.1 Interpretations for Witness Functions

Per context_ok (Section 8.3), the interpretation for witness functions must be a function *interp* such that, if a value v exists for which a specific property is true, then *interp* returns some value that makes that property true. The property is:

$$\mathsf{term}\ [var :: formals \mapsto v :: args]\ \pi\ prop \neq \mathtt{NIL}$$

In higher-order logic (HOL), one can define a function which returns such a v, if such a v exists, using Hilbert's choice operator ε. Hilbert's choice operator allows us to write $\varepsilon v.\ P\ v$ to denote a value v such that $P\ v$ is true, for a property P (any function which returns a Boolean value, i.e. any instance of type $\alpha \to \mathsf{bool}$). Hilbert's choice operator satisfies the following axiom of HOL:

$$\forall P.\ \ (\exists v.\ P\ v) \Longrightarrow P\ (\varepsilon v.\ P\ v)$$

HOL leaves the other case, i.e. $\neg(\exists v.\ P\ v)$, unspecified, i.e. one can think of $\varepsilon v.\ P\ v$ as returning an arbitrary value of the correct type in case P is not true for any input value.

In the proof that some interpretation function exists (3), we instantiate the existential quantifier for the Witness case with the following function that uses Hilbert's choice operator.

$$\lambda args.\ \varepsilon v.\ \mathsf{term}\ [var :: formals \mapsto v :: args]\ \pi\ prop \neq \texttt{NIL}$$

8.6.2 Interpretations for Recursive Functions

Constructing interpretations for recursive functions is more involved. We will explain our design choice using an example which also illustrates how termination obligations are computed and why we included the None option for function bodies in the logical context (Section 8.2). The example we will use is the definition of the following function called `f`. Note that `f` makes a nested recursive call.

```
(f n k) = (if (< n 1)
              k
              (f (- n 1) (cons k (f (- n 1) nil))))
```

Calculation of the termination obligations first evaluates callmap `"f"` applied to the right-hand side of the definition. This evaluation results in a list containing information about the calls to `f`: both recursive calls can assume `(not (< n 1))`, the first one is called with arguments `(- n 1)` and `(cons k (f (- n 1) nil))`, and the second call with arguments `(- n 1)` and `nil`. Termination obligations are generated based on this information. For the sake of this example, let's assume the user has supplied `(+ n k)` as the measure. With this measure the following three termination obligations are generated. Here we write $(q = \texttt{T}) \vee (p = \texttt{NIL})$ as $p \Longrightarrow q$.

```
(ordp (+ n k)) = T
(not (< n 1)) ⟹ (ord< (+ (- n 1) (... f ...))) (+ n k))
(not (< n 1)) ⟹ (ord< (+ (- n 1) nil) (+ n k))
```

Note that these termination obligations mention the name of the function `f` which is being defined. Thus, the proof of these termination obligations must be performed in a context where the name `f` can appear, but its defining equation is not yet available for use in proofs (because `f` is not yet defined). This awkward intermediate form of the context is provided by the None-alternative for context entries. The syntactic inference rules $\vdash_\pi$ do not provide a way to use the defining equation for None entries. A None-version of the new context is what is used in the proofs of the termination obligations in the definition of definition_ok (Figure 3).

We use the user's proof of the termination obligations to show that some interpretation of the Milawa-defined function always exists. We show, for all Milawa function definitions for which the termination obligations have been proved, that evaluation of the function in a deterministic big-step operational semantics for the logic's term-expressions terminates with a value for the function application. We take the evaluation of this operational semantics to be the interpretation of the Milawa function. We use Hilbert's choice operator, described above, to define this formally in HOL as follows. Here $\Downarrow_{\mathsf{ap}}$ is the big-step operational semantics explained below.

$$\lambda args.\ \varepsilon result.\ (name, \mathsf{Fun}\ name, args, \pi)\ \Downarrow_{\mathsf{ap}}\ result$$

We omit the definition of $\Downarrow_{\mathsf{ap}}$, but note that it is a standard deterministic big-step operational semantics for computing the value of the logic's terms, if they are treated as Lisp functions. The only quirk in this semantics is that function calls to functions with names other than *name* are given special treatment: calls to other functions are evaluated according to the interpretation function in the context π. Evaluation of calls to *name* proceed in the normal manner: the body of the function is expanded for evaluation. We have shown that a proof of the termination obligations always implies that $\Downarrow_{\mathsf{ap}}$ evaluates to some *result*, for any *args* list of the correct length.

9 Specification of the Runtime

In order to prove that Milawa's kernel (Section 5) is faithful Milawa's logic (Section 8), we need a formal specification of the programming language in which Milawa's kernel is written. This section presents a formal specification, in terms of a structured operational semantics, for the Lisp dialect that was outlined informally in Section 5. Later sections describe how we have used this operational semantics to show that Milawa's kernel is faithful to the Milawa logic (Section 10), and prove that the underlying Jitawa Lisp implementation correctly implement this operational semantics (Section 12).

9.1 Syntax

The abstract syntax of Jitawa's Lisp dialect is defined as shown in Figure 4. Note that the core of this language is syntactically the same as the syntax of terms and functions in Milawa's logic, e.g. Const, Var and If match exactly. Indeed the type for S-expressions *sexp* is borrowed from the definition of Milawa's logic (Section 8). However, this datatype adds a few new functions, Define, Print, Error and Funcall, and macros, e.g. Defun which expands into an application of Define.

9.2 Evaluation Semantics

We define a big-step operational semantics as a inductive relation $\xrightarrow{\mathsf{ev}}$ that specifies how objects of type *lterm* evaluate. Following Common Lisp, we separate the store k for functions from the environment *env* for local variables. We model the I/O output stream *io* as a string, i.e. a list of characters produced as output. We also include a error state component called *ok*: if *ok* is false at the end of the computation then an error message interrupted the execution and the result can be ignored. Our evaluation relation $\xrightarrow{\mathsf{ev}}$ explains how some t, of type *lterm*, may be evaluated with respect to some particular k, *env*, *io* and *ok* to produce a resulting *ans*, of type *sexp*, and an updated k', io' and ok'.

As an example, the following rule shows how Var terms are evaluated. We only permit the evaluation of bound variables, i.e. $x \in \mathsf{domain}\ env$. Read the following as saying that if x can be found in the environment then its value is read from the environment.

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

$$
\begin{array}{lcll}
\mathit{lterm} & ::= & \mathsf{Const}\ \mathit{sexp} & \\
& | & \mathsf{Var}\ \mathit{string} & \\
& | & \mathsf{App}\ \mathit{lfunc}\ (\mathit{lterm}\ \mathsf{list}) & \\
& | & \mathsf{If}\ \mathit{lterm}\ \mathit{lterm}\ \mathit{lterm} & \\
& | & \mathsf{LambdaApp}\ (\mathit{string}\ \mathsf{list})\ \mathit{lterm}\ (\mathit{lterm}\ \mathsf{list}) & \\
& | & \mathsf{Or}\ (\mathit{lterm}\ \mathsf{list}) & \\
& | & \mathsf{And}\ (\mathit{lterm}\ \mathsf{list}) & (\text{macro}) \\
& | & \mathsf{List}\ (\mathit{lterm}\ \mathsf{list}) & (\text{macro}) \\
& | & \mathsf{Let}\ ((\mathit{string} \times \mathit{lterm})\ \mathsf{list})\ \mathit{lterm} & (\text{macro}) \\
& | & \mathsf{LetStar}\ ((\mathit{string} \times \mathit{lterm})\ \mathsf{list})\ \mathit{lterm} & (\text{macro}) \\
& | & \mathsf{Cond}\ ((\mathit{lterm} \times \mathit{lterm})\ \mathsf{list}) & (\text{macro}) \\
& | & \mathsf{First}\ \mathit{lterm} \mid \mathsf{Second}\ \mathit{lterm} \mid \mathsf{Third}\ \mathit{lterm} & (\text{macro}) \\
& | & \mathsf{Fourth}\ \mathit{lterm} \mid \mathsf{Fifth}\ \mathit{lterm} & (\text{macro}) \\
& | & \mathsf{Defun}\ \mathit{string}\ (\mathit{string}\ \mathsf{list})\ \mathit{sexp} & (\text{macro}) \\
\\
\mathit{lfunc} & ::= & \mathsf{Define} \mid \mathsf{Print} \mid \mathsf{Error} \mid \mathsf{Funcall} & \\
& | & \mathsf{PrimitiveFun}\ \mathit{lprim} \mid \mathsf{Fun}\ \mathit{string} & \\
\\
\mathit{lprim} & ::= & \mathsf{Equal} \mid \mathsf{Symbolp} \mid \mathsf{SymbolLess} & \\
& | & \mathsf{Consp} \mid \mathsf{Cons} \mid \mathsf{Car} \mid \mathsf{Cdr} \mid & \\
& | & \mathsf{Natp} \mid \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Less} &
\end{array}
$$

Fig. 4 Abstract syntax of Jitawa's Lisp dialect.

Our evaluation relation is defined inductively with auxiliary 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 *lfunc*) is applied: first the arguments are evaluated using $\xrightarrow{\mathsf{evl}}$, then the apply relation $\xrightarrow{\mathsf{ap}}$ determines the result of the application.

