A Trustworthy, Extensible Theorem Prover
Ph.D. Dissertation Proposal

Jared Davis
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233, USA
jared@cs.utexas.edu

October 22, 2007
Contents

1 Introduction 1

2 Sketch of our proposal 3
  2.1 Formal verification 3
  2.2 Our choice of logic 4
  2.3 Computers checking proofs 5
  2.4 Our proof checker 7
  2.5 Proof checker extensions 7
  2.6 Proposed extensions 8
  2.7 Using extensions 10

3 Present and remaining work 11
  3.1 Sketch of our logic 11
  3.2 Our proof checker 16
  3.3 Building Proofp-checkable proofs 18
  3.4 Extending Proofp with a new rule 18
  3.5 Proving the new rule is sound 20
  3.6 Remaining work 23

4 Related work 24
  4.1 Current theorem provers 24
  4.2 Embedded proof checkers 28
  4.3 Independent proof checking 29
  4.4 Meta reasoning 30
1 Introduction

Programs have precise semantics, so we can use mathematical proof to establish their properties. These proofs are often too large to validate with the usual “social process” of mathematics, so instead we develop and check them with theorem-proving software. This software must be sophisticated enough to make the proof process tractible, but this very sophistication casts doubt upon the whole enterprise: who verifies the verifier?

In this thesis, we propose developing a useful, mechanically-verified theorem prover. Our program will satisfy two often-conflicting goals:

- **Trust.** Our prover will be based on a well-understood logic and should only accept theorems. The soundness-critical code will be easy to identify and short enough for a good programmer or mathematician to review in a few hours.

- **Capability.** Our system will provide tools to help the user construct proofs. For example, it will permit the user to set up lemmas that can be automatically reused to make progress in new proof attempts.

There are many ways to approach these goals. Ours is to prove our program is sound, i.e., if it claims a formula is a theorem, then it is a theorem.

Of course, we cannot meaningfully use our program to prove its own soundness, since this would be like asking someone if they ever lie. Instead, we imagine two programs, \(A\) and \(B\). \(A\) is a proof checker that only accepts proofs composed of the most primitive steps, like instantiation and cut; \(A\) is so simple the social process of mathematics can establish both its soundness and the consistency of the logical theory it implements (so we know theorems are “always true”). Meanwhile, \(B\) is the practically-useful, automated theorem prover we are proposing to verify. In this thesis, we will construct an \(A\)-style proof that shows \(B\) is sound, and check this proof with \(A\). Then, since we trust \(A\), and since \(A\) says \(B\) is sound, we can also trust \(B\).

Our first task is to write the proof checker, \(A\). Which logic should \(A\) implement? We will use a computational, quantifier-free, first-order logic of total, recursive functions with induction, modeled after the ACL2 logic. Our logic, like any other, puts forth a syntactic definition of proof, so writing a proof checker just means translating this definition into a program. This is straightforward and could be done in any reasonable programming language, so which language should we use? Our logic, like ACL2’s, is compatible with
Common Lisp, so we can treat the functions we define as Lisp programs. And there is good reason to write $A$ as a program in our logic. Since our goal is to use $A$ to prove the soundness of $B$ ("if $B$ accepts $\phi$, then $\phi$ is provable"), we need a way to express provability in our logic. Writing $A$ in our logic lets us do this quite easily, i.e., we can say "$\phi$ is provable when there is a proof of $\phi$ that $A$ accepts."

We also need to develop the theorem prover, $B$. Like $A$, we write $B$ as a program in our logic so we can reason about its definition. $B$ will be far more sophisticated than $A$, e.g., it will include a rule-driven simplifier which can employ calculation. But this means the proof of $B$’s soundness will be a deep result, which is concerning since $A$-style proofs are tedious to write and excessively large.

How can we construct this proof? Our approach is first to use ACL2, a mature and capable theorem prover, to “sketch out” the proof (normally ACL2 is thought of as a formal and trusted tool, but here we are only using it in an informal capacity.) Using ACL2 in this way allows us to plan our proof without needing to confront, simultaneously, the problem of building $A$-style proofs. After we have a solid idea of how the proof should go (and are reasonably convinced it is, in fact, a theorem), we can begin working on translating our sketch into an $A$-style proof.

How big will this $A$-style proof be? Will it be practical to use $A$ to check it? Here our approach is to layer the verification of $B$. That is, instead of going directly from $A$ to $B$, we will use $A$ to verify $A'$, a slightly richer proof checker, then use $A'$ to verify $A''$, etc., until we get to $B$. Each successive proof checker accepts a new kind of proof step that is not available in $A$, e.g., perhaps $A'$ adds a tautology checker so it can prove any tautology in one step, whereas it might take hundreds or thousands of steps to prove some tautology with $A$. Once each $A^i$ has been verified, we can trust it as much as we trust $A$, and we can make use of its new capabilities as we set out to verify $A^{i+1}$.

Successfully verifying $B$ will make the following contributions:

- **A new tool.** Our final program will be suitable in domains from circuit analysis to program verification, and will convey greater confidence than tools with larger, less deliberately-defined cores.

- **Metatheory as a prover design.** We will counter the size of formal proofs by adding new proof methods to raise our level of abstraction, while verifying these methods to ensure soundness.
• Extensible proof methods. Users may develop new, custom proof methods for their domains either as tactic-style programs for $B$, or as new provers ($B', B'', \ldots$) which can be verified with $B$. The work we have done to verify $B$ will provide useful lemmas for verifying these new provers.

• Efficient proof construction. We will formally verify several extended proof methods. By having shown these algorithms can be trusted, we may freely use them without checking their work. For example, $B$ will include a verified rewriter.

• Potential target for other systems. External programs may be able to construct higher-level, $B$- or $B'$-style proofs more easily than low-level $A$-style proofs.

2 Sketch of our proposal

We will now give a more detailed tour of our proposal. We begin by discussing the notions of formal proof and formal verification (§2.1), the logic we will use (§2.2), the potential for errors when computers check proofs (§2.3), and our $A$-style proof checker (§2.4). We then turn our attention to the extended proof checkers ($A', A'', \ldots, B$) and describe how they can be implemented (§2.5), what types of proof methods they will implement (§2.6), and finally how they can be put to use in the proof of $B$’s soundness (§2.7).

2.1 Formal verification

Formal verification is the use of mathematical proof to show hardware or software designs have desirable properties. We adopt a Hilbert-esque notion of proof as a “step-by-step, syntactically checkable deduction as may be carried out within a consistent, formal logical calculus.” [DLP77] We call this formal proof.

The “very idea” of formal verification was challenged by James Fetzer [Fet88], who argued programs are not mathematical entities because they run on computers; hence, as in other applied sciences where measurements are imprecise and natural laws are imperfectly understood, strict deductive proofs are inappropriate. But we, like Hoare [Hoa69], view Computer Science as an “exact science” of abstract mathematics; our programming languages
have pure semantics independently from any physical computers, and it is through these semantics we wish to analyze a program’s design.

DeMillo, Lipton, and Perlis [DLP77] contested the value of formal proof, arguing proof is instead the social process whereby mathematicians come to agree a formula is a theorem. We call this *informal proof*. Formal proofs, they argued, are too long and detailed to be believable, and cannot convey intuition to the reader. This objection was not widely accepted [Mac01, §6], for as Fetzer [Fet88] observed, the validity of a formula and our belief in its validity are distinct; Winston and O’Brien may agree two plus two makes five, but their consensus does not make it so. In contrast, only truths may be derived in a sound logical framework, so formal proofs can serve as “objective evidence” of the truth of a statement.

