The Dissertation Committee for Jared Curran Davis
certifies that this is the approved version of the following dissertation:

A Self-Verifying Theorem Prover

Committee:

J Strother Moore, Supervisor

E Allen Emerson

John Harrison

Warren A. Hunt, Jr.

Matt Kaufmann

Vladimir Lifschitz

A Self-Verifying Theorem Prover

by

Jared Curran Davis, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2009

For my wonderful wife—

It’s done!

Acknowledgments

I thank all of my committee members, Allen Emerson, John Harrison, Warren Hunt, Matt Kaufmann, Vladimir Lifschitz, and J Moore for their time, expertise, advice, suggestions, and encouragement. This work would not have been started, let alone finished, without them.

I thank Dave Greve, Robert Krug, Sandip Ray, Erik Reeber, and Eric Smith, each of whom has played a significant role in my learning to use ACL2. I also thank Andrew Gacek, John Matthews, and Bill Young for their special interest in this project and for useful feedback.

I thank the many people I have worked with at Rockwell Collins and Centaur Technology, including Bob Boyer, Dave Greve, David Hardin, Warren Hunt, Dan Luu, Terry Parks, Ray Richards, Al Sato, Anna Slobodova, Eric Smith, Sudarshan Srinivasan, and Sol Swords. Working with these people has been a wonderful experience, and I especially thank Glenn Henry and Matt Wilding for bringing these people together.

I thank the many other people who have participated at our weekly ACL2 seminar during my time at UT, including William Cook, John Erickson, Fares Fraij, Ian Johnson, Hanbing Liu, Serita Nelesen, Grant Passmore, David Rager, Mark Reitblatt, Julien Schmaltz, Peter Seidel, Alex Spiridonov, Matyas Sustik, Ian Wehrman, Chad Wellington, Bill Young, and Qiang Zhang. This group has been a great source of ideas, feedback, and technical support.

I thank our systems administrators, particularly Amy Bush, John Chambers, Fletcher Mattox, and Jason Pepas, for always going out of their way to provide me with access to reliable and powerful computing resources.

I thank the National Science Foundation (NSF) and the Defense Advanced Research Projects Agency (DARPA) for their support: this material is based upon work supported by DARPA and the NSF under grant number CNS-0429591, and by DARPA under contract NBCH30390004. During this project, I have made heavy use of the Mastodon cluster at the University of Texas at Austin, which was made possible, in part, by the NSF under CISE research infrastructure grant EIA-0303609.

Finally, I thank my family and friends for their encouragement and support, and most of all Laura, my wife, for her love and patience.

A Self-Verifying Theorem Prover

Publication No. ________________

Jared Curran Davis, Ph.D.
The University of Texas at Austin, 2009

Supervisor: J Strother Moore

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 create and check them with theorem-proving software. This software must be advanced enough to make the proof process tractable, but this very sophistication casts doubt upon the whole enterprise: who verifies the verifier?

We begin with a simple proof checker, Level 1, that only accepts proofs composed of the most primitive steps, like Instantiation and Cut. This program is so straightforward the ordinary, social process can establish its soundness and the consistency of the logical theory it implements (so we know theorems are "always true").

Next, we develop a series of increasingly capable proof checkers, Level 2, Level 3, etc. Each new proof checker accepts new kinds of proof steps which were not accepted in the previous levels. By taking advantage of these new proof steps, higher-level proofs can be written more concisely than lower-level proofs, and can take less time to construct and check. Our highest-level proof checker, Level 11, can be thought of as a simplified version of the ACL2 or NQTHM theorem provers. One contribution of this work is to show how such systems can be verified.

To establish that the Level 11 proof checker can be trusted, we first use it, without trusting it, to prove the *fidelity* of every Level n to Level 1: whenever Level n accepts a proof of some ϕ, there exists a Level 1 proof of ϕ. We then mechanically translate the Level 11 proof for each Level n into a Level $n-1$ proof—that is, we create a Level 1 proof of Level 2's fidelity, a Level 2 proof of Level 3's fidelity, and so on. This layering shows that each level can be trusted, and allows us to manage the sizes of these proofs.

In this way, our system proves its own fidelity, and trusting Level 11 only requires us to trust Level 1.

Table of Contents

Chapter 1

Introduction

Programming is an "exact science" in that our understanding of what a program does follows from the pure, abstract semantics of the language in which it is written. [43] Formal verification is the use of mathematical proof to show programs have desirable properties with respect to these semantics; for instance, we might prove a sorting program returns an ordered permutation of its input. The ability to show a program produces correct results for all possible inputs separates formal verification from testing, code reviews, and other software-engineering approaches.

There is no way to guarantee an actual computer will obey the semantics of its instruction set as it runs. Computers often suffer from design and manufacturing defects, and even the most well-built machine will misbehave when subjected to power surges, magnetic interference, physical tampering, and ordinary wear and tear. Formal verification, then, is not a tool for making claims about the reliability of systems in the real world, but is useful only for establishing a necessary precondition, viz. the program is properly written. [26]

Before formal verification can begin, we need to develop a precise, mathematical explanation of the semantics of the programming language. This can be a daunting task if the language is complicated, but basic approaches such as operational and denotational semantics are well known. After a mathematical model of the program has been developed, we can attempt to prove it behaves as desired.

What is a mathematical proof? Usually it is a concise document, written in a mixture of symbolic notations, diagrams, and English or another language, that carefully explains why a formula is a theorem. Confidence in such proofs is obtained through a social process: the proof is reviewed by editors, published, presented at conferences, and eventually read by the greater mathematical community. Once many experts have examined a proof without finding any flaws, its veracity becomes increasingly certain. [24]

Unfortunately, there is little hope that this approach can be used to check proofs about interesting programs. Program proofs are excessively large and—well, boring. This makes it difficult to gather a critical mass of interested mathematicians, and hard to believe their examination has been truly rigorous. Also, the social process is slow. This may be appropriate in traditional mathematics where the basic concepts have been fixed for decades, but it is not acceptable for programs which are being frequently changed to meet new requirements. [24]

A different kind of proof comes from mathematical logic. A formal proof is written as a long list of symbolic formulas. Each of these formulas must either be an axiom or must follow from previous formulas using a simple rule of inference. Mathematicians find formal proofs to be longer, more difficult to write, and more tedious to check than ordinary proofs. On the other hand, this rigid format makes checking each individual step of a formal proof quite easy, which allows proof checking to be automated with computer programs.

Writing formal proofs can also be automated to some degree. For certain limited classes of problems, such as propositional tautologies and basic arithmetic, fully automatic decision procedures are known. For more difficult problems, interactive approaches that make use of human guidance are usually necessary.

One style of interaction is called The Method in Boyer-Moore theorem provers such as NQTHM [17] and ACL2 [50]. When a user begins working to prove a new theorem, he gives a little guidance—“induct *this* way”, or “consider *these* cases”—then turns the problem over to a rewriter. The rewriter uses libraries of reusable rules, each of which has been proven earlier, to simplify the resulting cases. Some cases will be proven outright, and he can inspect the rest to decide which additional rules or hints are needed. Over time, his library of rules becomes a potent strategy for reasoning in his problem domain.

Using programs to write and check formal proofs is a good fit for formal verification. Tedium is no problem for computers, and once the proofs have been discovered, computerized checking is typically much faster than the social process. Also, when we guide the prover mainly through indirect advice, e.g., “use *these* lemmas” or “use *this* domain-specific solver,” it can often discover updated proofs after the program we are verifying has been changed. [60]

1.1 The Dissertation

Can computer-checked proofs be trusted? With three caveats, yes.

First, we should guard against the possibility that the computer used to run our program will make a mistake. While there is no way to guarantee no errors have occurred, the chance can be lessened by double- or triple-checking proofs with many computers. Ideally, the computers used should be manufactured at different facilities and based upon different designs to lessen the chance that they share some equivalent flaw. [1]

Second, we need to have confidence in the mathematical logic which is being implemented. Roughly speaking, the axioms must be true and the rules of inference must be truth-preserving, so that anything which is provable is true. Social proof is an appropriate mechanism for establishing such results, and introductory courses on mathematical logic usually cover how such proofs may be carried out.

Finally, we must ensure the proof-checking program is written correctly, i.e., it only accepts valid proofs. Unfortunately, the decision procedures, rewriters, and special-purpose programs which are used by theorem-proving software bear little resemblance to the simple rules of inference from mathematical logics. In theorem provers like ACL2 [50] and PVS [70], this disconnect can sometimes lead to "proofs" of non-theorems.