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

User-defined functions can be applied, according to $\xrightarrow{\mathsf{ap}}$, whenever a function with the right number of parameters can be found in the function store k under the given *name*.

$$\frac{k(\mathit{name}) = (\mathit{formals}, \mathit{body}) \ \wedge\ (\mathsf{length}\ \mathit{vals} = \mathsf{length}\ \mathit{formals}) \wedge (\mathit{body}, [\mathit{formals} \leftarrow \mathit{vals}], k, \mathit{io}, \mathit{ok}) \xrightarrow{\mathsf{ev}} (\mathit{ans}, k', \mathit{io}', \mathit{ok}')}{(\mathsf{Fun}\ \mathit{name}, \mathit{vals}, \mathit{env}, k, \mathit{io}, \mathit{ok}) \xrightarrow{\mathsf{ap}} (\mathit{ans}, k', \mathit{io}', \mathit{ok}')}$$

Our Lisp implementation, Jitawa, performs dynamic compilation. With regards to dynamic compilation, an interesting case is how user-defined functions are introduced. New definitions can simply be added to the function store k by evaluation of Define. Any attempt at overwriting existing definitions, i.e. $\mathit{name} \in \mathsf{domain}\ k$, causes an error.

$$\frac{\mathit{ok}' = (\mathit{ok} \wedge \mathit{name} \notin \mathsf{domain}\ k) \ \wedge\ k' = k[\mathit{name} \mapsto (\mathit{formals}, \mathit{body})]}{(\mathsf{Define}, [\mathit{name}, \mathit{formals}, \mathit{body}], \mathit{env}, k, \mathit{io}, \mathit{ok}) \xrightarrow{\mathsf{ap}} (\mathsf{nil}, k', \mathit{io}, \mathit{ok}')}$$

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 compilation of function calls within *body* depend 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. An 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`.

Jitawa's input language includes a number of macros And, Let, Cond, First etc. These simply evaluate to their macro expansion. For example, First expands to an application of the Car primitive function.

$$\frac{(\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Car})\ [x], env, k, io, ok) \xrightarrow{\mathsf{ev}} (ans, k', io', ok')}{(\mathsf{First}\ x, env, k, io, ok) \xrightarrow{\mathsf{ev}} (ans, k', io', ok')}$$

Most of Jitawa's macros mimic standard Common Lisp macros. The odd one out is Defun, which is a primitive function in Common Lisp. In Jitawa, Defun is a macro that expands to a quoted call to Jitawa's primitive definition mechanism Define. Using `defun`, we can write the above definitions of `even` and `odd` without the quotes that were used above.

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

The semantics of Defun is an expansion into Define applied to appropriate Const arguments.

Other $\xrightarrow{\mathsf{ev}}$ rules of interest are evaluation of Funcall, Print and Error. Dynamic function calls can be performed using Funcall, which has a semantics that is exactly the same as that of Fun except that it reads the name of the function to be called from the list of arguments.

$$\frac{(\mathsf{Fun}\ name, vals, env, k, io, ok) \xrightarrow{\mathsf{ap}} (ans, k', io', ok')}{(\mathsf{Funcall}, name :: vals, env, k, io, ok) \xrightarrow{\mathsf{ap}} (ans, k', io', ok')}$$

An evaluation of Print appends to the output stream a string representation of the arguments passed to Print:

$$\frac{io' = io +\!\!+ \mathsf{sexp2string}\ (\mathsf{list2sexp}\ (\mathsf{Sym}\ \texttt{"PRINT"} :: vals))}{(\mathsf{Print}, vals, env, k, io, ok) \xrightarrow{\mathsf{ap}} (\mathsf{nil}, k, io', ok)}$$

An evaluation of Error prints its arguments just like Print, except that it also sets the *ok* flag to false to indicate that execution was interrupted.

$$\frac{io' = io +\!\!+ \mathsf{sexp2string}\ (\mathsf{list2sexp}\ (\mathsf{Sym}\ \texttt{"ERROR"} :: vals))}{(\mathsf{Print}, vals, env, k, io, ok) \xrightarrow{\mathsf{ap}} (\mathsf{nil}, k, io', \mathsf{false})}$$

Once the *ok* flag has become false, the *ok* flag will forever be stuck at false and the other state components may change arbitrarily. For example, the following rule states that any function application when *ok* is false can result in any result *ans* as long as *ok* remains false.

$$\frac{k(name) = (formals, body) \ \wedge \ (\mathsf{length}\ vals = \mathsf{length}\ formals)}{(\mathsf{Fun}\ name, vals, env, k, io, \mathsf{false}) \xrightarrow{\mathsf{ap}} (ans, k, io, \mathsf{false})}$$

We omit the rest of the definition of the semantics since the other cases (If, Car, Second etc.) are largely standard and follow the style described above.

9.3 Parsing and Printing

So far, our semantics deals only with abstract syntax. The real implementation must, of course, deal with real ASCII syntax. We model 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 7.4. Also, the parser we verified in previous work [39] 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,

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

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, abbr2string, 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:

$$\forall s\ a\ rest.\ \ \mathsf{next\_sexp}\ (\mathsf{abbr2string}\ ((s, a)::rest))\ =\ (s, \mathsf{abbr2string}\ rest)$$

9.4 Specification of Read-Eval-Print Loop

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.

$$\frac{\mathsf{is\_eof}\ input = (\mathsf{true}, rest)}{(input, k, output) \xrightarrow{\mathsf{exec}} (output, \mathsf{true})}$$

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

$$\frac{\begin{array}{l}\mathsf{is\_eof}\ input = (\mathsf{false}, input') \wedge \\ \mathsf{next\_sexp}\ (input') = (s, rest) \wedge \\ (\mathsf{sexp2term}\ s, [], k, output, \mathsf{true}) \xrightarrow{\mathsf{ev}} (ans, k', output', ok') \wedge \\ output'' = output' +\!\!+ (\mathsf{sexp2string}\ ans) +\!\!+ \texttt{"\textbackslash n"} \wedge \\ (rest, k', output'', \mathsf{true}) \xrightarrow{\mathsf{exec}} (output''', ok'')\end{array}}{(input, k, output) \xrightarrow{\mathsf{exec}} (output''', ok' \wedge ok'')}$$

10 Proving Faithfulness to the Logic

Now that we are equipped with a formal specification of the underlying Jitawa Lisp implementation (Section 9) and a formal model of the Milawa logic (Section 8), we can turn to the problem of proving that the code for Milawa's kernel is faithful to the inference rules of the Milawa logic.

10.1 From ASCII to Shallow Embeddings in HOL4

The most immediate hurdle is the low-level nature of the specification of Jitawa's read-eval-print loop. This specification is stated in terms of parsing ASCII characters from an input stream and then evaluating them w.r.t. an operational semantics defined in terms of inductive relations over a deep embedding. For purposes of verification it would be much more convenient to have Milawa's kernel — a 2,000-line functional program — represented directly as the equivalent logic functions (a shallow embedding) in the logic, i.e. in HOL. In this section, we explain how we have coded up a proof tool [41] which can perform this translation from deep to shallow embeddings automatically and, at the same time, produce a proof that the two representations really compute the same result w.r.t. Jitawa's specification.

Before delving into the details of our proof tool, let's have a closer look at the problem. Below is the first and one of the simplest functions in the Milawa kernel.

```
(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))
```

The top-level Jitawa semantics describes how S-expressions are to be parsed from an input stream of ASCII characters and evaluated. To get the Milawa kernel loaded into Jitawa, the definition of `lookup-safe` and the other kernel functions are read into Jitawa as characters, parsed, and stored as definitions. So to verify Milawa's kernel, we really need to prove a property about how Jitawa interprets this 2,000 line string of Lisp code.

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

$$\mathsf{Defun}\ \texttt{"LOOKUP-SAFE"}\ [\texttt{"A"}, \texttt{"X"}]\ (\mathsf{Dot}\ (\mathsf{Sym}\ \texttt{"IF"})\ (\ldots))$$

When Jitawa then evaluates this Defun expression, a definition for `lookup-safe` is added to its list of functions. The new entry looks roughly like this:

```
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"))
```

Verifying programs directly w.r.t. this deep embedding is hopelessly tedious.

To avoid the manual effort involved, 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 explained in the next section, but the net effect of using it on `lookup-safe` is easy to see. The tool provides us with a simple HOL function,

$$\begin{array}{lcl}
\mathsf{lookup_safe}\ a\ x & = & \mathsf{if\ consp}\ x\ \mathsf{then} \\
& & \quad \mathsf{if}\ a = \mathsf{car}\ (\mathsf{car}\ x)\ \mathsf{then} \\
& & \quad\quad \mathsf{if\ consp}\ (\mathsf{car}\ x)\ \mathsf{then} \\
& & \quad\quad\quad \mathsf{car}\ x \\
& & \quad\quad \mathsf{else\ cons}\ (\mathsf{car}\ (\mathsf{car}\ x))\ (\mathsf{cdr}\ (\mathsf{car}\ x)) \\
& & \quad \mathsf{else\ lookup_safe}\ a\ (\mathsf{cdr}\ x) \\
& & \mathsf{else\ Sym}\ \texttt{"NIL"}
\end{array}$$

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

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

Here *state* is Jitawa's mutable state which has, e.g., the I/O output stream and the list of function definitions. The state is not changed by lookup_safe because `lookup-safe` is a pure function. Extraction of impure functions is also possible, as will be explained in Section 10.2.3.