There is little hope mathematicians will be willing or able to prove properties about interesting programs informally, as “the proofs of even very simple programs run into dozens of printed pages.” [DLP77] But unlike the vaguely-defined social process behind informal proofs, formal proofs involve only simple rules whose application can be checked by computer programs. By automating the construction and checking of formal proofs, formal verification becomes possible.

### 2.2 Our choice of logic

Before we can build and check formal proofs, we must decide upon a “formal logical calculus” to use. Modern theorem provers do not agree on any standard, and this choice is “a matter of taste and experience” [LP99] which may be viewed “eclectically and pragmatically.” [Mac01, §8]

We propose using a simplified version of the ACL2 logic [KM98, KMM00]. Our objects will be the symbols and naturals, recursively closed under ordered pairing. We will eliminate guards [KM94, §4.3] and packages to simplify the connection with Common Lisp. We will also adopt infinitely-many primitive constants and a new rule, called base evaluation, for applying the basic functions like `cons` and `+` to constants; this is much like McCarthy’s [McC60] Lisp interpreter, `apply`, which had special cases to evaluate “elementary S-functions” like `cons`.

The major characteristics of the ACL2 logic will be preserved. Our logic will be first-order, will lack explicit quantifiers, and will have equality as its only predicate symbol. We will directly adopt Shoenfield’s [Sho67] rules of propositional calculus and ACL2’s instantiation and induction rules. We will
permit the introduction of total, untyped, recursive functions, the introduction of Skolem functions, and induction up to \( \varepsilon_0 \).

Finally, parting with ACL2 to follow the work of G"odel [G"od31], we will extend our logic with an integrated proof checker so we may establish metatheorems about provability, e.g., “\( A' \) only accepts provable formulas.” We will also add a rule of computational reflection as described by Harrison [Har95], to allow the use of metatheorems during proofs, e.g., “\( A' \) accepts \( \phi \), so \( \phi \) must be true.”

The logic just described will be rather restrictive, notably lacking types, quantifiers, and higher-order functions. But, as Kaufmann, Manolios, and Moore [KMM00] have noted, these limitations often “can be overcome without undue violence to the intuitions you are trying to capture.” As evidence of this claim, consider the diverse uses of the similarly-restrictive ACL2 system, which include the verification of:

- processor models [BKM96, Moo98, GWH00, Saw00],
- RTL designs [Rus98, RF00],
- circuit models [Hun00, HR05, HR06],
- virtual machines [BM96, LM03],
- compilers [BT00, Goe00],
- imperative programs [LM04], and
- other algorithms [RMT03, TB03, RRAHMM04, MZ05].

There are also advantages to using a simple logic. For example, term quotation and reflection are more straightforward when no types are involved and term equality does not rely on reductions [Bar01, How88]. Also, because our terms are so simple, our system will not need a type checker, type inference engine, or much in the way of interfacing layers such as parsers and term rendering.

2.3 Computers checking proofs

Formal proofs are too long for humans to check reliably, but computers are well suited to this task. Our proof checking program, which we called \( A \) in the
introduction, will be a function called \textit{Proofp}, defined in our logic; our logic will be compatible with Common Lisp, so we can use a Lisp system to run this function on a computer. There are several Lisp implementations, operating systems, and hardware platforms to choose from, as shown in Figure 1.

<table>
<thead>
<tr>
<th>Allegro</th>
<th>CLISP</th>
<th>CMUCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux x86, x86-64, PPC</td>
<td>Various Linux, BSD</td>
<td>Linux x86, Alpha</td>
</tr>
<tr>
<td>MacOSX PPC, Intel</td>
<td>MacOSX (Fink)</td>
<td>Various BSD x86</td>
</tr>
<tr>
<td>Windows 32/64</td>
<td>Windows (Cygwin)</td>
<td>MacOSX PPC</td>
</tr>
<tr>
<td>Solaris SPARC, AMD64</td>
<td>Solaris x86, SPARC</td>
<td>Solaris SPARC</td>
</tr>
<tr>
<td>Misc. Unix 32/64</td>
<td>Misc. Unix 32/64</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GCL</th>
<th>OpenMCL</th>
<th>SBCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux x86, PPC, SPARC, ...</td>
<td>Linux x86-64, PPC</td>
<td>Linux x86, PPC, SPARC, ...</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>MacOSX x86-64, PPC</td>
<td>Various BSD x86</td>
</tr>
<tr>
<td>Windows 32 (Cygwin)</td>
<td>Windows (Cygwin)</td>
<td>MacOSX PPC, Intel</td>
</tr>
<tr>
<td>Solaris SPARC</td>
<td>Solaris SPARC</td>
<td>Solaris x86, SPARC</td>
</tr>
</tbody>
</table>

Figure 1: A sampling of Common Lisp systems

Computers, operating systems, and Lisp compilers are not perfect, and their defects might cause our program to incorrectly accept an invalid proof. To make this less likely, we suggest using a heterogeneous collection of platforms when checking proofs of interest. This idea, called \textit{n-version programming} [Avi95], is not without precedent in computer-assisted proof [Mac01, ch. 4]. Because separate groups have independently developed these hardware platforms, operating systems, and Lisp implementations\footnote{CMUCL and SBCL are forks of the same code base, but the other Lisps are original.}, it is unlikely a diverse combination of these systems will share an equivalent error.

This argument is admittedly informal, but as Dijkstra [Dij82] wrote, “\textit{Scientific thought} derives its effectiveness from our willingness to acknowledge the smallness of our heads” and deal with problems “in depth and in isolation.” There are no formally verified processors, operating systems, and programming environments available for us to use, and we must start somewhere.
2.4 Our proof checker

Gödel [Göd31] defined a proof checker called Bu² within his logic in order to prove his incompleteness theorem. Bu depended upon 43 auxiliary definitions which dealt with encoding proofs as objects in the logic, and with recognizing valid, encoded proof steps.

Our logic will be more complex, and our Proof function will make use of auxiliary definitions including primitive Lisp functions (such as cons, natp, and +), basic list utilities (e.g., len, app, and memberp), recognizers and constructors for terms and formulas, substitution operations, and recognizers for valid proof steps. Our working draft of Proof involves around 100 definitions, totalling under 1,000 lines of Lisp. In addition to Proof, we will also have a small command loop to read instructions from input files.

It will be difficult to write interesting Proof-checkable proofs since Proof only implements basic rules of inference and provides no automation. However, it is simply written and is short enough to be thoroughly reviewed. Part of our dissertation will be an explanation of why this program correctly implements our logic, and why our logic is reasonable.

2.5 Proof checker extensions

For our system to be useful, we will need to make proof construction easier and more automatic. We plan to do this by adding new proof methods that are not found among the inference rules of our logic. In the introduction, we called these $A', A'', \ldots, B$. To maintain our trust in the system, we will need to ensure these new proof methods are sound, i.e., they can only be used to derive provable formulas.

Our Proof function is shown graphically in Figure 2. For simplicity we will ignore the function arities, axioms, and theorems it takes as inputs, and only write $Proof(x)$, where $x$ is the alleged proof to check. Proofs will be made up of proof steps, and each step will have a method of proof, a conclusion, and (except for axioms and theorems) some subproofs. A proof step will be acceptable when its method can be applied to its subproofs to obtain its conclusion, e.g., the contraction method can be applied to a subproof of $\phi \lor \phi$ to conclude $\phi$. A proof will be accepted by Proof when all its steps are proper applications of the inference rules of our logic.