A well-known approach to reconciling this difference is to write these algorithms in a fully expansive manner. For instance, whereas an ordinary tautology-checking program might simply say "yes, ϕ is a tautology," a fully expansive program would additionally produce a formal proof of ϕ. Constructing fully expansive proofs sometimes incurs a considerable efficiency penalty. [23] On the other hand, when the LCF approach [31] is followed, the overhead of producing proofs is often not prohibitive [39], and today there are several fully expansive provers available, including Isabelle/HOL [67], HOL [33], and HOL Light [40].

This dissertation explores a different approach. Can we establish, in advance, that a useful, automated theorem prover can be trusted?

Our theorem prover is named Milawa, and it is probably best regarded as an "academic strength" imitation of the "industrial strength" ACL2. Many features of ACL2's reasoning engine are not reimplemented in Milawa, including its primitive type-reasoning, arithmetic procedure [47], meta rules [46], generalized equivalence relations [21], functional instantiation [15], and support for external tools [55]. On the other hand, Milawa's rewriter has many capabilities and heuristics which are similar to ACL2's, e.g., it can use rewrite rules which have free variables, forcibly assume hypotheses, etc., and overall, The Method is the same.

The Milawa logic is a simple, first-order logic of total, recursive functions with induction, styled after the logic of ACL2 [53]. Like ACL2's logic, Milawa's logic is "computational" and resembles functional programming, so it is straightforward to run Milawa-logic functions as Common Lisp programs. To allow the reader to have confidence in our logic, in Chapter 2 we present a rigorous description of its formulas and rules, and a social proof explaining why these rules are sound.

How might we establish that theorem prover obeys the rules of its logic? First, we would need a formal definition of provability for the logic being implemented, and confidence that this formalization is correct. Second, we would need a convincing and accurate mathematical model of the program's behavior. Finally, we would like to connect these two concepts with a believable proof of the following property, which we call the *fidelity* of the theorem prover: every formula accepted by the theorem prover is provable in the logic.

Since theorem provers are complex programs, it is difficult to trust a social proof of their behavior. It would also not be very convincing to have a prover verify its own fidelity, since this would be like asking someone "Do you ever lie?" So ideally, this proof should be carried out using some other automated proof system which is so simple that the ordinary, social process can show it is trustworthy.

In this dissertation, we formalize provability by introducing a function in the Milawa logic, named `proofp`, that determines whether an object represents a valid proof. We say a formula ϕ is provable when `proofp` accepts some proof of it. To make this formalization as convincing as possible, we write `proofp` very simply, and it intentionally bears a strong resemblance to the logical definition of proof. The development of `proofp` is covered in Chapter 3.

We also write our theorem prover in the Milawa logic. The advantage of this approach is that the Common Lisp program and its mathematical model are very closely related, which makes it easier to believe the model is accurate. We make an "academic" choice to keep this connection as simple as possible, even though it means putting up with some efficiency limitations, e.g. we must always use arbitrary-precision arithmetic, cannot perform destructive updates, have no arrays or hash tables, and lack parallelism. A more "industrial" approach would do away with these limitations by adding features such as the guards, single-threaded objects, hash cons, fast association lists, and parallelism primitives which are available [73, 37, 16] in the ACL2 system. Such features allow for more efficient execution, but complicate the connection with the programming language.

Since we have modeled both provability and the theorem prover in the Milawa logic, it is convenient to carry out the fidelity proof in this same logic. It does not take much additional infrastructure to write a program around `proofp`, in Common Lisp, which allows its user to define new functions and check proofs of theorems. This program provides no automation for finding proofs, but it is simple enough to be validated by the social process. We explain its implementation in Chapter 4.

Finally, we develop a proof which shows that our theorem prover only accepts formulas that satisfy our definition of provability. It takes some work to develop such a proof, and the rest of this introduction explains our approach. After the proof has been constructed, we check it with our simple proof-checking program, on a variety of computers, to ensure its validity.

In the end, we can freely use the Milawa theorem prover and have confidence that whenever it claims to have proven a formula, that formula is indeed true.

1.2 Planning the Proof

Our theorem prover is a complicated program, so it is challenging to prove properties about its behavior. Meanwhile, it is practically difficult to write `proofp`-style proofs because they are excessively large and repetitive. How, then, can we construct a `proofp`-style proof of our program's fidelity?

We begin by using ACL2 to develop a proof plan. ACL2 is normally thought of as a trustworthy tool, but here we are using it only informally as a familiar, mature environment in which to "sketch" how the proof can be completed. This planning process is useful because it separates the intellectual task of discovering why the statement is true from the engineering task of constructing and checking such a large formal proof. Because the ACL2 logic is so similar to the Milawa logic, it is straightforward to model Milawa in ACL2.

Milawa employs a number of proof techniques. To establish the fidelity of the whole of Milawa, we must show that any claim made by these techniques can be justified using the rules of the Milawa logic. As an example, Milawa may use "evaluation" to reduce ground terms to constants, e.g., given $\mathit{fact}(5)$, evaluation will produce 120. The claim being made is that the formula, "$\mathit{fact}(5) = 120$," is provable.

In our proof plan, our basic approach to verifying these proof techniques is as follows. First, we write a fully expansive version of the technique. Then, we use ACL2 to show that for all sensible inputs, (1) the fully expansive version produces a valid `proofp`-style proof, and (2) this proof has "the right conclusion."

In the case of evaluation, we begin by developing a fully expansive evaluator. Whereas our ordinary evaluator will produce 120 when given $\mathit{fact}(5)$, this new function constructs a `proofp`-style proof that concludes $\mathit{fact}(5) = 120$. We then use ACL2 to show (1) when the definitions used to evaluate some term, x, are valid, then the fully expansive evaluator produces a valid proof, and (2) this proof concludes $x = x'$, where x' is the result of evaluating x with our ordinary evaluator.

In our ACL2 proof plan, the fully expansive evaluator provides a constructive method of justifying any claim made by our evaluator. If our goal was only to provide an ACL2 proof of Milawa's fidelity, there would be no need to ever run this function. But, as we will see in the next section, the fully expansive evaluator (and the fully expansive versions of our other techniques) are also useful in converting our ACL2 proof sketch into a `proofp`-style proof.

After accounting for all of Milawa's proof techniques in this manner, we arrive at a fairly large ACL2 proof. What does the proof look like? Having followed The Method, it is a sequence of "events" involving roughly 2,700 definitions and 11,600 lemmas. The main steps in this sequence are as follows.

1. A utility library with the most basic functions about arithmetic, lists, and so on is introduced. This involves 386 definitions and 1,426 lemmas. We do not cover this in any depth since it is such typical ACL2 work.

2. Concepts from the Milawa logic are introduced, such as the encoding of terms, formulas, and proofs. Along the way, typical ACL2 lemmas are introduced for reasoning about these concepts. Altogether there are 242 functions, including `proofp`, and 1,980 lemmas.

3. Low-level functions for building `proofp`-style, fully expansive proofs are then developed. This includes functions for performing basic propositional manipulation, substituting into equalities, and so forth. For each function, ACL2 lemmas establish that, given valid inputs, a valid proof with the expected conclusion will result. Altogether, this involves 657 functions and 1,555 lemmas. This work is primarily addressed in Chapters 5 and 6.

4. Clauses, which provide the foundation for proof search, are introduced, along with Milawa's techniques for simplifying, updating, and case-splitting clauses. Fully expansive versions of these techniques are developed and shown to produce valid proofs. This involves another 289 definitions and 1,482 lemmas, and is covered in Chapter 7.

5. Milawa's assumptions system, which assists the rewriter by managing tables of equalities and Boolean equivalences, is then developed. In the fully expansive version of the assumptions system, "traces" are recorded as inferences are made, and these traces can later be compiled into fully expansive proofs to justify each inference. Altogether this takes 164 definitions and 908 lemmas, and is covered in Chapter 8.

6. The Milawa rewriter is then introduced. Rewriting is the main component of Milawa's proof strategy. To simplify goals, the rewriter makes use of assumptions, calculation, and user-supplied rules which are organized into "theories." The fully expansive version of the rewriter records traces which can later be compiled into proofs. In all, 699 functions and 3,143 lemmas are introduced. We discuss the rewriter in Chapter 9.

7. Other proof techniques are introduced and justified. These include "hints" which allow the user to generalize terms, use equalities in certain ways, consider particular instances of previous theorems, and so on. We then bundle these techniques together with the rewriter, case splitting algorithm, and so on, to form a tactic system for carrying out proofs. This involves 305 definitions and 1,120 lemmas, and is discussed in Chapter 10.