10.2 Automatic Translation: Deep to Shallow Embeddings

When converting deep embeddings into shallow embeddings our tool's task is to derive a definition of a function, e.g. lookup_safe, from a deep embedding `LOOKUP-SAFE` and prove a connection between the two representations, e.g.

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

The method by which we accomplish this has two phases. The first phase derives a theorem of the following form, for some *hypothesis* and some *expression*. Here *expression* has type *sexp* and *body* has type *lterm*.

$$\mathit{hypothesis} \implies (\mathit{body}, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{expression}, \mathit{state})$$

This derivation proceeds as a bottom-up traversal of the abstract syntax tree for the *body* of the function we are extracting. At each stage a lemma is applied to introduce the relevant syntax in *body* and, at the same time, construct the corresponding shallowly embedded operations over the *sexp* type in *expression*.

The second phase defines a shallow embedding using *expression* as the right-hand side of the definition and discharges (most of) the *hypothesis* using the induction that arises from the termination proof for the shallow embedding.

There is no guess work or heuristics involved in this algorithm, which means that well-written implementations can be robust.

10.2.1 Example: Append Function

An example will illustrate this algorithm. To keep our example clean of unnecessary clutter, consider `APPEND` defined as follows.

```
(defun APPEND (x y)
  (if (consp x)
      (cons (car x) (APPEND (cdr x) y))
      y))
```

For the *first phase*, we aim to derive a theorem describing the effect of evaluating the body of the `APPEND` function, i.e.

$$\begin{array}{l}\mathsf{If}\ (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Consp})\ [\mathsf{Var}\ \texttt{"X"}]) \\ \quad (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Cons})\ [\ldots, \mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [\ldots]]) \\ \quad (\mathsf{Var}\ \texttt{"Y"})\end{array} \tag{4}$$

Our bottom-up traversal starts at the leaves and works its way up the syntax tree. At the leaves, we have variable look-ups and thus instantiate v to `"X"` and `"Y"` in the following lemma to get theorems describing the leaves of the program.

$$v \in \mathsf{domain}\ \mathit{env} \implies (\mathsf{Var}\ v, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{env}\ v, \mathit{state})$$

Now that we have theorems describing the leaves, we can move upwards and instantiate lemmas for primitives, e.g. we instantiate a lemma for the Cdr primitive using Modus Ponens against:

$$\begin{array}{l}(\mathit{hyp} \implies (x, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{exp}, \mathit{state})) \implies \\ (\mathit{hyp} \implies (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Cdr})\ [x], \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{cdr}\ \mathit{exp}, \mathit{state}))\end{array}$$

When we encounter the recursive call to `APPEND` we, of course, do not have a description yet. In this case, we insert a theorem where *hypothesis* makes the assumption that some function variable *append*, of type $\mathit{sexp} \to \mathit{sexp} \to \mathit{sexp}$, describes this application.

$$\begin{array}{l}(\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathit{append}\ x\ y, \mathit{state}) \implies \\ (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathit{append}\ x\ y, \mathit{state})\end{array}$$

The result of the *first phase* is a theorem of the form

$$\mathit{hypothesis} \implies (\mathit{body}, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{expression}, \mathit{state})$$

Here *body* is the abstract syntax tree for the body of `APPEND`; and *expression* is the following, if $\mathit{env} = \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}$,

$$\begin{array}{l}\mathsf{if\ consp}\ x \neq \mathsf{nil}\ \mathsf{then}\\ \quad \mathsf{cons}\ (\mathsf{car}\ x)\ (\mathit{append}\ (\mathsf{cdr}\ x)\ y)\\ \mathsf{else}\ y\end{array} \tag{5}$$

and, with the same *env* instantiation, *hypothesis* is:

$$\begin{array}{l}\mathsf{consp}\ x \neq \mathsf{nil} \implies\\ (\mathsf{Fun}\ \texttt{"APPEND"}, [\mathsf{cdr}\ x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathit{append}\ (\mathsf{cdr}\ x)\ y, \mathit{state})\end{array}$$

Next, we enter the *second phase*: we define append so that its right-hand side is (5) with *append* replaced by the logical constant append. As part of the straightforward termination proof for this definition, we get an induction principle

$$\begin{array}{l}\forall P.\\ \quad (\forall x\ y.\ (\mathsf{consp}\ x \neq \mathsf{nil} \implies P\ (\mathsf{cdr}\ x)\ y) \implies P\ x\ y) \implies\\ \quad (\forall x\ y.\ P\ x\ y)\end{array} \tag{6}$$

which we will use to finalise the proof of the certificate theorem.

For the running example, let P abbreviate the following.

$$\lambda x\ y.\ (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \mathit{state})$$

We now restate the result of phase one using P and the definition of append, and arrive at:

$$\begin{array}{l}\forall x\ y.\ (\mathsf{consp}\ x \neq \mathsf{nil} \implies \mathsf{P}\ (\mathsf{cdr}\ x)\ y) \implies\\ \qquad (\mathit{body}, \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{append}\ x\ y, \mathit{state})\end{array} \tag{7}$$

Let code_for_append_in *state* state that the deep embedding (4) is bound to the name `APPEND` and parameter list `["X", "Y"]` in *state*. Now the operational semantics' rule for function application (Section 9.2) gives us the following lemma.

$$\begin{array}{l}\forall x\ y.\ (\mathit{body}, \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{append}\ x\ y, \mathit{state})\\ \qquad \wedge\, \mathsf{code\_for\_append\_in}\ \mathit{state} \implies \mathsf{P}\ x\ y\end{array} \tag{8}$$

By combining (7) and (8) we can prove:

$$\begin{array}{l}\forall x\ y.\ \ \mathsf{code\_for\_append\_in}\ \mathit{state} \implies\\ \qquad (\mathsf{consp}\ x \neq \mathsf{nil} \implies \mathsf{P}\ (\mathsf{cdr}\ x)\ y) \implies \mathsf{P}\ x\ y\end{array} \tag{9}$$

And a combination of (6) and (9) gives us:

$$\forall x\ y.\ \ \mathsf{code\_for\_append\_in}\ \mathit{state} \implies \mathsf{P}\ x\ y \tag{10}$$

An expansion of the abbreviation P shows that (10) is the certificate theorem we were to derive for `APPEND`: it states that the shallow embedding append is an accurate description of the deep embedding of the `APPEND` function.

$$\begin{array}{l}\forall x\ y\ \mathit{state}.\\ \quad \mathsf{code\_for\_append\_in}\ \mathit{state} \implies\\ \quad (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \mathit{state})\end{array} \tag{11}$$

10.2.2 Example: Reverse Function

Now consider an implementation for `REVERSE` which calls `APPEND`. In the first phase of the translation, the certificate theorem for `APPEND` (11) can be used to give a behaviour to Fun `"APPEND"`. The second phase follows the above proof very closely. The result is the following shallow embedding,

$$\begin{array}{rcl} \mathsf{reverse}\ x & = & \mathsf{if\ consp}\ x \neq \mathsf{nil\ then} \\ & & \quad \mathsf{append}\ (\mathsf{reverse}\ (\mathsf{cdr}\ x))\ (\mathsf{cons}\ (\mathsf{car}\ x)\ \mathsf{nil}) \\ & & \mathsf{else\ nil} \end{array}$$

and a similar certificate theorem:

$$\begin{array}{l} \forall x\ \mathit{state}. \\ \quad \mathsf{code\_for\_reverse\_in}\ \mathit{state} \Longrightarrow \\ \quad (\mathsf{Fun}\ \texttt{"REVERSE"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{reverse}\ x, \mathit{state}) \end{array}$$

Here code_for_reverse_in *state* also requires that code for `APPEND` is present.

10.2.3 More Advanced Language Features

The most advanced feature our Lisp language supports is dynamic function calls using Funcall: the name of the function to be called is the first argument to Funcall (Section 9.2). The equivalent in ML is a call to a function variable. The difference is that Funcall is potentially unsafe, e.g. if called with an invalid function name or with the wrong number of arguments. (ML's type system prevents such unsafe behaviour in ML.) We can support Funcall as follows. First two definitions:

$$\begin{array}{rl} \mathsf{funcall\_ok}\ \mathit{args}\ \mathit{state} = & \exists v.\ (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (v, \mathit{state}) \\ \mathsf{funcall}\ \mathit{args}\ \mathit{state} = & \varepsilon v.\ (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (v, \mathit{state}) \end{array}$$

We use the following lemma in the first phase of the translation algorithm whenever Funcall is encountered.

$$\mathsf{funcall\_ok}\ \mathit{args}\ \mathit{state} \Longrightarrow (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{funcall}\ \mathit{args}\ \mathit{state}, \mathit{state})$$

The result from phase two is a certificate theorem containing a side-condition which collects the hypothesis that the induction is unable to discharge, e.g. if we were translating a function `CALLF` that uses Funcall then we get:

$$\begin{array}{l} \forall x\ \mathit{state}. \\ \quad \mathsf{code\_for\_callf\_in}\ \mathit{state} \wedge \mathsf{callf\_side}\ x\ \mathit{state} \Longrightarrow \\ \quad (\mathsf{Fun}\ \texttt{"CALLF"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{callf}\ x\ \mathit{state}, \mathit{state}) \end{array}$$

So far, we have only considered pure functions, i.e. functions that don't alter *state*. Impure functions are also supported: they translate into shallow embeddings that take the *state* as input and produce a result pair: the return value and the new state. For example, Milawa's main loop handles `def` events using a function called `ADMIT-DEFUN`, which is an inpure function. Extraction of `ADMIT-DEFUN` produces a certificate of the form:

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

The two-phase algorithm outlined above works almost exactly the same way for these impure functions. The only difference is that the *expression* which is accumulated in phase one now covers both the return value and the return state, i.e. *expression* returns a pair.

$$\mathit{hypothesis} \implies (\mathit{body}, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} \mathit{expression}$$

The HOL functions that result from translations of impure functions tend to be less readable than those resulting from pure functions. The reason for this is that the implicit state that is carried around in the impure functions becomes explicit in the resulting shallow embeddings. Use of a state-monad in the extracted functions could possibly bring back some of the readability.