\footnote{\textit{Short for Bewisfigur, German for proof figure.}}
An extension will be a new proof checking function, say Proof2p, that accepts all the proof methods known to Proofp and also some new methods. For example, perhaps Proof2p will add Modus Ponens or a tautology checker. We will say Proof2p is sound when we can prove the soundness claim,

\[ \forall x : \text{Proof2p}(x) \rightarrow \text{Provablep}(\text{Conclusion}(x)), \]

where Provablep(\phi) is defined as,

\[ \exists p : \text{Proofp}(p) \land \text{Conclusion}(p) = \phi. \]

If this claim is a theorem, then Proof2p does not allow us to prove anything that cannot be proven by Proofp.

We could establish soundness claims by hand or with another theorem prover, but our trust in Proof2p would then depend on external proofs. To avoid this, we propose defining our extensions as functions in our logic, so we can express these soundness claims in our logic and prove them using only Proofp and already-verified extensions.

In the end, we will trust all formulas accepted by Proofp are valid because Proofp is short and simple enough to review thoroughly. One of these formulas will state Proof2p cannot accept formulas that are not accepted by Proofp. Hence, we can trust any formula accepted by Proof2p is true.

### 2.6 Proposed extensions

We plan to add several proof methods, which we have classified in Figure 3 as either derived rules of inference or heuristic theorem proving.
We will begin with derived rules of inference for manipulating propositional formulas, such as Modus Ponens and double negation rules. We will then add rules for equality, such as its transitivity and commutativity, and the substitution of equal terms for arguments to functions and lambdas.

More interesting derived rules of inference will include the deduction law, which allows us to prove $F \rightarrow G$ by temporarily assuming $F$ is true, then showing $G$ follows. Other classic metatheorems will follow the work of Shoenfield [Sho67] and Shankar [Sha94], and will include:

- **Tautology checking** for identifying propositional tautologies,

- **Equivalence substitution** for propositional formulas, i.e., replacing occurrences of $F$ with $G$ and vice versa after proving $F \leftrightarrow G$, and

- **Equality substitution** for formulas, i.e., replacing occurrences of $t_1$ for $t_2$ and vice versa after proving $t_1 = t_2$.

These proof techniques do not embody a generic proof strategy and do not take advantage of user-created knowledge such as lemma libraries. To address these deficiencies, our heuristic theorem proving extensions are intended to mimic the key strategies used by ACL2: to prove a formula, we will compile it into an equivalent term, which will be converted into a clause; we will simplify the clause by applying conditional rewrite rules, equality reasoning, and calculation. This approach allows previously-proven lemmas

---

### Figure 3: Overview of our extensions

<table>
<thead>
<tr>
<th>Derived Rules of Inference</th>
<th>Heuristic Theorem Proving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional Rules</td>
<td>Formula Compilation, Clausification</td>
</tr>
<tr>
<td>Equality Rules</td>
<td>Calculation of Ground Terms</td>
</tr>
<tr>
<td>Lambda Reduction Rules</td>
<td>Equality Reasoning</td>
</tr>
<tr>
<td>The Deduction Law</td>
<td>Conditional Term Rewriting</td>
</tr>
<tr>
<td>The Tautology Theorem</td>
<td>Destructor Elimination</td>
</tr>
<tr>
<td>Equivalence Substitution</td>
<td>Clause Simplification</td>
</tr>
<tr>
<td>Substitution of Equals for Equals</td>
<td></td>
</tr>
</tbody>
</table>

---

9
to be automatically reused in new proof attempts, so the user can focus on developing strategies for the prover to follow on its own.

2.7 Using extensions

Logicians often appeal to metatheorems during otherwise-formal proofs. For example, after proving the tautology theorem, one might say, “since $\phi$ is a tautology, let $p$ be a proof of $\phi$,” without explicitly saying how $p$ is to be constructed. In other words, we feel free to substitute the metatheoretically established “$\phi$ is provable” for a formal proof of $\phi$.

Reflection refers to techniques that capture this idea without a separate metalogic. In our system, the metatheoretic notion “$\phi$ is provable” will be an ordinary, formal proof of $\text{Provable}(\Gamma \phi \gamma)$, where $\Gamma \phi \gamma$ is the encoded form of $\phi$. To use this metatheorem, we will support computational reflection [Har95]:

\[
\begin{align*}
\text{Provable}(\Gamma \phi \gamma) \\
\phi
\end{align*}
\]

Suppose $\text{Proof2p}$ is an extended proof checker that adds tautology checking, and we have proven its soundness claim. We now want to be able to prove formulas with $\text{Proof2p}$ and its built-in tautology checker instead of using the less-capable $\text{Proofp}$. Our reflection rule is one of two capabilities needed for this; efficient computation of ground terms is the other. Suppose $p$ is a $\text{Proof2p}$-level proof of $\phi$. Then, we can convert $p$ into a $\text{Proofp}$-level proof as follows:

1. $\forall x, \text{Proof2p}(x) \rightarrow \text{Provable}(\text{Conclusion}(x))$ Soundness Theorem
2. $\text{Proof2p}(p) \rightarrow \text{Provable}(\text{Conclusion}(p))$ Instantiation
3. $\text{Proof2p}(p) \rightarrow \text{Provable}(\Gamma \phi \gamma)$ Computation
4. $\text{Proof2p}(p)$ Computation
5. $\text{Provable}(\Gamma \phi \gamma)$ Modus Ponens
6. $\phi$ Reflection

Recall that our logic will be compatible with Common Lisp, and our extensions such as $\text{Proof2p}$ will be functions in our logic. As a result, a Common Lisp system can run our extensions. To justify steps 3 and 4 above, we will use Lisp to evaluate $\text{Conclusion}(p)$ and $\text{Proof2p}(p)$.

Many theorem provers, including ACL2, PVS, Isabelle/HOL [BN02], and Coq, allow evaluations in a programming language to be treated as proofs of
equality. But Harrison [Har95] has referred to this transition as a “glaring leap of faith,” and Lisp evaluation is certainly no basic rule of inference. To help justify our approach, an early extension will be an evaluator in the spirit of McCarthy’s [McC60] function apply, and we will prove our evaluator produces a value that is logically equal to its input term. We believe our evaluator correctly models the semantics of Lisp evaluation for our fragment of the language, but this argument can only be made informally since Lisp’s evaluator is not defined in our logic.

3 Present and remaining work

In this section, we describe our logic (§3.1), the working draft of our proof checker (§3.2), and the construction of Proofp-level proofs (§3.3). We then introduce a simple extension of Proofp (§3.4) and explain how we were able to verify this extension (§3.5). Finally, we remark upon what still needs to be done (§3.6).

3.1 Sketch of our logic

Our universe, $U$, contains the naturals and symbols, closed under ordered pairing. We take some notational conventions from Lisp:

- We write the ordered pair of $a$ and $b$ as $(a \cdot b)$,

- $(x_1 x_2 \ldots x_n \cdot b)$ is shorthand for $(x_1 \cdot (x_2 \ldots x_n \cdot b))$,

- $(\cdot)$ is shorthand for the symbol nil,

- $(x)$ is shorthand for $(x \cdot \text{nil})$, and

- $(x_1 x_2 \ldots x_n)$ is shorthand for $(x_1 \cdot (x_2 \ldots x_n))$.

Our terms are primitive constants, variables, function applications, and $\lambda$ abbreviations. We write terms in the typewriter font.