The ACL2 proof of Milawa's fidelity may not, by itself, be entirely convincing. ACL2 is a complex computer program which has not undergone a rigorous, mechanical verification, and has a much larger "trusted core" than fully expansive provers such as HOL Light.

1.3 Self-Verification

During the course of the ACL2 proof, fully expansive versions of each of Milawa's proof techniques were developed. Because of this, it is not difficult to assemble a fully expansive version of Milawa which can emit `proofp`-style proofs of the theorems it claims to have proven.

Since (1) Milawa is quite similar to ACL2 in terms of its logic and the basic proof strategy it implements, and (2) an ACL2 proof of Milawa's fidelity has been developed, the following becomes a possible strategy for constructing a `proofp`-style proof of Milawa's fidelity: first, redo the ACL2 proof in Milawa, then have it emit a `proofp`-checkable justification of its work.

Translating the ACL2 proof into Milawa took some work. As an important step in this effort, we developed a user interface within ACL2 for finding and building Milawa proofs. This interface allows us to introduce the Milawa counterparts to ACL2 functions and theorems quite easily, and keeps our ACL2 proof sketch in sync with the actual Milawa proof. We present an overview of this interface in Chapter 11.

Although the two provers are largely similar, there are a number of differences in the specifics of their implementations. This gap was bridged from both sides. First, as the ACL2 proof was being developed, a conscious effort was made to avoid using "complicated" features of ACL2 which would be difficult to implement in Milawa. Some examples of this are as follows.

1. ACL2's elaborate type-reasoning system could not be entirely disabled, but it was possible to weaken it in many ways. For one, ACL2 automatically infers

certain type information when new definitions are submitted, so the prover was instructed not to use these rules. Certain built-in type-reasoning rules were also disabled, and no type-reasoning lemmas were ever added. Finally, where possible, new aliases were used to hide primitive functions such as `car` and `+` to prevent special, deeply built-in type-reasoning about these functions.

2. It was not possible to disable ACL2's arithmetic procedure, but most arithmetic reasoning was avoided. Aliases were used for functions such as `<`, `+`, and `natp`, so the arithmetic procedure would not see functions it knew about when it inspected clauses. Also, no arithmetic lemmas were ever added.

3. ACL2 was instructed not to "generalize" terms or "cross-fertilize" equalities automatically during proofs. It is usually straightforward to introduce lemmas to avoid needing these techniques.

4. Forward-chaining rules were completely avoided, as were more advanced kinds of reasoning, such as custom equivalence relations, congruence rules, meta-rules, and external tools.

Second, when it became evident that avoiding some ACL2 feature would be too difficult, the feature was added to Milawa instead. This would require redoing some of the ACL2 proof, since Milawa itself had been changed. But since The Method had been followed, most proofs were robust in the face of such changes. Over time, several features were added to Milawa's rewriter, including ancestors checking, free-variable matching, and forcing, to make it powerful enough to carry out the proof. And in the end, the Milawa proof matches ACL2's almost lemma-for-lemma.

1.4 A Verified Stack

Unfortunately, it is not practical for the fully expansive version of Milawa to emit a `proofp`-style proof of the entire fidelity argument. Even despite some work to make the low-level proof-building routines more efficient, the proof is overwhelmingly large and its construction is well beyond the capacity of our computers. (A more detailed discussion of proof sizes and capacity limitations may be found in Chapter 12.)

Even so, the goal of verifying Milawa with `proofp` is still possible, via the less-direct approach of introducing and verifying a sequence of increasingly capable proof checkers. We use the word *levels* to describe this sequence. That is, we say that `proofp` is the Level 1 proof checker; the objects it accepts are Level 1 proofs and may use only Level 1 proof steps. These Level 1 steps correspond to the primitive rules of inference of the Milawa logic. At each new level in the sequence, we allow new kinds of proof steps to be used. For instance, given a proof of $(A \vee B) \vee C$, it follows that $A \vee (B \vee C)$. We call this *right associativity*. Level 2 expands upon Level 1 by accepting steps such as right associativity, in addition to all of the Level 1 steps.

Because of our previous work, it is relatively easy to develop an ACL2 proof of the fidelity of the Level 2 proof checker—that is, whenever a Level 2 proof of ϕ is accepted, there exists a Level 1 proof of ϕ. We redo this proof in Milawa, and use the fully expansive version of Milawa to emit a Level 1 proof that establishes the fidelity of the Level 2 proof checker. This proof is small enough to practically construct and check.

This is progress. By taking advantage of these new steps, Level 2 proofs can be written more concisely than Level 1 proofs. It takes eight Level 1 proof steps to carry out the work of a single right-associativity step, and 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. Yet, we have a Level 1 proof that establishes the fidelity of Level 2, so to trust Level 2 we only need to trust 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. We call this process bootstrapping, and cover it in Chapter 12.

In our most sophisticated proof checker, Level 11, a single proof step might involve the application of several tactics, which carry out tasks such as splitting a clause into subgoals, generalizing terms, using equalities, explicitly instantiating theorems, and rewriting terms while making use of assumptions, calculation, and user-supplied rules.

1.5 Organization of the Dissertation

Because this dissertation is quite long, we now provide a map which summarizes our organization. We encourage the reader to return to this map from time to time to recall the overall structure of the dissertation.

Part 1. The Trusted Core

We claim our theorem prover may be trusted because its logic is sound and we have proven it obeys the rules of this logic. More precisely, we have proven that whenever our program accepts some formula, ϕ, there exists a proof of ϕ that is accepted by our Level 1 proof checker. To agree with our claim, the reader must trust that our logic is sound, we have modeled our program accurately, and we have have properly formalized provability. Furthermore, for our fidelity proof to be trusted, the reader must understand how it is checked. In Part 1, we set out to address these concerns.

Chapter 2. The Logic. So the reader may have confidence that our logic is sound, we begin with a rigorous presentation of the syntax of our logic, and an enumeration of its axioms and rules of inference. We provide an informal, “social” proof that axioms are valid, and our rules of inference are validity-preserving.

Chapter 3. The Proof Checker. We describe how terms, formulas, and proofs may be encoded as objects in our logic, and introduce `proofp`, which serves as our formalization of provability. To gain confidence that `proofp` only accepts legitimate proofs, the reader should examine its implementation and compare it to the rules in Chapter 2.

Chapter 4. System Implementation. We develop a way to execute functions from our logic using a Common Lisp system. Since we introduce `proofp` as a function in our logic, this mechanism allows us to run `proofp` from within Common Lisp. We then implement, in Common Lisp, a primitive command-line program that allows a user to introduce a sequence of definitions and theorems for `proofp` to check. Notably, one of our program's commands allows us to install a new proof checker after verifying its fidelity. We introduce and verify our theorem prover by using these commands. The reader should examine both the connection with Common Lisp and how these commands are processed to understand how the theorem prover is modeled and how its proof is checked.

Part 2. Building Proofs

In Part 2, we explain how derived rules of inference can be implemented as functions that construct fully expansive proofs, and how, in our ACL2 proof plan, we can reason about the proofs constructed by these functions. (Later, in Part 4, we explain how this proof plan can be converted into a form that can be checked by our program from Part 1.) These derived rules become “subroutines” of our fully expansive proof techniques, and play a crucial role in allowing us to describe and reason about proofs at higher levels of abstraction.

Chapter 5. Propositional Calculus. We explain our basic approach to implementing derived rules as proof-building functions, and the details of how we reason about these functions. We then develop a number of rules to make propositional reasoning more convenient, ranging from rules as simple as Modus Ponens to inductively derived rules that can prove any propositional tautology.

Chapter 6. Equality. We begin to look beyond propositional reasoning. We begin with simple derived rules for reasoning about equalities, e.g., the reflexivity and transitivity of equality, and explain how we can write and reason about derived rules which require the availability of certain axioms and theorems. We develop rules for carrying out "deep" equality substitutions, and implement a McCarthy-style evaluator for ground terms as a derived rule.

Part 3. Theorem Proving

In Part 3, we introduce our theorem prover. We style the prover after ACL2. It can carry out a backward (goal-directed) proof search, making use of case-splitting, rewriting, and and other techniques such as induction, generalization, and destructor elimination. As we introduce each algorithm, we explain how we can justify its use.