10.3 Verifying Milawa's Proof Checkers and Reflection

Once the entire Milawa kernel had been translated into shallow embeddings as described above, we turned our attention to verifying the faithfulness of the functions in Milawa's kernel.

The largest and most important pure function in Milawa is its initial proof checker, logic_proofp (Section 5.2). This function checks alleged proofs, given an *appeal* (an S-expression encoding of an alleged proof) it returns true or false (in Lisp `T` or `NIL`, respectively). This proof checker walks through the appeal, checking that each proof step is a valid use of some inference rule. Additional inputs are *axioms* and *thms* which are lists of formulas that can be assumed to be theorem in this proof; logic_proofp also takes an arity table *atbl* as input.

When Milawa starts, it uses logic_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 logic_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 logic_proofp would accept. Thus the fidelity of logic_proofp is key to whether Milawa's kernel is faithful to Milawa's logic.

We have proved that logic_proofp is faithful to the inference rules of the Milawa logic (Section 8.4), i.e. we have proved

$$\mathsf{milawa\_faithful}\ \mathsf{logic\_proofp}$$

Here milawa_faithful is defined to say that a *checker*, in this case logic_proofp, is faithful if, whenever it is given syntactically well-formed inputs (appeal_syntax_ok and atbl_ok) and lists *thms* and *axioms* where each element can be derived using Milawa's inference rules (thms_inv), the conclusion of the alleged proof is provable using the syntactic inference rules of the Milawa logic, if *checker* returns true (i.e. returns something non-`NIL` in Lisp).

$$\begin{array}{l}
\mathsf{milawa_faithful}\ \mathit{checker} = \\
\quad \forall \mathit{appeal}\ \mathit{axioms}\ \mathit{thms}\ \mathit{atbl}. \\
\qquad \mathsf{appeal_syntax_ok}\ \mathit{appeal} \wedge \mathsf{atbl_ok}\ \pi\ \mathit{atbl} \wedge \\
\qquad \mathsf{thms_inv}\ \pi\ \mathit{thms} \wedge \mathsf{thms_inv}\ \pi\ \mathit{axioms} \wedge \\
\qquad \mathit{checker}\ (\mathsf{a2sexp}\ \mathit{appeal})\ \mathit{axioms}\ \mathit{thms}\ \mathit{atbl} \neq \mathsf{Sym}\ \texttt{"NIL"} \\
\qquad\quad \implies \vdash_\pi \mathsf{conclusion_of}\ \mathit{appeal}
\end{array}$$

To accommodate the reflective installation of new proof checkers, the invariant we describe in the next section requires that milawa_faithful must always hold for whatever function is the current proof checker. It turns out that Milawa's checks of the fidelity claim, that were explained in Section 5.4, are sufficient to show that a new-proofp may only be installed when it satisfies the milawa_faithful property, i.e., reflection is sound.

10.4 Milawa's Invariant

As it executes, Milawa's kernel carries around state with several lists and mappings that must be kept consistent. Its *explicit* state, state that explicitly carried around as S-expression in the code, 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, e.g. 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, e.g. `error`, `print`, `define`, `funcall`.

There is also *implicit* state maintained within Jitawa's semantics:

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

Finally, for our soundness proof, there is also *logical* state:

- a logical context π must also be maintained.

A key part of our proof is to formalize the invariant that relates these state elements. The full definition including all auxiliary definitions is a few hundred lines long and thus too long to list here. However, for the most part, the invariant simply states the obvious dependencies and relationships between the state components: for example, each entry in the function table (explicit state) must have a corresponding entity inside the runtime (implicit state), and, since this is a reflective theorem prover, each function in the logic (logical state) must have an entry in the function table (explicit state).

Some details are not so 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 specification only stores an abstraction of this syntax (a more abstract datatype). 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 a few primitives that are not part of the logic (e.g., `funcall`, `print`, `error`). To further complicate things, some of the state components can seemingly lag behind: for reasons that will be explained next, 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.

The function table starts off out of sync with the logical context because of the role `logic.proofp` plays in the `switch` events' checks for the *fidelity claim*. The fidelity claim that the user must prove, in the Milawa logic, mentions `logic.proofp`, which is part of Milawa's kernel implementation. There was essentially a design choice here: should the (rather long) definition of `logic.proofp` be part of the axioms in the Milawa logic? Both yes and no are workable design decisions. Choosing to include the definition of `logic.proofp` as an axiom in the logic would have made the kernel's implementation and invariant neater, at the expense of introducing some 90 definitional axioms to the definition of the Milawa logic that tie the logic to this particular implementation of the Milawa kernel. The choice was made *not* to include `logic.proofp` as definitional axioms in the logic and to, instead, use the function table to force the user to introduce a definition of `logic.proofp` in the logical state. In the current setup, the user must run through definitions that exactly match the kernel's internal representation of `logic.proofp`.

We proved each of the event handling functions, `admit-defun`, `admit-switch` etc., maintains an invariant which ensures that all items in the `axioms` and `thms` lists can be derived using the inference rules of Milawa's logic (Section 8.4). As a result, the kernel's top-level event-handler loop also maintains this invariant.

10.5 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. The original version of the kernel [6] did not print out the formulas it has proved, but instead just printed, e.g.,

```
(PRINT (457 VERIFY LEN-OF-REV))
```

to say that it had accepted event number 457, in this case a theorem named `LEN-OF-REV`.

In order to make clearer what Milawa claims to have proved, we added 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 might 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 such a `THEOREM` line 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:

$$\begin{aligned}\mathsf{line\_ok}\ (\pi, l) \quad = \quad & (l = \texttt{"NIL"}) \vee \\ & (\exists n.\ (l = \texttt{"(PRINT (}n\ \ldots\ \texttt{))"}) \wedge \mathsf{is\_number}\ n) \vee \\ & (\exists \phi.\ (l = \texttt{"(PRINT (THEOREM }\phi\texttt{))"}) \wedge \mathsf{context\_ok}\ \pi \wedge \models_\pi \phi)\end{aligned}$$

We then prove that Milawa's top-level function, milawa_main, only produces output lines that satisfy line_ok, assuming that no runtime errors were raised during execution, i.e., that *ok* is true. Here compute_output (defined in Figure 5) is a high-level

```
defun_ctxt π cmd =
  returns π updated with new function definition based on cmd

witness_ctxt π cmd =
  returns π updated with new witness definition based on cmd

print_thm n cmd =
  let φ = car (cdr cmd) in
    "(PRINT (THEOREM φ))"

print_event_number n cmd =
  let x = car cmd in
  let y = car (cdr cmd) in
    "(PRINT (n x y))"

milawa_command π cmd =
  if car cmd = Sym "DEFINE" then (defun_ctxt π cmd, []) else
  if car cmd = Sym "SKOLEM" then (witness_ctxt π cmd, []) else
  if car cmd = Sym "PRINT" then (π, [(π, print_thm cmd)]) else (π, [])

milawa_commands π n cmds =
  if consp cmds then [] else
    let cmd = car cmds in
    let l1 = [(π, print_event_number n cmd)] in
    let (π, l2) = milawa_command π cmd in
      l1 ++ l2 ++ milawa_commands π (n+1) (cdr cmds)

compute_output cmds =
  [([], "NIL"), ([], "NIL"), ([], "NIL"), ([], "NIL"), ([], "NIL")] ++ milawa_commands [] 1 cmds

output_string [] = ""
output_string ((π, line)::xs) = line ++ "\n" ++ output_string xs
```

Fig. 5 Annotated output lines produced by successful Milawa run.

specification of what π-annotated lines a successful execution produces as output.

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

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:

$$
\begin{array}{l}
\forall cmds\ state. \\
\quad \mathsf{code\_for\_milawa\_main\_in}\ state \wedge \mathsf{milawa\_main\_side}\ cmds\ state \Longrightarrow \\
\quad (\mathsf{Fun}\ \texttt{"MILAWA-MAIN"}, [cmds], state) \xrightarrow{\mathsf{ap}} (\mathsf{milawa\_main}\ cmds\ state)
\end{array}
$$

11 The Bottom Layer: the x86 Model

As mentioned earlier, our final target is a proof which reaches down to the concrete x86 machine code which runs our Jitawa Lisp runtime. In order to verify Jitawa down to level of machine code, we required a specification of x86 instruction set architecture (ISA). The ISA specification we use is a 64-bit port of a previously developed [42] model of 32-bit x86.