- We have a primitive constant for every $x \in U$, which we will write as '$x$. For example, '$(1 \cdot 2)$ is the constant corresponding to $(1 \cdot 2)$.

- We have a variable for every symbol except t and nil. For example, $x$, $y$, $\text{foo}$, and $\text{bar}$, are variables.
We have a function name for every symbol except nil; quote; first; second; third; fourth; fifth; and; or; list; cond; let; let*; pequal*; por*; and pnot*. We associate an arity with each function name. A function application is written as $(f \ t_1 \ldots \ t_n)$ where $f$ is a function name of arity $n$, and each $t_i$ is a term.

A lambda abbreviation is written as $((\lambda \ x_1 \ldots \ x_n) \ \beta) \ t_1 \ldots \ t_n)$, where each $x_i$ is a distinct variable, and $\beta$ and each $t_i$ are terms. To make substitution simpler, $\beta$ may have no free variables besides the $x_i$.

Since we do not allow $t$ or nil to be variable symbols, no ambiguity arises when we omit the quotes on 't and 'nil; similarly we will not bother to quote numbers when we wish to use them as constants.

Our formulas are equalities between terms (written $t_1 = t_2$), negations (written $\neg F$), and disjunctions (written $F \lor G$). Other connectives (e.g., $\rightarrow$, $\wedge$, and $\leftrightarrow$) are treated as abbreviations. Our formulas have no quantifiers, and we interpret free variables in formulas as universally quantified at the top level.

Our rules for propositional calculus are shown in Figure 4, and our rules for equality are shown in Figure 5. Most of these are taken from ACL2 [KMM00, §6] and Shoenfield [Sho67].

We take for granted certain base functions, inspired by Common Lisp, shown in Figure 6. These functions are total; they can be applied to any objects in our universe. For each base function $f$ of arity $n$, and for all constants $c_1, \ldots, c_n$, we add an axiom of the form $(f \ c_1 \ldots \ c_n) = x$, where $x$ is the appropriate constant. These axioms allow us to use primitive calculations in proofs, e.g., we can prove $(+ \ 1 \ 2) = 3$ in one step. These axioms also allow us to develop an evaluator for arbitrary ground terms, based on McCarthy’s [McC60] apply function.

Like Kaufmann and Moore [KM98, p. 31–47], we add several “symbolic axioms” to explain the behavior of our base functions, e.g., $(\text{consp} \ (\text{cons} \ x \ y)) = \text{t}$ and $x \neq y \rightarrow (\text{equal} \ x \ y) = \text{nil}$. We also add an encoding of the ordinals up to $\varepsilon_0$, taken from Manolios and Vroon [MV06], which we will use for termination proofs and to justify induction. Our induction rule is based on ACL2’s rule [KMM00, p. 80], and is shown in Figure 7. We finally define our proof checker in the logic so we can reason about provability.
\[ \neg A \lor A \] Propositional schema

\[ A \lor A \] Contraction

\[ A \]

\[ B \lor A \] Expansion

\[ A \lor (B \lor C) \]

\[ (A \lor B) \lor C \] Associativity

\[ A \lor B \]

\[ \neg A \lor C \]

\[ B \lor C \] Cut

\[ A \]

\[ A/\sigma \] Instantiation

Figure 4: Propositional rules

\[ x = x \] Reflexivity axiom

\[ x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_1 = x_2 \rightarrow y_1 = y_2 \] Equality axiom

\[ x_1 = y_1 \rightarrow \cdots \rightarrow x_n = y_n \rightarrow \]

\[ (f \; x_1 \; \ldots \; x_n) = (f \; y_1 \; \ldots \; y_n) \] Functional equality schema

\[ (\lambda \; (x_1 \; \ldots \; x_n) \; \beta) \; t_1 \; \ldots \; t_n) = \]

\[ \beta/\left[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\right] \] Beta reduction schema

Figure 5: Equality axioms
(if x y z)  Returns y if x is non-nil, z otherwise
(equal x y)  Checks if x and y are the same

(consp x)  Checks if x is a pair
(natp x)  Checks if x is a natural
(symbolp x)  Checks if x is a symbol

(cons x y)  Builds the pair (x . y)
(car x)  Accesses the first element of an ordered pair*
(cdr x)  Accesses the second element of an ordered pair*

(< x y)  Checks if x is less than y†
(+ x y)  Performs natural-number addition of x and y†
(- x y)  Performs natural-number subtraction of y from x†

(symbol-< x y)  Checks if x is a smaller symbol than y†

* after interpreting non-pair arguments as (nil . nil)
† after interpreting non-natural arguments as 0
‡ after interpreting non-symbolic arguments as nil

Figure 6: Base functions
Induction rule.

We may derive a formula, $F$, from:

- A term, $m$, called the measure,
- A set of formulas, $\{q_1, \ldots, q_k\}$,
- For each formula $q_i$, a set of substitution lists, $\Sigma_i = \{\sigma_{i,1}, \sigma_{i,2}, \ldots, \sigma_{i,h_i}\}$, and
- Proofs of each of the following formulas:
  - **Basis step**
    $$ F \lor q_1 \lor \cdots \lor q_k $$
  - **Inductive steps**
    For each $1 \leq i \leq k$,
    $$ F \lor \neg q_i \lor \neg F/\sigma_{i,1} \lor \cdots \lor \neg F/\sigma_{i,h_i} $$
  - **Ordinal step**
    $$(\ord m) = t$$
  - **Measure steps**
    For each $1 \leq i \leq k$ and $1 \leq j \leq h_i$,
    $$ \neg q_i \lor (\ord{m/\sigma_{i,j}}) = t $$

* $\ord{m}$ is our recognizer for encoded ordinals
† $\ord< m$ is our well-ordering on encoded ordinals

Figure 7: Induction rule
3.2 Our proof checker

We have developed a draft of our proof checker, which is complete except for the reflection rule and efficient computation. We have ported our program to several Lisp implementations. Our proof checker is similar to Gödel’s [Göd31, pp. 163–171] Bw program, which was based on 43 auxiliary definitions, including:

- Basic operations to encode recursive structures as prime powers (1–10),
- Constructors for encoded formulas (13–16),
- Recognizers for encoded variables, types, formulas, and sequences of formulas (11, 12, 17–23),
- Variable binding and substitution operations (25–33, 37), and
- Recognizers for valid proof steps (34–36, 38–43).

Our logic is more complex, and our Proofp program relies upon around 100 definitions.

Gödel encoded terms as numbers using prime powers. Our encoding is more readable since we can use lists and symbols.

- We encode ‘\(x\) as (quote \(x\)),
- We encode the variable for the symbol \(v\) as \(v\),
- We encode \((f \; t_1 \ldots \; t_n)\) as \((f \; \gamma \; t_1 \ldots \; t_n)\), where \(\gamma\) represents the encoding of \(t_i\), and
- We encode \((\lambda \; (x_1 \ldots \; x_n) \; \beta) \; t_1 \ldots \; t_n\) as
  \[((\text{lambda} \; (x_1 \ldots \; x_n) \; \gamma \; \beta) \; \gamma \; t_1 \ldots \; \gamma \; t_n).\]

Since we do not allow quote to be used as a function name, there is no confusion as to whether an encoded term represents a constant or a function call. Similarly, we do not permit pequal*, pnot*, or por* to be used as function names, so we can encode the formulas without overlapping the terms as follows:

- We encode \(t_1 = t_2\) as \((\text{pequal*} \; \gamma \; t_1 \ldots \; \gamma \; t_2)\),
• We encode \( \neg F \) as \((\text{pnot* } \neg F)\), and

• We encode \( F \lor G \) as \((\text{por* } \neg F \lor \neg G)\).