Chapter 7. Clauses. Clauses form the basis for our backward proof search: we represent each goal to be proven as a clause, then apply reversible reductions to the goal to obtain simpler goals. We develop some basic ways to manipulate and simplify clauses, and introduce our if-lifting and case-splitting algorithms, which can be used to break a complex goal into simpler subgoals.

Chapter 8. Assumptions. When rewriting a goal such as $A \vee B$, we may assume A is false as we rewrite B. Our rewriter makes use of an assumptions system which records what has been assumed and can canonicalize terms which are known to be equivalent.

Chapter 9. Rewriting. Rewriting with lemmas is the main tool in our style of theorem proving. Our rewriter can simplify clauses using assumptions, calculation, and conditional rewrite rules whose hypotheses are relieved by backchaining (recursive rewriting). It implements many features from ACL2's rewriter, such as ancestors checking and forcing.

Chapter 10. Tactics. We implement a tactic system which can be used to compose our clause reductions and manage a backward proof. We provide tactics for routines like case splitting and rewriting from previous chapters, and also tactics for other techniques such as induction, generalization, and destructor elimination. Together, these tactics form our theorem prover.

Part 4. Self-Verification

Taken together, Parts 2 and 3 introduce our theorem prover and explain how it is verified in ACL2. But our goal is to instead carry out the fidelity proof using the program from Part 1. Our strategy is to first recreate the ACL2 proof plan using Milawa, then use the fully expansive versions of our proof techniques to emit proofs which can be checked by our program.

Chapter 11. User Interface. We develop a user interface for applying the tactics from Milawa. The user interface is integrated into ACL2. It allows us to easily introduce the Milawa counterparts of ACL2 functions and theorems. It can also use the fully expansive versions of our proof techniques to emit proofs for our program from Part 1 to check. This interface is only a tool for using Milawa, is not verified, and need not be trusted.

Chapter 12. Bootstrapping. Using the interface, we direct Milawa to carry out the proofs from our ACL2 proof plan. Once Milawa has proved its own fidelity, we emit proofs for our program to check. To manage the size of these proofs, we introduce and verify a number of intermediate proof checking levels. Finally, we run our program on a number of computers and Lisp systems to check the proofs. As a result, only the program in Part 1 must be trusted.

Part I

The Trusted Core

Chapter 2

The Logic

A prerequisite to writing a theorem prover or proof checker is to decide upon a mathematical logic to use. Modern theorem provers do not agree on any standard, and this choice is “a matter of taste and experience” [57] which may be viewed “eclectically and pragmatically.” [60, ch. 8]

The Milawa logic is a simpler variant of the ACL2 logic [50, 53]. Our objects are the symbols and naturals, recursively closed under ordered pairing. By comparison, ACL2 additionally includes characters, strings, and non-natural integers, rationals, and complex rationals. The removal of these types is intended to make the object system as simple as practically possible, and does not reduce the expressivity of the logic. For instance, one might represent characters with their ASCII codes, strings with lists of characters, integers with sign/magnitude pairs, rationals with numerator/denominator pairs, and complex rationals with pairs of rationals.

To make Common Lisp execution of our logical functions as simple as possible, we do away with packages, guards, single-threaded objects, and so on. We associate a primitive constant with each of our objects, and use a new rule, base evaluation, to explain the behavior of our primitive functions on constants. This rule is similar to the way in which McCarthy’s [63] Lisp interpreter uses special cases to evaluate “elementary S-functions” like cons.

Our logic is first-order, lacks explicit quantifiers, and has equality as its only predicate symbol. We directly adopt Shoenfield’s rules of propositional calculus [83], which are usually used in descriptions of the ACL2 logic, and we keep ACL2's instantiation rule. We adopt an induction rule similar to that of ACL2 and NQTHM [12]. Also like ACL2, we permit the introduction of total, untyped, recursive functions, and the introduction of Skolem (quantifier witnessing) functions.

The lack of static typing, quantifiers, and higher-order functions makes our logic fairly restrictive. Despite similar restrictions, Boyer-Moore provers have been successfully used in the verification of hardware modules [77, 79, 48, 49], processor models [45, 20, 65, 36, 80], machine- and byte-code programs [14, 59], operating systems [8], virtual machines [19, 58], compilers [94, 6, 29], and other algorithms [74, 89, 76, 66].

There are some advantages to using a simple logic. For example, term quotation and reflection are more straightforward when terms are untyped and term equality does not rely on reductions [3, 44]. Also, because our terms are so simple, we do not need a type checker, type inference engine, or much in the way of interfacing layers like parsers and term rendering. All together, this helps to keep the source code for our proof checker small, which is important since our trust in the proof checker is to rely upon the social process.

Finally, using a simple logic seems particularly appropriate. The techniques we develop should be adaptable to more powerful logics without trouble, whereas if we were to begin with a powerful logic, we might rely upon features that are not available in weaker logics. It may even be easier to follow our approach to verify theorem provers in more expressive logics, where more powerful rules of inference would be available, and hence fewer intermediate proof-checkers might be necessary.

In this chapter, we provide a rigorous introduction to our logic and an explanation of why its rules may be trusted. Little here is novel, and our goal is only to lay a proper foundation for later chapters.

2.1 Formulas

The first step in presenting a mathematical logic is to describe rigorously the syntax of its *formulas*. Like statements in a programming language, our formulas are built from more primitive elements of syntax. The smallest units of syntax are called *tokens*. We have a *numeric token* corresponding to every natural number and a *symbolic token* corresponding to every string of ASCII characters.

We will use the `typewriter font` to write fragments of syntax. For compatibility with the Common Lisp reader, we may write down a particular numeric token in many ways. For instance, the token for sixty-four may be written in decimal as `64`, in hexadecimal as `#x40`, in octal as `#o100`, or in binary as `#b1000000`. We consider these variations not to be different tokens, but only as different ways to write the same token.

To write down a symbolic token, we typically use vertical-bar characters to denote the beginning and end of the token. We also adopt an escape convention so that a backslash in front of any character just represents that character. As with numeric tokens, we consider variants such as `|f\o\o|` and `|foo|` as merely different ways of writing the same token.

This use of vertical bars allows us to distinguish symbolic tokens from other syntactic elements. For instance `123` is a numeric token while `|123|` is symbolic. But these bars are often unnecessary. When a symbolic token cannot be confused for a number, is entirely in upper-case, and does not include various problematic characters (such as non-printable ASCII characters, parentheses, commas, colons, spaces, quotes, and so on, as in Common Lisp), then we may drop the vertical bars and write it in a case-insensitive manner. Here are some examples.

Notation	Corresponding String
`|Hello, World!|`	Hello, World!
`|a\\|b|`	a|b
`|C:\\Windows|`	C:\Windows
`||`	*the empty string*
`|f\o\o|`	foo
`append`	APPEND
`Int32`	INT32
`math.square`	MATH.SQUARE
`3+x`	3+X

The rest of our syntax is based on *token trees*, which we define recursively as follows: every token is a token tree, and whenever a and b are token trees, so too is the ordered pair of a and b, which we write as `(`a `.` b`)`. Just as there are many ways to write down the same number or symbol, we will adopt some notational conveniences from Lisp for writing down token trees.

Abbreviation	Meaning
`()`	`nil`
`(`x`)`	`(`x `. nil)`
`(`x_1 x_2 $\ldots$ x_n`)`	`(`x_1 `. (`x_2 $\ldots$ x_n`))`
`(`x_1 x_2 $\ldots$ x_n `.` b`)`	`(`x_1 `. (`x_2 $\ldots$ x_n `.` b`))`
`'`x	`(quote `x`)`

Whenever x is a token tree, we say `'`x is a *constant*. The *variables* are any symbolic tokens except for `t` and `nil`. The *function names* are any symbolic tokens except for:

`nil`	`pequal*`	`first`	`and`
`quote`	`pnot*`	`second`	`or`
	`por*`	`third`	`list`
		`fourth`	`cond`
		`fifth`	`let`
			`let*`

We simultaneously define the *terms* and the *free variables* of a term—which we denote as FREEVARS(t)—as follows. Every constant is a term with no free variables.

Every variable, v, is a term whose only free variable is v, itself. A *function application*, `(`$f\ \ t_1\ \ \ldots\ \ t_n$`)`, is a term when f is a function name and $t_1, \ldots, t_n$ are terms; its free variables are $\bigcup_{i=1\ldots n} \textsc{freevars}(t_i)$. This definition does not include any notion of arity checking, but we address well-formedness in the next section. A *lambda abbreviation*,

$$\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1\ \ldots\ t_n\texttt{)},$$

is a term when $x_1, \ldots, x_n$ are distinct variables, called the *formals*, β is a term, called the *body*, $t_1, \ldots, t_n$ are terms, called the *actuals*, and $\textsc{freevars}(\beta) \subseteq \{x_1, \ldots, x_n\}$; its free variables are $\bigcup_{i=1\ldots n} \textsc{freevars}(t_i)$.