Our ISA specification covers only a subset of the x86 instruction set architecture. This ISA model includes the following instructions:

```
ADD ADC AND XOR OR SBB SUB CMP TEST INC DEC MOV MOVZX CMPXCHG XADD
XCHG NEG NOT POP PUSH CALL LEA SHL SHR SAR JMP JE JNE JS JNS JA JNA
JB JNB CMOVE CMOVNE CMOVS CMOVNS CMOVA CMOVNA CMOVB CMOVNB CPUID RET
LOOP LOOPE LOOPNE DIV MUL
```

We trust that our ISA specification is reasonably accurate for the small number of instructions it covers. The 32-bit ISA specification from which this specification originates was extensively tested against real x86 hardware. Our 64-bit port has also been tested, but not as extensively.

Our specification of 64-bit x86 is a conventional ISA model providing a next-state function which consists of fetch, decode and execute functions. The state space consists of sixteen 64-bit registers (`RAX`, `RBX`, ..., `R15`), six Boolean status flags (`CF`, `PF`, `AF`, `ZF`, `SF`, `OF`), a flat unsegmented byte-addressable memory and an instruction cache. The instruction cache is made explicit in order to model the hazards that out-of-date instruction caches can cause when just-in-time or dynamic compilation is performed [42].

An example will give a taste for what level of detail this x86-64 specification captures. Below is a theorem which describes the precondition and the update an execution of instruction `add rbx,[rax]` causes. Informally, this instruction loads a 64-bit word from the address held in register `RAX` and adds the value from that memory location to the `RBX` register. Our x86 ISA is a model of concrete machine code and thus concerns the concrete byte encoding of this x86 instruction. The machine code encoding of `add rbx,[rax]` is `480318`, i.e. bytes `0x48`, `0x03` and `0x18`. In order for this instruction to be able to execute, these bytes must appear in memory at the location where the program counter (instruction pointer) `RIP` is pointing, and the 64-bit entity for which register `RBX` holds the address must be accessible. If this precondition is true, then the value of `RAX` is updated, the program counter is bumped along to point at the next instruction (hence +3 below), and the status flags are updated according to the result of the arithmetic operation. Below all arithmetic is over finite 64-bit words, and some x is used to indicate that value x is accurate — as opposed to none which would mean *undefined* or *unknown*.

```
can_read_mem_word64 (read_reg RAX) s ∧
(read_instr (read_rip s) s = some 0x48) ∧
(read_instr (read_rip s+1) s = some 0x03) ∧
(read_instr (read_rip s+2) s = some 0x18)
⟹
(x64_next s = some (write_rip (read_rip s+3)
                   (write_reg RBX (read_reg RAX+read_mem_word64 (read_reg RAX))
                   (write_flag ZF (read_reg RAX+read_mem_word64 (read_reg RAX)=0)
                   (write_flag SF ... CF ... OF ... PF ... s ...))))))
```

Clearly, this x86 model is working at a much lower level of abstraction than Jitawa's high-level specification, which, in contrast, concerns ASCII character streams, abstract datatypes and infinite precision natural numbers.

12 Constructing and Verifying the Runtime

In this section, we describe how we have gone from the Jitawa language specification, Section 9, to an actual x86 implementation that has been proved to implement Jitawa's Lisp language. This part of our work builds on a long line of previous work on machine-code verification [42, 39, 43]. However, the following text will not assume knowledge of our previous work and will, for brevity, omit certain details that the interested reader can find in the papers mentioned above.

12.1 Method

Our implementation of Jitawa consists of 8,200 lines of formally verified x86 machine code. Most of this code was not written and verified by hand. Instead, we produced the implementation and its verification proof using a combination of manual verification, decompilation and proof-producing synthesis. At a high-level, these are the steps we followed:

1. We started by defining a straightforward stack-based bytecode language into which we can easily compile Jitawa Lisp programs using a simple compilation algorithm.
2. Next, we defined a heap invariant, which relates x86 states to abstract Lisp states, and proved that certain machine instruction "snippets" implement basic Lisp operations and maintain the heap invariant.
3. These snippets of verified machine code were then given to our extensible synthesis tool [44] which we used to synthesise verified x86 machine code for our compilation algorithm, i.e. we coded up the compilation algorithm as a functional program from which that the synthesis tool can produce verified x86 using the small verified snippets.
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 the parsing/printing of s-expressions from input/output streams 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.

The next few sections describe each of these steps. Readers that are familiar with our previous work [39] will notice that steps 2 and 3 correspond very closely to how we synthesised verified machine-code for our evaluation function lisp_eval when constructing verified Lisp interpreters.

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 *prim*	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. 6 Abstract syntax of our bytecode.

12.2 Compilation to Bytecode

Jitawa performs dynamic compilation, it compiles all expressions before they are executed. In order to break this compilation into manageable stages, we compile Jitawa terms to an intermediate language, which we will call a bytecode. This bytecode is stack based. Its rather short list of instructions is shown in Figure 6.

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

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

At this level of abstraction, a system state s consists of the following components:

- a code store: a list of bytecode instructions,
- a list of information describing where code for previously compiled functions is stored,
- a list of non-atom constants (for use with LookupConst),
- input and output streams, and
- an *ok* flag indicating whether an error has occurred.

The compile function appends the new code to the end of the code store. At present, compile does not perform any optimizations except for tail-call elimination and a simple optimization that speeds up evaluation of LambdaApp, Let and LetStar.

In what follows, we will give a flavour of the operational semantics of this bytecode. 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 $\xrightarrow{\mathsf{next}}$ relation also updates a program counter pc and the system state s.

The simplest example of $\xrightarrow{\mathsf{next}}$ is the Pop instruction, which just removes an element off the expression stack and advances the program counter to the next

instruction.

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

The instruction for performing a function call, Call, is not much more complicated: Call *pos* changes the program counter to *pos* and push a return address onto the return stack.

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

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.

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

The Print instruction is slightly more exotic: it appends the string representation of the top stack element, given by sexp2string (Section 9.3), onto the output stream, which is part of the system state s. It leaves the stack unchanged.

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

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 according to the compile function.

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

At first sight, it might seem odd that the definition of the operational semantics mentions the compile function. We mention compile because that specifies unambiguously what an implementation of Compile must do.

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

12.3 Jitawa's Heap Invariant

The verification of Jitawa required several invariants at different layers of abstraction. Most of the interesting relationships between low-level and high-level objects is specified in the 'heap invariant' which relates S-expressions and bytecode to values in various memory modelling functions and register contents. This invariant has the form:

$$\mathsf{lisp\_inv}\ \mathit{constants}\ \mathit{abstract}\ \mathit{concrete}$$

Here, *constants* is a tuple of values that parametrize the entire development, e.g. the size of the heap, the location of the bottom of the stack etc. These are universally quantified throughout. At this level, the *abstract* state consists of six S-expression 'registers', an S-expression stack, a stack for return addresses, ASCII I/O streams, and a code store, i.e. a list of abstract bytecode instructions. The *concrete* state consists of various 64-bit values (that will later be in registers) and a few memory mapping functions (one for the heap and expression stack, one for the symbol table and one for the code heap). The memory mapping functions map four-byte-aligned 64-bit addresses to 32-bit values.

The full details of lisp_inv will not be presented here. However, a few key parts will be explained. The lisp_inv invariant states that there exists some heap function h, a mapping from natural numbers to a datatype of heap objects. Heap objects are either a Block, a Ref or a Empty cell. Values are either data (DataFixedNum or DataSymIndex) or an address, HeapAddr. We define a relation lisp_x to specify how S-expression x are to relate to values v w.r.t. a specific heap function h and symbol list *syms*.

$$\mathsf{lisp\_x}\ (h, \mathit{syms})\ (v, x)$$

An S-expression representing a natural number corresponds to a DataFixedNum value. The number must be less than 2^{30}. (Jitawa reports an error if a larger number is produced, e.g. via addition or through parsing.)

$$\mathsf{lisp\_x}\ (h, \mathit{syms})\ (v, \mathsf{Val}\ n) = (v = \mathsf{DataFixedNum}\ n) \wedge n < 2^{30}$$

S-expressions representing symbols correspond to a DataSymIndex value which holds an index into the list of symbols.

$$\begin{aligned}\mathsf{lisp\_x}\ (h, \mathit{syms})\ (v, \mathsf{Sym}\ s) = \exists n.\ &(v = \mathsf{DataSymIndex}\ n) \wedge n < 2^{29} \wedge \\ &(\mathsf{list\_lookup}\ n\ \mathit{syms} = s)\end{aligned}$$

S-expressions that are a Dot pair are represented as an address into the heap. At this address, the heap must contain a Block containing the representations of the two sub-expressions.

$$\begin{aligned}\mathsf{lisp\_x}\ (h, \mathit{syms})\ (v, \mathsf{Dot}\ x\ y) = \exists u\ w\ n.\ &(v = \mathsf{HeapAddr}\ n) \wedge (h\ n = \mathsf{Block}\ (u, w)) \wedge \\ &\mathsf{lisp\_x}\ (h, \mathit{syms})\ (u, x) \wedge \mathsf{lisp\_x}\ (h, \mathit{syms})\ (w, y)\end{aligned}$$

The lisp_inv invariant relates the heap h, values v and symbols list *syms* to the concrete memory mappings and 32-bit values. We represent each heap value v as a 32-bit word that is appropriately tagged so that we can distinguish the types at runtime. Block elements are mapped into the real memory as eight-byte-aligned 64-bit entries, i.e. 32 bits for each component value that they carry. The

expression stack is represented as a simple array of value/*sexp* representations. Runtime checks make sure that the stack pointer always points into this array.