Finally, we encode proofs using appeals. Each appeal represents a single step in the proof, and is a tuple of the form:

\([(\text{method conclusion [subproofs] [extras]}))\),

where:

• The \textit{method} is the name of the proof rule being used,

• The \textit{conclusion} is the formula this step purports to prove,

• If present, the \textit{subproofs} are a list of subsidiary appeals which must also be checked before this appeal can be considered valid (rules of inference have subproofs, while axioms are “atomic” and do not), and

• If present, the \textit{extras} contain any additional information needed to justify this step, e.g., an appeal to \textit{instantiation} should specify the substitution to be used.

For each rule of inference, we introduce a function to check if an appeal is a valid application of the rule. For example, our instantiation rule allows us to prove \( A/\sigma \) from a proof of \( A \), so we write the function \textit{InstantiationOkp}(x), which checks that:

• The method is instantiation,

• There is a single subproof, call its conclusion \( A \),

• The extras contain a substitution list, call it \( \sigma \), and

• The conclusion is \( A/\sigma \).

Finally, we introduce \textit{ProofStepOkp}, which checks if a single step in the proof is valid by inspecting its method and calling the appropriate rule-checker. We recursively extend \textit{ProofStepOkp} across the entire proof to obtain \textit{Proofp}, our whole-proof checker.
3.3 Building $\text{Proofp}$-checkable proofs

It is impractical to write $\text{Proofp}$-checkable proofs by hand. For example, even our proof of $x \neq y \rightarrow z \neq x \lor z \neq y$, shown in Figure 8, takes a full page. Accordingly, we have developed functions to construct proofs for us. These builders typically use some input proofs, formulas, or terms to create a new proof of a certain shape. We do not need to trust these builders since we can check their output with $\text{Proofp}$.

We begin with simple builders for our primitive rules. For example, $\text{BuildPropSchema}$ conses together a proof of $\neg A \lor A$ given the formula $A$, and $\text{BuildCut}$ conses together a proof of $B \lor C$ given proofs of $A \lor B$ and $\neg A \lor C$. These are used to create new builders that act like derived rules of inference, e.g., “commutativity of or” is not a primitive rule, but we can derive $B \lor A$ from a proof of $A \lor B$ as follows:

1. $A \lor B$ Given
2. $\neg A \lor A$ Propositional schema
3. $B \lor A$ Cut

We can translate these steps into a function, $\text{BuildCommuteOr}$, which creates a proof of $B \lor A$ using a proof, $x$, of $A \lor B$ as input:

$$\text{BuildCommuteOr}(x : A \lor B) = \text{BuildCut}(x, \text{BuildPropSchema}(A)).$$

We have developed many builders, including functions for manipulating propositions, reasoning about logical equality, dealing with lambdas, and handling the special equal, if, iff, and not functions. Our most sophisticated builders perform large tasks such as if-lifting, clause splitting, evaluation, and rewriting. These tools allow us to describe the proofs we wish to construct more concisely, but the proofs they generate can become large and checking them can be computationally expensive.

3.4 Extending $\text{Proofp}$ with a new rule

We have verified a new proof checker, $\text{Proof2p}$, which extends proofp by additionally accepting “commutativity of or” inferences in one step. Since proofp only needed two steps to achieve the same effect, the added power of $\text{Proof2p}$ over $\text{Proofp}$ is negligible. But verifying this extension required us to use proofp to reason about itself and its relationship to $\text{Proof2p}$, and shows we can handle all the “generic” work involved with verifying extensions.
Figure 8: Proof-checkable proof of $x \neq y \rightarrow z \neq x \lor z \neq y$
The definition of \( \text{Proof2p} \) is straightforward. We begin by introducing a new function, \( \text{CommuteOrOkp}(x) \), which accepts the appeal \( x \) only if:

- The method is commute-or,
- There is a single subgoal whose conclusion has the form \( A \lor B \), and
- The conclusion is \( B \lor A \).

We then introduce \( \text{Proof2StepOkp} \), which accepts the appeals recognized by \( \text{CommuteOrOkp} \) and \( \text{ProofStepOkp} \). We finally introduce \( \text{Proof2p} \), which recursively ensures every step in a proof is \( \text{Proof2StepOkp} \).

To show we can trust \( \text{Proof2p} \), we need to show it only accepts proofs of formulas that could be proven by \( \text{Proofp} \). That is, we want to show:

\[
\forall x : \text{Proof2p}(x) \rightarrow \text{Provablep}(\text{Conclusion}(x)),
\]

where \( \text{Provablep}(\phi) \) is defined as:

\[
\exists p : \text{Proofp}(p) \land \text{Conclusion}(p) = \phi
\]

The informal proof is straightforward. Assume \( \text{Proof2p}(x) \) holds, and inductively assume all the subproofs of \( x \) consist entirely of \( \text{Proofp} \)-level steps. If \( x \) is a not a commute-or step, then \( x \) is already accepted by \( \text{Proofp} \). Otherwise, \( x \) must be a commute-or step; assume it concludes \( B \lor A \) from its subproof of \( A \lor B \), and apply \( \text{BuildCommuteOr} \) to this subproof to obtain an entirely \( \text{Proofp} \)-level proof of \( x \)'s conclusion.

Our strategy for verifying other extensions is similar: we write a builder function that mimics our extension and can build a proof of any conclusion the extension might make. We then show whenever we use the extension in a high-level proof, our builder could have been used to create an equivalent low-level proof. Hence, the extension can only be used to accept provable formulas.

### 3.5 Proving the new rule is sound

We have constructed a \( \text{Proofp} \)-checkable proof of our soundness claim for \( \text{Proof2p} \). This was particularly challenging since, having no verified extensions to work with, our proof could use only the primitive inferences.

Our proof effort was carried out in three phases:
• Phase 1. We developed a collection of proof tools (e.g., an evaluator, a clause splitter, a rewriter) as ACL2 functions, and we used ACL2 to “informally” prove these tools are sound.

• Phase 2. We used these tools to build a Proofp-style proof of the soundness of Proof2p. This effort involves “translating” the ACL2-style proofs from Phase 1 into Proofp-checkable objects.

• Phase 3. We checked the proofs from Phase 2 with our small command loop program. This program is independent of ACL2 and can run in many Lisp environments.

Phase 1. We first focused on creating the tools we expected to need to prove Proof2p and other extensions are sound. For example, we developed an evaluator and a rewriter which can build Proofp-checkable proofs to justify their claims. We could have written these tools in any language, but developing them in ACL2 had two important advantages.

First, it gave us a straightforward way to turn our tools into extensions of Proofp. This would not be important if our goal were only to verify Proof2p, but we eventually want to develop a useful theorem prover which includes facilities like evaluation and rewriting. Writing our tools in ACL2, which is almost the same as our logic, will make it easy to introduce these functions into our system. If we had instead implemented these tools as C programs or Perl scripts, it would not be nearly so straightforward to reuse them in this way.

Second, using ACL2 allowed us to “informally” prove our tools are sound. Having these proofs available allowed us, in Phase 2, to focus on recreating rather than discovering proofs. To ease this translation and lessen the number of tools we needed to implement, we developed our ACL2 proofs without using the more sophisticated features of ACL2, and restricted ourselves to rewriting with lemmas, execution, destructor elimination, and induction.