We define the *formulas* as follows. `(pequal*` $t_1\ \ t_2$`)` is a formula when t_1, t_2 are terms, `(pnot*` F`)` is a formula when F is a formula, and `(por*` $F\ \ G$`)` is a formula when F and G are formulas. Since `pequal*`, `pnot*`, and `por*` are not function names, there is no confusion between terms and formulas.

2.2 Validity and Proof

A key idea of mathematical logic is that a notion of truth can be assigned to formulas, and formulas can be manipulated in ways which are truth-preserving. We now describe how this is done for our formulas.

An *arity table* is a mapping from a set of function names to natural numbers ("arities"). We say a term, t, is well-formed with respect to an arity table, *atbl*, if for every function application within t, `(`$f\ \ t_1\ \ \ldots\ \ t_n$`)`, *atbl* binds f to n. Likewise, a formula is well-formed with respect to an arity table when all terms within it are well-formed. Throughout this discussion, we assume that we have a fixed arity table, *atbl*, and we are working with terms and formulas that are well-formed with respect to it.

A *first-order structure*, $\mathcal{A}$, establishes the meaning of constants and function applications used within formulas. It consists of a set of mathematical objects called its *universe*, $|\mathcal{A}|$; an association for every constant, c, to some element of this universe, $c^{\mathcal{A}}$; and an association for every function name, f, of arity n, to a total, n-ary function, $f^{\mathcal{A}} : |\mathcal{A}|^n \to |\mathcal{A}|$.

Given a first-order structure, $\mathcal{A}$, an *interpretation*, I, is a function that maps every variable, v, to some object, $I(v)$, in $|\mathcal{A}|$. We extend the notion of interpretations to terms as follows. The interpretation of variables is already established, and the interpretation of constants is inherited from the first-order structure—that is, $I(c) = c^{\mathcal{A}}$. For function applications,

$$I(\texttt{(}f\ \ t_1\ \ \ldots\ \ t_n\texttt{)}) = f^{\mathcal{A}}(I(t_1), \ldots, I(t_n)).$$

Finally, for lambda abbreviations,

$$I(\texttt{((lambda (}x_1\ \ \ldots\ \ x_n\texttt{)}\ \ \beta\texttt{)}\ \ t_1\ \ \ldots\ \ t_n\texttt{)}) = I'(\beta),$$

where I' is the interpretation defined as

$$I'(v) = \begin{cases} I(t_i) & \text{if } v = x_i \text{ for some } i \\ I(v) & \text{otherwise.} \end{cases}$$

In this way, an interpretation allows us to map any term to an object in the universe, $|\mathcal{A}|$.

We also extend interpretations to assign truth values—true or false, which may as well be regarded as existing independently from $|\mathcal{A}|$—to formulas,

$$\begin{aligned} I(\texttt{(pequal*}\ \ t_1\ \ t_2\texttt{)}) &\leftrightarrow I(t_1) = I(t_2) \\ I(\texttt{(pnot*}\ \ F\texttt{)}) &\leftrightarrow \neg I(F) \\ I(\texttt{(por*}\ \ F\ \ G\texttt{)}) &\leftrightarrow I(F) \vee I(G), \end{aligned}$$

and say a formula is *valid* when every interpretation assigns true to it. Since this notion of validity is implicitly related to a particular first-order structure, to be precise we may sometimes refer to the validity of a formula, F, *in* some first-order structure, $\mathcal{A}$.

A *proof* is a syntactic object which, beginning from *axioms*, arrives at some formula, called its conclusion, by applying *rules of inference*; the axioms are particular, distinguished formulas which we know are valid, and rules of inference are certain syntactic transformations which, given valid formulas to begin with, produce new formulas which are also valid. The conclusion of any proof is said to be a *provable* formula.

2.3 Basic Rules of Inference

We begin with some simple rules of inference. Each rule allows us to prove some particular formula, given proofs of some related formulas called the *premises* of the rule.

Associativity	From	`(por* A (por* B C))`
	Derive	`(por* (por* A B) C)`
Contraction	From	`(por* A A)`
	Derive	A
Cut	From	`(por* A B)` and `(por* (pnot* A) C)`
	Derive	`(por* B C)`
Expansion	From	B
	Derive	`(por* A B)`
Propositional Schema	From	*nothing*
	Derive	`(por* (pnot* A) A)`

It is straightforward to see that these rules are validity-preserving. We also introduce the rule of **Functional Equality**: given a function name, f, of arity n, and terms $t_1, \ldots, t_n$ and $s_1, \ldots, s_n$, we may derive

```
(por* (pnot* (pequal* $t_1$ $s_1$))
      (por* (pnot* (pequal* $t_2$ $s_2$))
            ⋱
              (por* (pnot* (pequal* $t_n$ $s_n$))
                    (pequal* ($f$ $t_1$ $t_2$ ... $t_n$)
                             ($f$ $s_1$ $s_2$ ... $s_n$))) ... )),
```

and again it is easy to see this rule only allows us to derive valid formulas.

Before introducing our next rules of inference, we need to define *substitution*. A substitution list, σ, is a mapping from some finite number of variables to terms. We write $\sigma = [v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n]$ when σ maps $v_1, \ldots, v_n$ to $t_1, \ldots, t_n$, respectively, and we say the substitution of σ into the variable v, written v/σ, is defined as follows:

$$v/\sigma = \begin{cases} t_i & \text{for the least } i \text{ s.t. } v = v_i, \text{ if one exists} \\ v & \text{otherwise.} \end{cases}$$

We extend the notion of substitution to arbitrary terms. If t is a variable then t/σ has already been defined, and if t is a constant then $t/\sigma = t$. For function applications, $\texttt{(}f\ t_1\ \ldots\ t_n\texttt{)}/\sigma = \texttt{(}f\ t_1/\sigma\ \ldots\ t_n/\sigma\texttt{)}$, and for lambda abbreviations,

$$\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1\ \ldots\ t_n\texttt{)}/\sigma =$$
$$\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1/\sigma\ \ldots\ t_n/\sigma\texttt{)}.$$

We also extend substitution to formulas, as follows:

$$\texttt{(pequal* }t_1\ t_2\texttt{)}/\sigma = \texttt{(pequal* }t_1/\sigma\ t_2/\sigma\texttt{)}$$
$$\texttt{(pnot* }F\texttt{)}/\sigma = \texttt{(pnot* }F/\sigma\texttt{)}$$
$$\texttt{(por* }F\ G\texttt{)}/\sigma = \texttt{(por* }F/\sigma\ G/\sigma\texttt{)}.$$

Now that we have defined substitution, we introduce two additional rules of inference:

Instantiation	From	A
	Derive	A/σ
β-Reduction	From	*nothing*
	Derive	`(pequal* ((lambda (`x_1 ... x_n`)` β`)` t_1 ... t_n`)` $\beta/[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$`)`

To show these rules are validity-preserving, we will need a couple of lemmas about substitution. Our first lemma shows the interpretation of a term depends only upon the interpretation of its free variables.

Lemma 2.1. $\forall I, J : (\forall v \in \textsc{freevars}(t), I(v) = J(v)) \rightarrow I(t) = J(t)$

Proof by structural induction on t.

The constant and variable cases are trivial.

Suppose t is `(`f t_1 ... t_n`)`, let I, J be arbitrary interpretations, and assume $\forall v \in \textsc{freevars}(t), I(v) = J(v)$ so that our goal is to show $I(t) = J(t)$. For each i, we may inductively assume

$$(\forall v \in \textsc{freevars}(t_i), I(v) = J(v)) \rightarrow I(t_i) = J(t_i),$$

but since $\textsc{freevars}(t_i) \subseteq \textsc{freevars}(t)$, we have

$$\forall v \in \textsc{freevars}(t_i), I(v) = J(v),$$

and so we may conclude $I(t_i) = J(t_i)$. But now,

$$I(t) = f^{\mathcal{A}}(I(t_1), \ldots, I(t_n)) = f^{\mathcal{A}}(J(t_1), \ldots, J(t_n)) = J(t).$$

Finally, suppose t is `((lambda (`x_1 ... x_n`)` β`)` t_1 ... t_n`)`, let I, J be arbitrary interpretations, and assume $\forall v \in \textsc{freevars}(t), I(v) = J(v)$; our goal is to show

$I(t) = J(t)$, which is the same as showing $I'(\beta) = J'(\beta)$, where

$$I'(v) = \begin{cases} I(t_i) & \text{if } v = x_i \text{ for some } i \\ I(v) & \text{otherwise, and} \end{cases}$$
$$J'(v) = \begin{cases} J(t_i) & \text{if } v = x_i \text{ for some } i \\ J(v) & \text{otherwise.} \end{cases}$$

Now, for each i, we may inductively assume

$$(\forall v \in \text{FREEVARS}(t_i), I(v) = J(v)) \rightarrow I(t_i) = J(t_i),$$

and as before, since $\text{FREEVARS}(t)$ subsumes $\text{FREEVARS}(t_i)$, we may conclude $I(t_i) = J(t_i)$.