Another important part of the lisp_inv invariant is its specification of how abstract bytecode instructions are represented in memory. We define bc_ref to return the concrete representation of bytecode instructions, given their concrete location l. For example, the representation of the Pop instruction is defined to be represented by the following list of bytes.

$$\mathsf{bc\_ref}\ l\ \mathsf{Pop} \quad = \quad [\texttt{0x44}, \texttt{0x8B}, \texttt{0x4}, \texttt{0x9F}, \texttt{0x48}, \texttt{0xFF}, \texttt{0xC3}]$$

These bytes are encoded of x86 instructions, in this case `mov R8d, [RDI+4*RBX]` followed by `inc RBX`, which implement the bytecode instruction in question, as will be explained in Section 12.6. The lisp_inv invariant states that the list of abstract bytecode instructions is directly mapped into a list of bytes according to bc_ref. These bytes are stored in the mutable code heap.

12.4 Verification of x86 for Basic Lisp Operations

In order to prove that concrete x86 code implements Lisp operations w.r.t. our x86 model from Section 11, we make use of a previously developed Hoare logic for machine code [42,43]. The central concept in this Hoare logic is a judgement which we call a machine-code Hoare triple. These machine-code Hoare triples take the form:

$$\{\mathit{precondition}\}\ \mathit{code}\ \{\mathit{postcondition}\}$$

A few examples will give a feel for what they mean (a full definition is given in [42]). The following Hoare triple describes the `mov RBX, RAX` instruction, which is encoded as three bytes: `0x48`, `0x8B`, `0xD8`. Read the following theorem as saying that, if the program counter `PC` has a value p which points at this code, then the value of `RAX`, i.e. *rax*, is moved into register `RBX`, i.e. replaces its original value *rbx*. The operation also updates the program counter `PC` to point to the next x86 instruction, i.e. p+3 is stored into `PC`. Here $*$ is a separating conjunction from separation logic (read it informally as normal conjunction) and all arithmetic is over 64-bit words.

$$\begin{array}{l}\{\,\mathsf{rax}\ \mathit{rax} * \mathsf{rbx}\ \mathit{rbx} * \mathsf{pc}\ p\,\}\\ \ p : \texttt{488BD8}\\ \{\,\mathsf{rax}\ \mathit{rax} * \mathsf{rbx}\ \mathit{rax} * \mathsf{pc}\ (p{+}3)\,\}\end{array}$$

These Hoare triples concern total-correctness, i.e. they state that execution of the code terminates in a state where the postcondition is true whenever execution started from a state satisfying the precondition. These machine-code Hoare triples differ from conventional Hoare triples in that (i) the program counter is made explicit in the pre- and postconditions, (ii) execution does not need to start at the beginning of the code and exit at the end, and (iii) the executed code does not need to live in the middle of the Hoare triple, as explained in [42].

To prove that certain snippets of x86 code implement Lisp operations we define an invariant lisp which combines lisp_inv from above with the x86-specific resource

assertions rax, rbx etc.

$$\begin{array}{l}\mathsf{lisp}\ \textit{constants}\ (x_0, x_1, x_2, \ldots) = \\ \quad \exists w_0\ w_1\ w_2\ \ldots\ . \\ \qquad \mathsf{rax}\ w_0 * \mathsf{rbx}\ w_1 * \mathsf{rcx}\ w_2 * \ldots \\ \qquad \langle \mathsf{lisp\_inv}\ \textit{constants}\ (x_0, x_1, x_2, \ldots)\ (w_0, w_1, w_2, \ldots)\rangle\end{array}$$

Equipped with this definition of lisp, we can state theorems about machine code in terms of operations over *sexp*. The following theorem makes the above x86 `mov` instruction seem to be an operation over two S-expression registers: x_0 and x_1. In what follows, we abbreviate lisp *constants* by lisp.

$$\begin{array}{l}\{\,\mathsf{lisp}\ (x_0, x_1, x_2, \ldots) * \mathsf{pc}\ p\,\} \\ \ p : \texttt{488BD8} \\ \{\,\mathsf{lisp}\ (x_0, x_0, x_2, \ldots) * \mathsf{pc}\ (p{+}3)\,\}\end{array}$$

Proving such theorems for basic Lisp operations (e.g. moving, adding, subtracting, comparing, pushing, popping S-expressions) is a simple exercise in using the machine-code Hoare logic.

12.5 Synthesis of Code for Complex Lisp Functions

Once we had verified that certain snippets of x86 implement a number of basic Lisp operations, we used the result of this verification to teach a previously developed code synthesis tool [44] about Lisp. This synthesis tool takes the machine-code Hoare triple theorems as input and learns from them how to synthesise certain assignments and comparisons. For example, from the machine-code Hoare triple shown above, it would know how to synthesise

$$\mathsf{let}\ x_1 = x_0\ \mathsf{in}$$

into the x86 instruction `C3C3`. Given enough Lisp primitives, this synthesis tool can compile HOL functions that operate over the *sexp* type directly into x86 machine code.

For each run, the synthesis tool produces a certificate of correctness relating the generated code to the given function. An example will illustrate the use of this tool. Given the following function definition,

$$\begin{array}{l}\mathsf{list\_member}\ x_0\ x_1 = \\ \quad \mathsf{if}\ \neg\mathsf{consp}\ x_1\ \mathsf{then} \\ \qquad \mathsf{let}\ x_2 = \mathsf{Sym}\ \texttt{"NIL"}\ \mathsf{in}\ (x_0, x_1, x_2) \\ \quad \mathsf{else\ if}\ x_0 = \mathsf{car}\ x_1\ \mathsf{then} \\ \qquad \mathsf{let}\ x_2 = \mathsf{Sym}\ \texttt{"T"}\ \mathsf{in}\ (x_0, x_1, x_2) \\ \quad \mathsf{else} \\ \qquad \mathsf{let}\ x_1 = \mathsf{cdr}\ x_1\ \mathsf{in}\ \mathsf{list\_member}\ x_0\ x_1\end{array}$$

the synthesis tool generates the following x86 code,

```
49 F7 C1 01 00 00 00 48 74 09 41 BA 03 00 00 00 48 EB 18 46 8B 14
8E 4D 39 D0 48 74 08 46 8B 4C 8E 04 48 EB DB 41 BA 0B 00 00 00
```

for which it also proves a theorem stating that this x86 code snippet implements exactly list_member. Here 43 is the length of the code and list_member_pre x_0 x_1 is a termination condition.

$$
\begin{array}{l}
\mathsf{list\_member\_pre}\ x_0\ x_1 \implies \\
\{\,\mathsf{lisp}\ (x_0, x_1, x_2, x_3, \ldots) * \mathsf{pc}\ p\,\} \\
\ \ p : \texttt{49F7C101}\ \ldots \\
\{\,\mathsf{let}\ (x_0, x_1, x_2) = \mathsf{list\_member}\ x_0\ x_1\ \mathsf{in} \\
\quad \mathsf{lisp}\ (x_0, x_1, x_2, x_3, \ldots) * \mathsf{pc}\ (p{+}43)\,\}
\end{array}
$$

The tool constructs such theorems and code by composing the verified snippets of x86 that it has been given. It applies a loop rule, described in [43], to tie together loops. We used this synthesis tool to generate correct x86 code from HOL functions for parsing and printing of S-expressions and for compiling S-expressions into bytecode.

12.6 From Bytecode to Machine Code

Great care was taken when defining the concrete representation of the bytecode our compiler produces. The concrete representation of the bytecode was chosen to be strings of bytes that are machine code in themselves. This machine code is, in each case, an implementation of the bytecode instruction it represents. This means that it is sufficient to perform an x86 jump to the representation of a bytecode instruction sequence in order to execute that instruction sequence in x86.

In order to prove a Hoare triple stating that the bytecode representation is indeed correct, we define an assertion lisp_bytecode which specifies how the state of the bytecode semantics maps into the lisp assertion from above. Here xs is the expression stack; hd, tl and ++ are list-head, -tail and -append, respectively, and n2w converts a natural number into a 64-bit word. Note that head of the bytecode's expression stack is kept in the x_0 'S-expression register' and the tail of the bytecode's expression stack in the regular S-expression stack.

$$
\begin{array}{l}
\mathsf{lisp\_bytecode}\ (xs, rs, pc, s) = \\
\quad \exists x_1\ x_2\ x_3\ x_4. \\
\qquad \mathsf{lisp}\ (\mathsf{hd}\ (xs\ +\!\!+\ [\mathsf{nil}]), x_1, x_2, x_3, x_4, \ldots, \\
\qquad\quad \mathsf{tl}\ (xs\ +\!\!+\ [\mathsf{nil}]), \ldots) * \mathsf{pc}\ (\mathsf{n2w}\ pc)
\end{array}
$$

With this state assertion we can state and prove that the concrete representation of, e.g., Pop, i.e. bc_ref Pop, removes the top element from the stack.

$$
\begin{array}{l}
\{\,\mathsf{lisp\_bytecode}\ (top :: xs, rs, pc, s)\,\} \\
\ \ \mathsf{n2w}\ pc : \mathsf{bc\_ref\ Pop} \\
\{\,\mathsf{lisp\_bytecode}\ (xs, rs, pc + \mathsf{length\ Pop}, s)\,\}
\end{array}
$$

We proved similar theorems for each of the bytecode instructions and then from that that any string of bytecode instructions is executed by a jump to their concrete representation.