Our ACL2 proof style is library-centric: we focus on creating collections of rules which, together, can greatly simplify the terms we encounter during proofs. Like Bevier [Bev87] we keep our function definitions disabled — that is, the theorem prover only attempts to use definitions when it is explicitly told to do so. But unlike Bevier, we keep our rewrite rules enabled and focus on making them work well together. Sometimes we are able to predict useful rules as we introduce new functions, while other times we only realize we need a new rule after seeing a failed proof attempt. Because of this approach, our
ACL2 proof of the soundness claim for \textit{Proof2p} occurs only incidentally as one lemma among thousands of other rewrite rules.

\textbf{Phase 2.} To translate the soundness claim for \textit{Proof2p} into a \textit{Proofp}-style proof, we focus on recreating these lemma libraries. To do this, we developed a tactic-style interface to our proof tools from Phase 1. This interface can be driven interactively: we can pose new conjectures, then perform rewriting with lemmas, case splitting, destructor elimination, and so forth. We also developed an \textit{auto} tactical, which automatically tries using these different approaches to make progress. As we prove each lemma, our rewriter becomes more useful because it now has another rule to use. In the end, we were able to construct the \textit{Proofp}-counterparts of the all the lemmas leading up to and including the soundness claim for \textit{Proof2p}.

When generating proofs, we made use of Boyer and Hunt’s [BH06] memoization and hash-consing extension of ACL2, which greatly increased our proof-building speed and memory efficiency. When we saved proofs to disk, we used a structure-sharing representation where repeated terms can be named and referred to later.

Sometimes our proofs grew too large to deal with. Typically this could be addressed by choosing more careful proof strategies, e.g., controlling when definitions are expanded to avoid introducing case explosions, and by introducing intermediate lemmas as necessary. Other times, our proof-building tools had to be improved. As the most severe example of this, an early draft of our evaluation builder was written without paying attention to proof sizes, and its proof of $\text{fib}(2) = 2$ was over 790 million conses; a revised draft took only 35,000 conses to build the same proof.

\textbf{Phase 3.} Finally, we checked the generated proofs to ensure they were valid. We have implemented a small Common Lisp program to run our proof checker. This program is is independent from ACL2 and does not include any memoization code. It processes a list of instructions, e.g., “add \textit{this} theorem which is justified by \textit{this} proof” and “admit \textit{this} function whose termination is guaranteed by \textit{this} proof.” The proofs are checked in order, one by one.

It took 7.6 hours to check all our lemmas using OpenMCL as the Lisp environment on our development machine (a 2.2-GHz AMD Opteron system with 32 GB of memory running 64-bit Linux). We then double-checked the proofs in 4.1 hours using CMUCL on a different machine (a 2.13-GHz Intel Core 2 processor with 4 GB of memory running 32-bit Linux). For our thesis, we also plan to check our proofs using additional platforms to minimize any chance that a computer error has inappropriately caused some proof to be
accepted.

How big is the proof of the soundness claim? Including all the lemmas leading up to and including our soundness claim, we have over 200 definitions and 2,000 rewrite rules whose proofs take nearly 2 GB of disk space. These proofs are generated by about 6,000 lines of code which drive our tactic interface, and this build process takes about 50 minutes to complete when running in parallel on our development machine (which has 8 cores). A more detailed breakdown is shown in Figure 9.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Source lines</th>
<th>Defs</th>
<th>Rules</th>
<th>Build time</th>
<th>Proof size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilities</td>
<td>3,086</td>
<td>69</td>
<td>847</td>
<td>8.6m</td>
<td>488 MB</td>
</tr>
<tr>
<td>Logic</td>
<td>2,750</td>
<td>131</td>
<td>1,179</td>
<td>39.4m</td>
<td>1.4 GB</td>
</tr>
<tr>
<td>Build</td>
<td>389</td>
<td>28</td>
<td>114</td>
<td>6.6m</td>
<td>36 MB</td>
</tr>
<tr>
<td>Demo</td>
<td>132</td>
<td>6</td>
<td>42</td>
<td>20m</td>
<td>57 MB</td>
</tr>
<tr>
<td>Total</td>
<td>6,357</td>
<td>234</td>
<td>2,182</td>
<td>51m</td>
<td>1.9 GB</td>
</tr>
</tbody>
</table>

Statistics as of SVN Revision 401, Aug 2007. Line count excludes comments and blank lines. Rule count excludes trivial definition rules. Build time computed with omake -j 8 on lhug-1.cs.utexas.edu using ACL2 3.2.1 on OpenMCL. Total time is better than the sum since added parallelism is available when building multiple directories.

Figure 9: Proof size metrics by directory

Why is so much work needed for such a trivial extension? As Figure 9 suggests, the vast majority of our proof effort is not related to this particular extension at all, but instead is spent laying the groundwork for reasoning about arithmetic, lists, maps, terms, formulas, substitution, and proofs (the utilities and logic directories).

3.6 Remaining work

In Section 2.6 we proposed developing a number of extensions, many of which are far more complicated than the commutativity of or. The major remaining challenge is to show the rest of these extensions are sound using only Proofp and already-verified extensions.

We have already implemented these extensions in ACL2, and have created ACL2 proofs showing they are sound. In other words, we have completed Phase 1 for the entire project. We still need to translate these ACL2 proofs into the proper Proofp-checkable (or Proof2p-checkable, etc.) form. We believe this should be possible, as evidenced by our ability to verify the
commutativity of or extension. The tools we developed to complete this verification, and the thousands of lemmas we have proven and can now reuse, should give us a useful starting point for this work. As we progress, proof size should become less of an issue, since introducing new proof methods will allow higher-level proofs to be more compact. Nevertheless, we will likely need to further improve our proof tools, and we may also need to develop some new tactics.

We also need to address the issue of evaluation. We will need to set up an efficient evaluator (i.e., a call of Lisp’s eval function), fairly early in the stack of extensions so the reflective transition from higher-level proofs to Proofp-level proofs can be done effectively. Ideally, this would be our first extension, but proof size issues might force us to introduce some intermediate extensions first. If this proves difficult, an alternate approach would be to use Proof2p and later proof checkers directly, instead of extending Proofp with a reflection rule.

4 Related work

4.1 Current theorem provers

There are several general-purpose proof systems available, including the Boyer-Moore provers ACL2 [KM97] and NQTHM [BKM95]; higher-order logic provers such as HOL 4 [GCM05], HOL Light [Har96], Isabelle/HOL [NPW02], and PVS [ORSSC98]; and constructive type theory provers such as Coq [BC04] and Nuprl [TPG95].

Our logic is a slight variant of the ACL2 logic [KM98], which is the least expressive among these systems. The other systems provide quantifiers and higher-order functions, and use type systems to avoid logical paradoxes. The logics of Nuprl and Coq are intuitionistic, though this does not seem to have much impact on hardware and software verification. Despite the restrictiveness of our logic, many formal verification problems can still be expressed.

In the ACL2 system, proofs are “whatever the defthm command accepts.” This proof search is influenced by a database of implicit rules and also by explicit hints, and may involve rewriting, arithmetic reasoning, induction, BDDs, and other techniques. These proof methods are highly complex and do not resemble the rules of the ACL2 logic. The program has not been sub-
jected to any rigorous, formal analysis, and soundness bugs have occasionally
been discovered in official releases (see Figure 4.1). Proof attempts create
human-readable logs which explain at a high level what the prover is doing,
but these are not suitable for checking by other programs.