We may also inductively assume

$$(\forall v \in \text{FREEVARS}(\beta), I'(v) = J'(v)) \rightarrow I'(\beta) = J'(\beta),$$

and since $\text{FREEVARS}(\beta) \subseteq \{x_1, \ldots, x_n\}$, we arrive at $I'(x_i) = I(t_i) = J(t_i) = J'(x_i)$, so we have

$$\forall v \in \text{FREEVARS}(\beta), I'(v) = J'(v)$$

and we may conclude $I'(\beta) = J'(\beta)$, which was our goal. □

Our next lemma characterizes how a term's interpretation is changed by substitution. We introduce a new bit of notation: when I is an interpretation and σ is a substitution list, we write I_σ to denote the interpretation defined as follows: $I_\sigma(v) = I(v/\sigma)$.

Lemma 2.2. *$I_\sigma(t) = I(t/\sigma)$*

Proof by structural induction on t.

The constant and variable cases are trivial.

Suppose t is $\texttt{(}f\ t_1\ \ldots\ t_n\texttt{)}$. For each i we may inductively assume $I_\sigma(t_i) = I(t_i/\sigma)$. Then,

$$\begin{aligned}
I(t/\sigma) &= I(\texttt{(}f\ t_1\ \ldots\ t_n\texttt{)}/\sigma) \\
&= I(\texttt{(}f\ t_1/\sigma\ \ldots t_n/\sigma\texttt{)}) \\
&= f^{\mathcal{A}}(I(t_1/\sigma), \ldots, I(t_n/\sigma)) \\
&= f^{\mathcal{A}}(I_\sigma(t_1), \ldots, I_\sigma(t_n)) \\
&= I_\sigma(\texttt{(}f\ t_1\ \ldots\ t_n\texttt{)}).
\end{aligned}$$

Finally, suppose t is $\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1\ \ldots\ t_n\texttt{)}$. Then,

$$\begin{aligned}
I(t/\sigma) &= I(\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1\ \ldots\ t_n\texttt{)}/\sigma) \\
&= I(\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1/\sigma\ \ldots\ t_n/\sigma\texttt{)}) \\
&= I'(\beta),
\end{aligned}$$

where

$$I'(v) = \begin{cases} I(t_i/\sigma) & \text{if } v = x_i \text{ for some } i \\ I(v) & \text{otherwise.} \end{cases}$$

Meanwhile,

$$\begin{aligned}
I_\sigma(t) &= I_\sigma(\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1\ \ldots\ t_n\texttt{)}) \\
&= {I_\sigma}'(\beta),
\end{aligned}$$

where

$${I_\sigma}'(v) = \begin{cases} I_\sigma(t_i) & \text{if } v = x_i \text{ for some } i \\ I_\sigma(v) & \text{otherwise.} \end{cases}$$

Moreover, Lemma 2.1 will allow us to conclude $I'(\beta) = {I_\sigma}'(\beta)$ if we can show that $I'(v) = I'_\sigma(v)$ for all $v \in \textsc{freevars}(\beta)$. Since $\textsc{freevars}(\beta) \subseteq \{x_1, \ldots, x_n\}$, it suffices to show $I(t_i/\sigma) = I_\sigma(t_i)$. But this is something we may inductively assume. $\square$

We now extend Lemma 2.2 to explain how substitution affects the interpretation of a formula.

Lemma 2.3. *$I_\sigma(F) \leftrightarrow I(F/\sigma)$.*

Proof by structural induction on F.

If F is `(pequal*` t_1 t_2`)`, then

$$\begin{aligned} I(F/\sigma) &\leftrightarrow I(t_1/\sigma) = I(t_2/\sigma) \\ &\leftrightarrow I_\sigma(t_1) = I_\sigma(t_2) && \text{by Lemma 2.2} \\ &\leftrightarrow I_\sigma(F) \end{aligned}$$

If F is `(pnot*` G`)`, we may inductively assume $I(G/\sigma) \leftrightarrow I_\sigma(G)$, and

$$\begin{aligned} I(F/\sigma) &\leftrightarrow I(\texttt{(pnot*}\ G/\sigma\texttt{)}) \\ &\leftrightarrow \neg I(G/\sigma) \\ &\leftrightarrow \neg I_\sigma(G) \\ &\leftrightarrow I_\sigma(\texttt{(pnot*}\ G\texttt{)}) \\ &\leftrightarrow I_\sigma(F). \end{aligned}$$

If F is `(por*` G H`)`, we may inductively assume $I(G/\sigma) \leftrightarrow I_\sigma(G)$ and $I(H/\sigma) \leftrightarrow I_\sigma(H)$, and

$$\begin{aligned} I(F/\sigma) &\leftrightarrow I(\texttt{(por*}\ G/\sigma\ \ H/\sigma\texttt{)}) \\ &\leftrightarrow I(G/\sigma) \vee I(H/\sigma) \\ &\leftrightarrow I_\sigma(G) \vee I_\sigma(H) \\ &\leftrightarrow I_\sigma(\texttt{(por*}\ G\ \ H\texttt{)}) \\ &\leftrightarrow I_\sigma(F). \end{aligned}$$

□

With these lemmas in place, we can now establish that instantiation and β-reduction are validity-preserving.

Theorem 2.4. *The instantiation rule is validity-preserving.*

Proof. Suppose A is a valid formula. We want to show $I(A/\sigma)$ holds for every interpretation, I. This is quite simple,

$$\begin{aligned} I(A/\sigma) &\leftrightarrow I_\sigma(A) && \text{by Lemma 2.3} \\ &\leftrightarrow \text{true} && \text{by the validity of } A \end{aligned} \qquad \square$$

Theorem 2.5. *The β-reduction rule is validity-preserving.*

Proof. Let $\sigma = [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$; we want to show, for every interpretation I,

$$I(\texttt{((lambda (}x_1\ \ldots\ x_n\texttt{)}\ \beta\texttt{)}\ t_1\ \ldots\ t_n\texttt{))} = I(\beta/\sigma),$$

which is the same as $I'(\beta) = I(\beta/\sigma)$, where

$$I'(v) = \begin{cases} I(t_i) & \text{if } v = x_i \text{ for some } i \\ I(v) & \text{otherwise,} \end{cases}$$

which by Lemma 2.2 is the same as $I'(\beta) = I_\sigma(\beta)$.

This follows from Lemma 2.1 if we can establish $I'(v) = I_\sigma(v)$ for all $v \in \textsc{freevars}(\beta)$. In fact, since $\textsc{freevars}(\beta) \subseteq \{x_1, \ldots, x_n\}$, it suffices to show $I'(x_i) = I_\sigma(x_i)$ for all i. But this is straightforward:

$$I'(x_i) = I(t_i) = I(x_i/\sigma) = I_\sigma(x_i). \qquad \square$$

2.4 Primitive Functions

We now axiomatize the behavior of twelve primitive functions which will be used as building blocks for recursive definitions. The names and arities of our primitive functions are listed in the following table.

`if`	3	`natp`	1	`consp`	1
`equal`	2	`<`	2	`cons`	2
`symbolp`	1	`+`	2	`car`	1
`symbol-<`	2	`-`	2	`cdr`	1

In a moment we will introduce a list of axioms that characterize the behaviors of these functions, but first we would like to explain their intended semantics. Recall that meaning is ascribed to function names via a first-order structure, $\mathcal{A}$. And so, below, we describe the first-order structures which capture our intended semantics for these primitives, which we call the *standard structures*.