12.7 I/O

So far, we have glossed over how the machine code implementation deals with I/O. Our Jitawa implementation calls upon the external C routines `fgets` and `fputs` to carry out I/O. These external calls require assumptions in our proof. For instance, for reading characters 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:

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

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.

12.8 Jitawa is Correct

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 $\xrightarrow{\mathsf{exec}}$ is reached or an error message is produced.

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

This specification allows us to resort to an error message even if the evaluated s-expressions would have a meaning in terms of the $\xrightarrow{\mathsf{exec}}$ 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.

13 Top-Level Soundness Theorem

Now that we have a theorem from Section 10 saying that the Milawa theorem is faithful to the Milawa logic, and a theorem from Section 12 saying that Jitawa correctly executes Milawa's kernel, we can combine these into a single top-level theorem proved in HOL. This top-level theorem states that when Milawa is run on top of our verified runtime Jitawa, it can only produce output with lines that show formulas that are true. These formulas are true w.r.t. the semantics of the Milawa logic (see definition of line_ok, Section 10.5).

The top-level theorem, shown below, can informally be read as follows. Suppose the input to Jitawa is the code for Milawa's kernel followed by a call to its main function on any *input*. Such an invocation of Jitawa causes it to either abort with an error message, or succeed and print some line_ok output followed by `SUCCESS`. Here strings are lists of characters, hence the use of list append (++) on strings.

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

This theorem is a total-correctness theorem, in the sense that it guarantees that Milawa will terminate, for all inputs, when run on Jitawa. However, it is partial in that any execution is allowed to exit with an error message. Error messages are caused by e.g. execution of the `error` function inside Jitawa, running out of heap space or arithmetic overflow.

Note that the theorem above allows a trivial implementation of Jitawa which simple exits immediately regardless of input. However, we can show by running Milawa that it is not a trivial exit-always implementation. Jitawa can run through Milawa's entire bootstrapping process without hitting an error message. This shows both that Jitawa is not a trivial implementation and that each proof performed during Milawa's bootstrapping process is indeed true w.r.t. our semantics of the Milawa logic.

13.1 Two Minor Bugs Found

It is worth noting that no soundness bugs were found during our proof. However, to our surprise, 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 as axioms. 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 to host Milawa.

Both of these bugs can be traced back to the original Milawa kernel and Davis' dissertation [6], where the bugs even appear in the main text (pages 125–126 and 140, respectively). The existence of these bugs took us by surprise, since the Milawa kernel was intentionally written to be simple, clean and obviously correct;

and many careful people combed through the kernel's code and the text of Davis' dissertation. The existence of these bugs simply confirms that formal verification does reveal bugs better than careful code reviews.

13.2 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. There were two immediate candidates.

Evaluation through reflection. In the original version of Milawa, reflection was only used to execute new proof checkers. We can also, of course, use the reflected functions for efficient evaluation of arbitrary constant terms. We have implemented this as a new event `admit-eval` that provides a mechanism by which user-defined functions can be evaluated in the runtime, given constant arguments. The result of this evaluation is stored as a theorem, in the form of an equation, $\vdash_\pi$ *function_name* 'x_1 'x_2 ... 'x_n = '*ans*.

Support for partial functions. The fact that our semantics does not explicitly require that functions terminate suggests it may be possible to support partial functions in the Milawa logic. Indeed, it was easy to prove, with the current semantics, that definitions of tail-recursive functions may be soundly admitted as theorems without termination proofs. We proved that a variant of definition_ok, where the termination obligations have been replaced by a syntax check for tail-recursion, is a sufficient condition to that the new context is well-formed. Our approach to partial functions seem to have been simpler than the method described by Manolios and Moore [45] on adding support for partial functions to ACL2. Note that we only proved that it is sound to admit tail-recursive functions without termination proofs. We did not add support for partial functions to Milawa's kernel because doing so would have meant that Milawa's kernel might not terminate for all inputs. To keep our top-level theorem clean of clutter, we wanted Milawa to terminate for all inputs.

14 Discussion of Related Work

Davis' dissertation [6] describes how the Milawa theorem prover is constructed by bootstrapping from a small trusted kernel. In this paper we have described the Milawa theorem prover, Milawa's trust story and how we have verified in HOL4 that Milawa 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 runtime. This theorem goes from the logic all the way down to the machine code, and we believe it is the strongest evidence to date of a theorem prover's soundness.

Harrison's proofs [9] of soundness and consistency for the HOL Light theorem prover are the most closely related work. He proved two theorems to demonstrate the soundness of higher-order logic (HOL): he used a version of HOL with an extra axiom to prove ordinary HOL consistent, and separately used ordinary HOL to prove the consistency of HOL with one axiom removed. Our soundness proof can be more direct, simply because HOL is a more expressive logic than Milawa's

logic: we define the semantics of Milawa in HOL (which is also the logic HOL4 implements) and proved soundness and consistency directly in HOL.

Harrison developed a shallow embedding of HOL Light's 400-line OCaml kernel (except for its definitional mechanism), and proved soundness w.r.t. this implementation. Our work goes further by proving a connection to an operational semantics for our implementation language, and through that a connection to a verified implementation of the language runtime.

Recently, Kumar et al. [46,47] have extended Harrison's original formalisation of HOL light, added definition mechanisms to the logical inferences, and produced a verified CakeML [48] implementation of the kernel functions. This ongoing project aims to prove an end-to-end soundness theorem for the HOL light theorem prover, which is similar to our soundness theorem for Milawa. At the time of writing, Kumar et al. have yet to prove that the soundness of the kernel implies the soundness of the entire theorem prover.

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. [49,50]. 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 simpler, e.g. Milawa's S-expressions do not contain complex rationals, characters or strings, which clutter proofs. On the other hand, we pushed our modeling and proofs further: we proved the soundness of Milawa's inference rules and its implementation.

Our work concerns the verification of an idealized ACL2-like theorem prover. Many other types of proof tools have been verified. As a few examples, McCune and Shumsky [51] proved the soundness of Ivy, a first-order logic proof checker that checks proofs from the external, high-performance provers Otter and MACE. Ridge and Margetson proved soundness and completeness of a simple first-order tableau prover [52] that can be executed in Isabelle/HOL by rewriting. Marić verified a SAT solver [53] with many modern optimizations. Marić suggests that his SAT solver is to be used as an automatically Isabelle/HOL-code-generated implementation. The current Isabelle/HOL code generator [54] is an easy route to reasonably fast implementations, but it fails to carry over the verification proof to the implementation, i.e. there is no Isabelle/HOL theorem by which Marić could connect the implementation to the properties he proved of his SAT solver. Marić could potentially have carried over the verification result using the code synthesis techniques that we have developed [44,41].

15 Conclusions

This paper has presented the Milawa theorem prover and proofs that establish its soundness down to the concrete x86 machine code that runs it. We have used HOL4 to formally specify the Milawa logic, to prove this logic is sound, to prove that Milawa's kernel is faithful to the Milawa logic, and to prove that our Lisp implementation, Jitawa, correctly executes Milawa's kernel. We have connected these results in a top-level HOL4 theorem which, we believe, constitutes the most comprehensive evidence of a theorem prover's soundness to date.

As a consequence of this result, we can trust the Milawa theorem prover even though it directly uses high-level proof automation without generating de Bruijn-style evidence. But how far can we trust it? Could there still be bugs? We must admit that a thoroughgoing skeptic still has several avenues of attack:

1. We trust our top-level theorem because it has been mechanically checked by HOL4. HOL4 is an LCF-style prover that implements a well-known logic, features a small kernel, and is generally regarded as a trustworthy system. However, it is possible that there could be bugs in HOL4, or, more likely, in its ML runtime or in supporting software, e.g., the C compiler that built the runtime or the operating system, or with the hardware we used. Perhaps we exploited such a bug to carry out our verification?
2. Our top-level soundness theorem makes a claim about how our program will behave when it is executed on a HOL4 model of X86. We wrote this model carefully and have informally tested it, but we may have made a mistake. For that matter, real hardware may have bugs: ensuring that a physical computer properly implements its ISA is an inexact and very difficult problem. Perhaps a real computer will not execute Milawa as our model assumes?
3. Our program relies on the operating system to protect it from other programs and to carry out basic I/O operations (Section 7.4). This is a large code base that almost surely has bugs. Perhaps the OS cannot be trusted, or perhaps our I/O operations have not been modeled or implemented correctly?
4. Our top-level soundness theorem uses HOL4 definitions to formalize concepts such as the semantics of the logic. Perhaps there is an error in a definition like $\models_\pi$ that means our theorem does not say quite what we intend?