HOL systems have a more explicit notion of proof. Theorems in HOL
are objects of type \texttt{thm} and represent sequents $\Gamma \vdash t$ where $\Gamma$
is a set of assumptions and $t$ is a conclusion. The \texttt{thm} type is \textit{abstract},
so the only way to construct a \texttt{thm} is to use primitive, built-in functions corresponding to
HOL’s rules of inference. For example, HOL’s reflexivity rule is:

\[
\emptyset \vdash t = t
\]

The corresponding function, \texttt{REFL}, takes a \texttt{term}-typed argument $t$
as input and produces the \texttt{thm} with no assumptions and conclusion $t = t$. As another
example, HOL’s rule for discharging assumptions is:

\[
\Gamma \vdash t_2
\]
\[
\frac{}{\Gamma - \{t_1\} \vdash t_1 \rightarrow t_2}
\]

In other words, if $t_2$ follows from $\Gamma$, then $t_1 \rightarrow t_2$ follows after we remove $t_1$
from $\Gamma$. The corresponding function, \texttt{DISCH}, takes $t_1$ and a \texttt{thm} of the form
$\Gamma \vdash t_2$ as inputs, and produces the \texttt{thm}, $\Gamma - \{t_1\} \vdash t_1 \rightarrow t_2$.

If the type system is implemented correctly, the only way to create a \texttt{thm}
object is to invoke functions like \texttt{REFL} and \texttt{DISCH}. As a result, any \texttt{thm}-type
object must have been created entirely by following the rules of inference.
Consequently, the intermediate steps of a proof need not be stored, although
sometimes proof recording schemes have been added to HOL systems [Won95,
BN00, OS06] to facilitate proof translation or external double-checking.

The PVS system [ORS92] also uses an abstract type to represent theo-
rems, but provides more powerful primitives than HOL systems, particularly
HOL Light [Har96]. For example, rewriting with lemmas is a primitive rule
in PVS. Griffioen and Huisman [GH98] regarded PVS favorably, but were
somewhat critical on this point: “these decision procedures sometimes cause
soundness problems... PVS still seems to contain a lot of bugs and frequently
new bugs show up.”

Coq and Nuprl have another, well-defined notion of proof. Certain types
are called \textit{propositions}. Whenever the type of an object $x$ is a proposition, we
say $x$ itself is a \textit{proof} of that proposition. No abstract type is used; instead
## ACL2 Release

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Soundness Bugs Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>October, 1998 (Version 2.3)</td>
<td></td>
<td>Subversive recursions</td>
</tr>
<tr>
<td>August, 1999 (2.4)</td>
<td></td>
<td>Immediate force mode</td>
</tr>
<tr>
<td>June, 2000 (2.5)</td>
<td></td>
<td>Metafunctions with hypotheses</td>
</tr>
<tr>
<td>November, 2001 (2.6)</td>
<td></td>
<td>Linear arithmetic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Evaluation in proofs</td>
</tr>
<tr>
<td>November, 2002 (2.7)</td>
<td></td>
<td>Functional instantiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BDDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Guards</td>
</tr>
<tr>
<td>March, 2004 (2.8)</td>
<td></td>
<td>Tautology checking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACL2 arrays</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proof checker commands</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defining packages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tracking axioms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Type prescriptions in equivalences</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Redundancy and single-threaded objects</td>
</tr>
<tr>
<td>October, 2004 (2.9)</td>
<td></td>
<td>Package names</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tracking program mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Macro expansion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linear arithmetic</td>
</tr>
<tr>
<td>August, 2005 (2.9.3)</td>
<td></td>
<td>Program mode in defconst</td>
</tr>
<tr>
<td>February, 2006 (2.9.4)</td>
<td></td>
<td>Meta rules with local events</td>
</tr>
<tr>
<td>June, 2006 (3.0)</td>
<td></td>
<td>Program mode in local</td>
</tr>
<tr>
<td>August, 2006 (3.0.1)</td>
<td></td>
<td>Local table events</td>
</tr>
<tr>
<td>December, 2006 (3.1)</td>
<td></td>
<td>Package witnesses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Forcing in linear arithmetic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Redundancy and measures</td>
</tr>
<tr>
<td>April, 2007 (3.2)</td>
<td></td>
<td>Unknown/hidden packages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Meta rules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Redefinition and program mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Raw lisp code in tracing</td>
</tr>
</tbody>
</table>

Figure 10: Some corrected ACL2 soundness bugs
Source: ACL2 release notes [KM07]
the proof rules are directly encoded into the type system as typing rules. This is the Curry-Howard isomorphism: the proposition “$P$ implies $Q$” can be encoded as the arrow type of functions from $P$ to $Q$, i.e., $P \rightarrow Q$.

Like the thm approach, the correctness of these systems depends on a relatively small kernel. The type system needs to be correctly implemented, and the typing rules for propositions need to correspond to the logic. Since whole proof terms are stored, this is potentially less space-efficient than the abstract thm type approach. In Coq, this is somewhat alleviated by a complex notion of term equality wherein reducibly-equivalent terms are said to be equal. For example, Coq can prove $2+3 = 5$ with a single use of its reflexivity rule.

In most systems, proofs are constructed with fully-expansive, goal-directed scripts called tactics. If a tactic attempts to build a theorem with an invalid inference, an error will be caused and the proof attempt will fail. Tactics can often be combined into strategies or tacticals, which can respond to failure by trying other tactics, etc. Tactics need not be trusted and can be written by the user (whereas trusted code must be written by the theorem prover’s authors) since all their work will be checked by the thm constructors. Even though our system does not use an abstract thm type, we can emulate tactics by having them construct proofs instead of thms, and checking these proofs with Proofp.

The ACL2 system does not have an explicit notion of proof, and its version of tactics and tacticals, proof checker macros, are rarely used. Instead, the built-in rewriter is controlled by adding lemmas. Indirect advice about how to use these lemmas can also be added, making the approach somewhat flexible and automatic: the user focuses on setting up rules that will work well together, and the system applies this strategy to new problems. When the default strategy is insufficient to prove a troublesome conjecture, extra hints can be given by the user or automatically suggested by user-developed heuristics (see, e.g., [Dav04]).

Nothing prevents a tactic-based system from following the heuristic rewriting approach. Boulton [Bou92] has implemented tactics to emulate some of NQTHM’s automation in HOL, and lemma-based simplification is available in most provers, e.g., autorewrite in Coq, the rewrite package in Nuprl, rewrite_tac in HOL 4, and the simp tactic in Isabelle/HOL. Indeed, in our system we have primarily used a few tactics that emulate some of ACL2’s automation.
4.2 Embedded proof checkers

General-purpose theorem provers are expressive enough that new proof-checking programs can be expressed in their logics. In fact, the current draft of our system can be understood as a proof checker written inside ACL2. This has been done many times in various projects.

In a somewhat unique effort, Shankar [Sha94] wrote a proof checker in NQTHM for Shoenfield’s first-order logic with Cohen’s Z2 axioms in order to prove Gödel’s incompleteness theorem. His goal was to mechanically check classic metamathematical theorems, and he did not intend to produce a new theorem prover for everyday use. He used NQTHM to introduce extensions like tautology checking, but did not try to “bootstrap” these proofs into a form his proof checker could self-check.

More commonly, this approach has been used to study properties of simple proof checking programs. For example:

- J. von Wright [vW94a, vW94b] wrote a proof checker for higher-order logic in HOL. This involved defining a HOL specification, Is.proof, which describes the valid proofs. A primitive, imperative programming language was then defined within HOL, and a proof checking program was written in this language. HOL was used to show the imperative program implemented the high-level Is.proof specification.