The *atoms* are the naturals, $\mathbb{N}$, and the ASCII strings, $\mathbb{S}$; the *standard universe*, $\mathbb{U}$, is the recursive closure of the atoms under ordered pairing. The non-atoms of $\mathbb{U}$ are called *conses*. We write the *cons* (ordered pair) of a and b as $(a \ . \ b)$, and we adopt some abbreviations which are similar to those for token trees:

Abbreviation	Meaning
$()$	NIL
(x)	$(x \ . \ \text{NIL})$
$(x_1 \ x_2 \ \dots \ x_n)$	$(x_1 \ . \ (x_2 \ \dots \ x_n))$
$(x_1 \ x_2 \ \dots \ x_n \ . \ b)$	$(x_1 \ . \ (x_2 \ \dots \ x_n \ . \ b))$

In a standard structure, the universe is the standard universe (i.e., $|\mathcal{A}| = \mathbb{U}$) and the constants are mapped to $|\mathcal{A}|$ as follows. For every numeric token n, $(\texttt{'}n)^{\mathcal{A}}$ is the corresponding natural number; for instance, $(\texttt{'0})^{\mathcal{A}} = 0$, $(\texttt{'1})^{\mathcal{A}} = 1$, and so on. For every symbolic token, s, $s^{\mathcal{A}}$ is the corresponding ASCII string; for instance $(\texttt{'foo})^{\mathcal{A}} =$ FOO, $(\texttt{'bar})^{\mathcal{A}} =$ BAR, etc. Finally, for compound token trees, $(\texttt{'(}a \ . \ b\texttt{)})^{\mathcal{A}}$ is the ordered pair, $(\texttt{'}a^{\mathcal{A}} \ . \ \texttt{'}b^{\mathcal{A}})$. Note the one-to-one correspondence between the constants and the elements of $\mathbb{U}$.

We now introduce several total functions on $\mathbb{U}$. We write $\ll$ to denote the strict lexicographic ordering on ASCII strings, and write $\ominus$ for natural-number subtraction, e.g., $3 \ominus 5 = 0$.

$$\textsc{natfix}(x) = \begin{cases} x & \text{if } x \in \mathbb{N} \\ 0 & \text{otherwise,} \end{cases}$$

$$\textsc{symfix}(x) = \begin{cases} x & \text{if } x \in \mathbb{S} \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{if}(x, y, z) = \begin{cases} y & \text{if } x \neq \text{NIL} \\ z & \text{otherwise,} \end{cases}$$

$$\textsc{equal}(x, y) = \begin{cases} \text{T} & \text{if } x = y \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{symbolp}(x, y) = \begin{cases} \text{T} & \text{if } x \in \mathbb{S} \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{symbol-<}(x, y) = \begin{cases} \text{T} & \text{if } \textsc{symfix}(x) \ll \textsc{symfix}(y) \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{natp}(x) = \begin{cases} \text{T} & \text{if } x \in \mathbb{N} \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{lessp}(x, y) = \begin{cases} \text{T} & \text{if } \textsc{natfix}(x) < \textsc{natfix}(y) \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{plus}(x, y) = \textsc{natfix}(x) + \textsc{natfix}(y),$$

$$\textsc{minus}(x, y) = \textsc{natfix}(x) \ominus \textsc{natfix}(x),$$

$$\textsc{consp}(x) = \begin{cases} \text{T} & \text{if } x \text{ is an ordered pair} \\ \text{NIL} & \text{otherwise,} \end{cases}$$

$$\textsc{cons}(x, y) = (x \ \ . \ \ y),$$

$$\textsc{car}(x) = \begin{cases} a & \text{if } x \text{ is an ordered pair, } (a \ \ . \ \ b) \\ \text{NIL} & \text{otherwise, and} \end{cases}$$

$$\textsc{cdr}(x) = \begin{cases} b & \text{if } x \text{ is an ordered pair, } (a \ \ . \ \ b) \\ \text{NIL} & \text{otherwise.} \end{cases}$$

A standard structure maps the primitives to these functions as follows.

$$\begin{aligned}
\texttt{if}^{\mathcal{A}}(x,y,z) &= \text{IF}(x,y,z),\\
\texttt{equal}^{\mathcal{A}}(x,y) &= \text{EQUAL}(x,y),\\
\texttt{symbolp}^{\mathcal{A}}(x) &= \text{SYMBOLP}(x),\\
\texttt{symbol-<}^{\mathcal{A}}(x,y) &= \text{SYMBOL-<}(x,y),\\
\texttt{natp}^{\mathcal{A}}(x) &= \text{NATP}(x),\\
\texttt{<}^{\mathcal{A}}(x,y) &= \text{LESSP}(x,y),\\
\texttt{+}^{\mathcal{A}}(x,y) &= \text{PLUS}(x,y),\\
\texttt{-}^{\mathcal{A}}(x,y) &= \text{MINUS}(x,y),\\
\texttt{consp}^{\mathcal{A}}(x) &= \text{CONSP}(x),\\
\texttt{cons}^{\mathcal{A}}(x,y) &= \text{CONS}(x,y),\\
\texttt{car}^{\mathcal{A}}(x) &= \text{CAR}(x), \text{and}\\
\texttt{cdr}^{\mathcal{A}}(x) &= \text{CDR}(x).
\end{aligned}$$

We now introduce several axioms to capture the behavior of these primitives. It is straightforward to see that these formulas are valid in any standard structure.

Axiom 1. `reflexivity`

```
(pequal* x x)
```

Axiom 2. `equality`

```
(por* (pnot* (pequal* x1 y1))
      (por* (pnot* (pequal* x2 y2))
            (por* (pnot* (pequal* x1 x2))
                  (pequal* y1 y2))))
```

Axiom 3. `t-not-nil`

```
(pnot* (pequal* 't 'nil))
```

Axiom 4. `equal-when-same`
```
(por* (pnot* (pequal* x y))
      (pequal* (equal x y) 't))
```

Axiom 5. `equal-when-diff`
```
(por* (pequal* x y)
      (pequal* (equal x y) 'nil))
```

Axiom 6. `if-when-nil`
```
(por* (pnot* (pequal* x 'nil))
      (pequal* (if x y z) z))
```

Axiom 7. `if-when-not-nil`
```
(por* (pequal* x 'nil)
      (pequal* (if x y z) y))
```

Axiom 8. `consp-of-cons`
```
(pequal* (consp (cons x y)) 't)
```

Axiom 9. `car-of-cons`
```
(pequal* (car (cons x y)) x)
```

Axiom 10. `cdr-of-cons`
```
(pequal* (cdr (cons x y)) y)
```

Axiom 11. `consp-nil-or-t`
```
(por* (pequal* (consp x) 'nil)
      (pequal* (consp x) 't))
```

Axiom 12. `car-when-not-consp`
```
(por* (pnot* (pequal* (consp x) 'nil))
      (pequal* (car x) 'nil))
```

Axiom 13. `cdr-when-not-consp`

```
(por* (pnot* (pequal* (consp x) 'nil))
      (pequal* (cdr x) 'nil))
```

Axiom 14. `cons-of-car-and-cdr`

```
(por* (pequal* (consp x) 'nil)
      (pequal* (cons (car x) (cdr x)) x))
```

Axiom 15. `symbolp-nil-or-t`

```
(por* (pequal* (symbolp x) 'nil)
      (pequal* (symbolp x) 't))
```

Axiom 16. `symbol-<-nil-or-t`

```
(por* (pequal* (symbol-< x y) 'nil)
      (pequal* (symbol-< x y) 't))
```

Axiom 17. `irreflexivity-of-symbol-<`

```
(pequal* (symbol-< x x) 'nil)
```

Axiom 18. `antisymmetry-of-symbol-<`

```
(por* (pequal* (symbol-< x y) 'nil)
      (pequal* (symbol-< y x) 'nil))
```

Axiom 19. `transitivity-of-symbol-<`

```
(por* (pequal* (symbol-< x y) 'nil)
      (por* (pequal* (symbol-< y z) 'nil)
            (pequal* (symbol-< x z) 't)))
```

Axiom 20. `trichotomy-of-symbol-<`

```
(por* (pequal* (symbolp x) 'nil)
      (por* (pequal* (symbolp y) 'nil)
            (por* (pequal* (symbol-< x y) 't)
                  (por* (pequal* (symbol-< y x) 't)
                        (pequal* x y)))))
```

Axiom 21. `symbol-<-completion-left`

```
(por* (pequal* (symbolp x) 't)
      (pequal* (symbol-< x y)
               (symbol-< 'nil y)))
```

Axiom 22. `symbol-<-completion-right`

```
(por* (pequal* (symbolp y) 't)
      (pequal* (symbol-< x y)
               (symbol-< x 'nil)))
```