However, such a skeptic will also have plentiful objections to, say, a theorem prover that generates an explicit de Bruijn justification for some "simple" program to check. After all, any such checking program will at least be based on some understanding of the logic, on software stacks including at least an operating system for basic I/O and protection from other programs, on (in practice) some runtime system or compiler toolchain, on some belief about how the hardware is supposed to operate, and on the hardware itself.

Altogether it took around 4 man-years of programming and proof effort to develop and verify Milawa and Jitawa. Many aspects of Milawa's design have helped to make its verification possible. Its untyped, computational logic provides a very direct connection to Lisp evaluation, allows for a simple reflection principle, and is weak enough for its soundness to be proven directly in HOL. The simplicity of its programming language helps to avoid complexity in the Jitawa runtime, reducing the effort needed for its development and verification. Its kernel is small and straightforward enough to practically process in HOL4.

Much of this work, especially in the areas of machine-code verification, is likely to be reusable in future proof efforts. For instance, tools we developed are already being used in the CakeML [48] project. However, it would likely be considerably more difficult to carry out a similar verification project for any mainstream theorem prover. For instance, ACL2 exposes a much deeper connection to its Common Lisp runtime and has a much larger kernel that bears little relation to its logic. Meanwhile, LCF style systems like HOL4 and HOL Light implement much more powerful logics and include elaborate ML runtimes. It will be exciting to see if the

CakeML project will be able to overcome these challenges to produce a verified HOL light theorem prover.

For more information on Milawa and Jitawa, e.g. further reading, the actual proof scripts, or to download and run the verified code, see:

`http://www.cs.utexas.edu/~jared/milawa/Web/`
`http://www.cl.cam.ac.uk/~mom22/jitawa/`

Acknowledgements We thank Ruben Gamboa and Gerwin Klein for encouraging us to write this paper. We thank the anonymous JAR reviewers and Ramana Kumar for their very helpful feedback on drafts of this paper.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580
2. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (June 2000)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer-Verlag (2004)
4. Slind, K., Norrish, M.: A brief overview of HOL4. In Mohamed, O.A., Muñoz, C., Tahar, S., eds.: TPHOLs. LNCS, Springer (2008)
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
6. Davis, J.C.: A Self-Verifying Theorem Prover. PhD thesis, University of Texas at Austin (December 2009)
7. Boyer, R.S., Kaufmann, M., Moore, J.S.: The Boyer-Moore theorem prover and its interactive enhancement. Computers and Mathematics with Applications **29**(2) (1995) 27–62
8. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS. Springer-Verlag (1979)
9. Harrison, J.: HOL Light: An overview. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: TPHOLs. LNCS, Springer (2009)
10. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: Interactive Theorem Proving (ITP). LNCS, Springer (2011)
11. Harrison, J.: Towards self-verification of HOL Light. In Furbach, U., Shankar, N., eds.: IJCAR. LNAI, Springer (2006)
12. Griffioen, D., Huisman, M.: A comparison of PVS and Isabelle/HOL. In Gundy, J., Newey, M., eds.: Theorem Proving in Higher Order Logics (TPHOLS '98). Volume 1479 of LNCS., Springer-Verlag (September 1998) 123–142
13. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: SMT '09, ACM (2009) 1–5
14. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing. SAT '10, Springer (2010) 44–57
15. Järvisalo, M., Heule, M.J., Biere, A.: Inprocessing rules. In Gramlich, B., Miller, D., Sattler, U., eds.: Automated Reasoning. Volume 7364 of LNCS. Springer (2012) 355–370
16. Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Philosophical Transactions of the Royal Society A **363**(1835) (October 2005) 2351–2375
17. Wetzler, N., Heule, M., Hunt, Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: SAT '14. Volume 8561 of LNCS., Springer 422–429
18. Balabanov, V., Jiang, J.R.: Unified qbf certification and its applications. Formal Methods in System Design **41**(1) (2012) 45–65
19. Böhme, S., Fox, A., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: CPP '11. Volume 7086 of LNCS., Springer (December 2011) 183–198
20. McCune, W., Shumsky, O.: Ivy: A preprocessor and proof checker for first-order logic. In: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)

21. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: ICTAC '10. Volume 6255 of LNCS., Springer (2010) 260–274
22. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic **7**(1) (March 2009) 26–40
23. Marić, F.: Formalization and implementation of modern SAT solvers. Journal of Automated Reasoning **43**(1) (June 2009) 81–119
24. Hurd, J.: The OpenTheory standard theory library. In Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R., eds.: NASA Formal Methods. LNCS, Springer (2011)
25. Kaufmann, M., Moore, J.S.: Structured theory development for a mechanized logic. Journal of Automated Reasoning **26**(2) (2001) 161–203
26. Davis, J.: Reasoning about file input in ACL2. In Manolios, P., Wilding, M., eds.: ACL2 '06. (August 2006)
27. Kaufmann, M., Moore, J.: Design goals of ACL2. Technical Report 101, Computational Logic, Inc. (1994)
28. Rager, D.L., Hunt, Jr., W.A.: Implementing a parallelism library for a functional subset of LISP. In: International Lisp Conference (ILC). (March 2009) 18–30
29. Boyer, R.S., Hunt, Jr., W.A.: Function memoization and unique object representation for ACL2 functions. In: ACL2 '06, ACM (2006)
30. Hunt, Jr., W.A., Krug, R.B., Moore, J.: Linear and nonlinear arithmetic in ACL2. In Geist, D., ed.: Correct Hardware Design and Verification Methods (CHARME '03). Volume 2860 of LNCS., Springer-Verlag (2003) 319–333
31. Hunt, Jr., W.A., Kaufmann, M., Krug, R.B., Moore, J., Smith, E.W.: Meta reasoning in ACL2. In Hurd, J., Melham, T., eds.: Theorem Proving in Higher Order Logics (TPHOLS '05). Volume 3603 of LNCS., Springer Berlin (2005) 163–178
32. Brock, B., Kaufmann, M., Moore, J.S.: Rewriting with equivalence relations in ACL2. Journal of Automated Reasoning **40**(4) (May 2008) 293–306
33. Kaufmann, M., Moore, J.S., Ray, S., Reeber, E.: Integrating external deduction tools with acl2. Journal of Applied Logic **7**(1) (March 2009) 3–25
34. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995)
35. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part 1. Communications of the ACM **3**(4) (April 1960) 184–195
36. Shoenfield, J.R.: Mathematical Logic. The Association for Symbolic Logic (1967)
37. Shankar, N.: Metamathematics, Machines, and Gödel's Proof. Cambridge University Press (1994)
38. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. second edn. Academic Press (1997)
39. Myreen, M.O., Gordon, M.J.C.: Verified LISP implementations on ARM, x86 and PowerPC. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: TPHOLs. LNCS, Springer (2009)
40. Kaufmann, M., Slind, K.: Proof pearl: Wellfounded induction on the ordinals up to ϵ_0. In Schneider, K., Brandt, J., eds.: Theorem Proving in Higher Order Logics (TPHOLs). LNCS, Springer (2007) 294–301
41. Myreen, M.O.: Functional programs: conversions between deep and shallow embeddings. In: Interactive Theorem Proving (ITP). LNCS, Springer (2012)
42. Myreen, M.O.: Verified just-in-time compiler on x86. In Hermenegildo, M.V., Palsberg, J., eds.: Principles of Programming Languages (POPL), ACM (2010)
43. Myreen, M.O.: Formal verification of machine-code programs. PhD thesis, University of Cambridge (2009)
44. Myreen, M.O., Slind, K., Gordon, M.J.: Extensible proof-producing compilation. In de Moor, O., Schwartzbach, M.I., eds.: Compiler Construction (CC). LNCS, Springer (2009)
45. Manolios, P., Moore, J.S.: Partial functions in ACL2. J. Autom. Reasoning **31**(2) (2003) 107–127
46. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: HOL with definitions: Semantics, soundness, and a verified implementation. In Klein, G., Gamboa, R., eds.: Interactive Theorem Proving (ITP). LNCS, Springer (2014)
47. Myreen, M.O., Owens, S., Kumar, R.: Steps towards verified implementations of HOL light. In Blazy, S., Paulin-Mohring, C., Pichardie, D., eds.: Interactive Theorem Proving (ITP). LNCS, Springer (2013)

48. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In Jagannathan, S., Sewell, P., eds.: Principles of Programming Languages (POPL), ACM (2014)
49. Gordon, M.J.C., Hunt, Jr., W.A., Kaufmann, M., Reynolds, J.: An embedding of the ACL2 logic in HOL. In: International Workshop on the ACL2 Theorem Prover and its Applications (ACL2), ACM (2006) 40–46
50. Gordon, M.J.C., Reynolds, J., Hunt, Jr., W.A., Kaufmann, M.: An integration of HOL and ACL2. In: Formal Methods in Computer-Aided Design (FMCAD), IEEE Computer Society (2006) 153–160
51. McCune, W., Shumsky, O.: System description: Ivy. In: Automated Deduction (CADE). LNCS, Springer (2000) 401–405
52. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In Hurd, J., Melham, T.F., eds.: TPHOLs. LNCS, Springer (2005)
53. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50) (2010) 4333–4356
54. Haftmann, F., Bulwahn, L.: Code generation from Isabelle/HOL theories Isabelle2011-1 Documentation, `http://isabelle.in.tum.de`.