- Ridge and Margetson [RM05] wrote a first order theorem prover as definitions in Isabelle/HOL and, using Isabelle/HOL, proved the program to be sound and complete. The program does some proof search, but they mention its performance is not competitive with typical resolution provers.

- Harrison [Har06] has mimicked the implementation of HOL Light, an OCaml program, as a HOL Light specification. By assuming an additional axiom about sets, he can show the encoded HOL system is consistent. Without the axiom, he can show the encoded system except for the axiom of infinity is consistent. These results indicate “something close to the actual implementation of HOL” is sound.

In each of these efforts, an existing prover is trusted and is used to prove properties about a new proof checking core. Our project is complementary:
we are willing to trust our small core, and our interest is in the verification of new, extended proof methods. We do not propose to investigate the soundness of our core mechanically.

4.3 Independent proof checking

There have also been some projects where one system is used to check the work of another. This idea is somewhat like Boulton’s suggestion [Bou93] of separating proof search from proof construction in HOL, but may also be useful for “porting” results obtained in one system to another. A few such projects include:

- McCune and Shumsky [MS00] have written ACL2 functions to check proof objects emitted by Otter, a resolution prover, for validity. The majority of the proof search can be offloaded onto Otter, and the ACL2 program only checks that Otter did not make a mistake. No attempt is made to verify the resolution prover, which is an optimized C program, but since the ACL2 program checks the proof transcript, ACL2 can be used to show the combined system is sound.

- Caldwell and Cowles [CC02] describe preliminary work on independently checking Nuprl proofs with a program written in ACL2. As they emphasize, “we are not making claims about the correctness of Nuprl itself,” which was seen as impractically hard: Nuprl’s implementation apparently involves a 60,000-line Lisp core and a 40,000-line ML interface, with 167 rules of inference which are sometimes complicated, e.g., the arith rule. The project is apparently still in the early stages.

- Obua and Skalberg [OS06] have extended HOL Light with a proof recorder that tracks calls to the proof constructors. A complex structure-sharing scheme is used in order to combat the size of proof objects, and many proofs can be read into Isabelle/HOL and checked independently from HOL Light. The authors speculate that adding “higher inference rules,” such as rewriting, might help to make the emitted proofs smaller, but have not apparently tried to implement such a scheme.

Our project takes a different approach. Rather than checking some unverified proof method did not make a mistake after its every use, we want to verify proof methods so we need not check their work.
4.4 Meta reasoning

Our system will have an integrated proof checker defined in its own logic so
we can directly reason about provability. This allows us to show new proof
techniques are sound and can be trusted.

Even without an integrated proof checker, other theorem provers have
some support for meta reasoning. Most of this work follows, with minor
differences, the metafunction approach [BM81], which involves five steps:

1. An encoding for the relevant terms is introduced,
2. A semantic function, $ meaning(term, env) $, is introduced to evaluate an
encoded term with respect to some assignment of variables,
3. A “metafunction”, $ fn(term) $, is introduced to simplify encoded terms,
4. The user proves $ meaning(fn(term), env) = meaning(term, env) $, for all
well-formed encoded terms and for all environments, to demonstrate $ fn $
can be trusted, and
5. Some evaluation mechanism allows $ fn $ to be used to simplify encoded
terms in proofs.

In ACL2, a standard encoding (quoting) can be used, and $ meaning $ func-
tions for a fixed set of concepts can be introduced using the $ defevaluator $ facility. A metafunction, $ fn $, is a regular ACL2 program, written as a recur-
sive function that manipulates encoded terms. A built-in mechanism allows
the system to begin using a metafunction after the soundness theorem is
proven.

Metafunctions can be a useful tool for advanced ACL2 users, but they
have limitations. They are subservient to the rewriter and cannot keep state
between invocations, i.e., for building up databases of facts [SNG+04]. Also,
since the ACL2 simplifier is not a function in the ACL2 logic, metafunctions
can only call upon it heuristically [HKK+05]. That is, even if ACL2 can
rewrite $ term $ to $ term' $, we cannot assume $ term = term' $ when we try to prove
the soundness theorem.

ACL2’s proof search is controlled by a large amount of unverified code,
and it is difficult to imagine “lifting” any substantial part of this into meta-
functions. Many features, such as linear arithmetic, are deeply integrated
into the rewriter [BM88], keep state across invocations, or do not fit into the
metafunction paradigm of replacing one term with an equal one (e.g.,
generalization). We would also face a bootstrapping problem: even if we could
cleanly extract a proof technique like type reasoning into a metafunction,
could we prove the soundness theorem without using type reasoning? We do
not see much hope of moving in this direction.

Metatheoretic extensibility is a challenge for HOL systems, where to add
a new proof procedure “we must somehow rip open an abstract type, tinker
with it to add a new constructor, and then close it up again.” [Har95]

Slind [Sli92] proposed a scheme for allowing \texttt{mk_thm}, an “arbitrary” \texttt{thm}
constructor that does not correspond to any rule of inference, to be used
under restricted circumstances. First, the semantics of ML would be for-
malized in HOL, as would the HOL implementation. Then, \texttt{mk_thm t} is to
be permitted only if we can prove there is some function \texttt{f} that produces a
usual, fully-expansive HOL proof of \texttt{t}. This idea was never implemented.³

More recently Chaieb and Nipkow [CN05] have written and verified a
quantifier-elimination procedure for Presburger arithmetic in Isabelle/HOL.
They encode Presburger formulas with a new type, and define their own
\textit{meaning} function to map encoded formulas into Booleans (the formulas of
HOL). A metafunction-like elimination procedure is implemented in a subset
of HOL which can be compiled to ML using a HOL compiler [BN02]. Finally,
a new, experimental rule of inference is added to the system so executions
of the ML program are allowed to be treated as proofs of equality. This
system is apparently 200 times faster for solving Presburger formulas than
an equivalent, tactic-based solution. This technique avoids the burden of
formalizing an ML system and a HOL implementation as Slind proposed, but
the code for constructing \texttt{thm} objects remains separated from the logic, and
as a result we still cannot reason about \texttt{thm} construction and the provability
of formulas.

Metafunctions are also supported in Coq. Grégoire and Mahboubi [GM05]
have introduced a procedure for reasoning about equality between polyno-
mials in commutative rings. They define a new type to represent encoded
polynomials over a ring and provide a \textit{meaning} function as above. They
show a metafunction-like canonicalization routine preserves the meaning of
encoded terms, and their procedure can then be used in proofs via Coq’s
evaluation/reduction facilities. As in Chaieb and Nipkow’s work, no method
is available for reasoning about the rules of inference and provability of for-

³Correspondence with Konrad Slind, August 2006.
mulas in general.

We are not aware of any support for metareasoning in PVS.

Knoblock and Constable [KC86] proposed two strategies for adding metareasoning to Nuprl. One approach involved a hierarchy of languages, where each $PRL^{n+1}$ would include an encoding of the $PRL^n$ proofs. In the other, a stack of languages would not be needed, and instead part of $PRL^1$ would be directly encoded into $PRL^0$. These ideas were never implemented.\footnote{Correspondance with Robert Constable, August 2006.}
References


[BM88] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear


[CN05] Amine Chaieb and Tobias Nipkow. Verifying and reflecting quantifier elimination for Presburger arithmetic. In *Logic


37


[TPG95] Cornell University The PRL Group. Implementing mathematics with the Nuprl proof development system, October 1995.