Axiom 23. `disjoint-symbols-and-naturals`

```
(por* (pequal* (symbolp x) 'nil)
      (pequal* (natp x) 'nil))
```

Axiom 24. `disjoint-symbols-and-conses`

```
(por* (pequal* (symbolp x) 'nil)
      (pequal* (consp x) 'nil))
```

Axiom 25. `disjoint-naturals-and-conses`

```
(por* (pequal* (natp x) 'nil)
      (pequal* (consp x) 'nil))
```

Axiom 26. `natp-nil-or-t`

```
(por* (pequal* (natp x) 'nil)
      (pequal* (natp x) 't))
```

Axiom 27. `natp-of-plus`

```
(pequal* (natp (+ a b)) 't)
```

Axiom 28. `commutativity-of-+`

```
(pequal* (+ a b) (+ b a))
```

Axiom 29. `associativity-of-+`

```
(pequal* (+ (+ a b) c)
         (+ a (+ b c)))
```

Axiom 30. `plus-when-not-natp-left`

```
(por* (pequal* (natp a) 't)
      (pequal* (+ a b) (+ '0 b)))
```

Axiom 31. `plus-of-zero-when-natural`

```
(por* (pequal* (natp a) 'nil)
      (pequal* (+ a '0) a))
```

Axiom 32. `<-nil-or-t`

```
(por* (pequal* (< x y) 'nil)
      (pequal* (< x y) 't))
```

Axiom 33. `irreflexivity-of-<`

```
(pequal* (< a a) 'nil)
```

Axiom 34. `less-of-zero-right`

```
(pequal* (< a '0) 'nil)
```

Axiom 35. `less-of-zero-left-when-natp`

```
(por* (pequal* (natp a) 'nil)
      (pequal* (< '0 a)
               (if (equal a '0) 'nil 't)))
```

Axiom 36. `less-completion-left`

```
(por* (pequal* (natp a) 't)
      (pequal* (< a b)
               (< '0 b)))
```

Axiom 37. `less-completion-right`

```
(por* (pequal* (natp b) 't)
      (pequal* (< a b)
               'nil))
```

Axiom 38. `transitivity-of-<`

```
(por* (pequal* (< a b) 'nil)
      (por* (pequal* (< b c) 'nil)
            (pequal* (< a c) 't)))
```

Axiom 39. `trichotomy-of-<-when-natp`

```
(por* (pequal* (natp a) 'nil)
      (por* (pequal* (natp b) 'nil)
            (por* (pequal* (< a b) 't)
                  (por* (pequal* (< b a) 't)
                        (pequal* a b)))))
```

Axiom 40. `one-plus-trick`

```
(por* (pequal* (< a b) 'nil)
      (pequal* (< b (+ '1 a)) 'nil))
```

Axiom 41. `natural-less-than-one-is-zero`

```
(por* (pequal* (natp a) 'nil)
      (por* (pequal* (< a '1) 'nil)
            (pequal* a '0)))
```

Axiom 42. `less-than-of-plus-and-plus`

```
(pequal* (< (+ a b) (+ a c))
         (< b c))
```

Axiom 43. `natp-of-minus`

```
(pequal* (natp (- a b)) 't)
```

Axiom 44. `minus-when-subtrahend-as-large`

```
(por* (pequal* (< b a) 't)
      (pequal* (- a b) '0))
```

Axiom 45. `minus-cancels-summand-right`

```
(pequal* (- (+ a b) b)
         (if (natp a) a '0))
```

Axiom 46. `less-of-minus-left`

```
(por* (pequal* (< b a) 'nil)
      (pequal* (< (- a b) c)
               (< a (+ b c))))
```

Axiom 47. `less-of-minus-right`

```
(pequal* (< a (- b c))
         (< (+ a c) b))
```

Axiom 48. `plus-of-minus-right`

```
(por* (pequal* (< c b) 'nil)
      (pequal* (+ a (- b c))
               (- (+ a b) c)))
```

Axiom 49. `minus-of-minus-right`

```
(por* (pequal* (< c b) 'nil)
      (pequal* (- a (- b c))
               (- (+ a c) b)))
```

Axiom 50. `minus-of-minus-left`

```
(pequal* (- (- a b) c)
         (- a (+ b c)))
```

Axiom 51. `equal-of-minus-property`

```
(por* (pequal* (< b a) 'nil)
      (pequal* (equal (- a b) c)
            (equal a (+ b c))))
```

Axiom 52. `closed-universe`

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

These axioms can be used to reason symbolically about the primitives, but we also need a mechanism for explaining how the primitives operate on particular, concrete values. For this, we introduce the **base evaluation** rule of inference: when f is a primitive function of arity n, and $c_1, \ldots, c_n$ are constants, we may derive

$$\texttt{(pequal* (}f\ c_1\ \ldots\ c_n\texttt{) '}x\texttt{)},$$

where `'`x is the constant which satisfies $f^{\mathcal{A}}(c_1{}^{\mathcal{A}}, \ldots, c_n{}^{\mathcal{A}}) = (\texttt{'}x)^{\mathcal{A}}$ in a standard structure. For instance, using base evaluation, we may derive formulas such as `(pequal* (+ '3 '5) '8)` and `(pequal* (cons 'a 'b) '(a . b))`. It is trivial to see that the base evaluation rule only allows us to prove formulas which are valid in a standard structure.

2.5 Abbreviations

It is not convenient to write programs in terms of the primitives alone, so we adopt some abbreviations which make certain terms easier to write.

Recall that the numeric tokens, the symbol `t`, and the symbol `nil` are not terms. Since we often wish to use numeric constants and the constants `'t` and `'nil` in terms, we adopt a convention wherein every numeric token, n, may be used as an abbreviation for `'`n, `t` abbreviates `'t`, and `nil` abbreviates `'nil`.

We often wish to work with lists of elements. Following the convention from Lisp, we typically represent the empty list as NIL, and represent the list of $x_1, \ldots, x_n$

as (x_1 x_2 ... x_n). In a standard structure, the following abbreviations allow us to access the leading elements of such a list, or produce NIL when the list is not long enough.

Abbreviation	Meaning
`(first` x`)`	`(car` x`)`
`(second` x`)`	`(first (cdr` x`))`
`(third` x`)`	`(second (cdr` x`))`
`(fourth` x`)`	`(third (cdr` x`))`
`(fifth` x`)`	`(fourth (cdr` x`))`

Meanwhile, the abbreviation `list` may be used to construct a list from an arbitrary number of arguments.

Abbreviation	Meaning
`(list)`	`nil`
`(list` x_1`)`	`(cons` x_1 `nil)`
`(list` x_1 ... x_n`)`	`(cons` x_1 `(list` x_2 ... x_n`))`

The primitive control structure, `if`, interprets NIL as false and any other object as true. We define the abbreviation `and` as a way to ask if a litany of arguments are all non-NIL, and the abbreviation `or` as a way to ask if at least some argument is non-NIL. Following the convention from Lisp, `and` returns its final argument when all of its arguments are non-NIL, and `or` returns its first non-NIL argument when one exists.

Abbreviation	Meaning
`(and)`	`t`
`(and` x_1`)`	x_1
`(and` x_1 ... x_n`)`	`(if` x_1 `(and` x_2 ... x_n`) nil)`
`(or)`	`nil`
`(or` x_1`)`	x_1
`(or` x_1 ... x_n`)`	`(if` x_1 x_1 `(or` x_2 ... x_n`))`

Case-structured if-expressions can be introduced using `cond`, which takes a list of conditions and results as arguments, and returns the first result whose condition evaluates to non-NIL, or NIL if all of the conditions evaluate to NIL. That is, `(cond)` abbreviates `nil`, while

$$\texttt{(cond (}\mathit{cond}_1\ \mathit{result}_1\texttt{)}\ \ldots\ \texttt{(}\mathit{cond}_n\ \mathit{result}_n\texttt{))}$$

abbreviates

$$\texttt{(if }\mathit{cond}_1\ \mathit{result}_1\ \texttt{(cond (}\mathit{cond}_2\ \mathit{result}_2\texttt{)}\ \ldots\ \texttt{(}\mathit{cond}_n\ \mathit{result}_n\texttt{)))}.$$

Lambda abbreviations are particularly cumbersome to write due to the requirement that every free variable be bound. To make this easier, given unique variables $\mathit{var}_1, \ldots, \mathit{var}_n$,

$$\texttt{(let ((}\mathit{var}_1\ \mathit{term}_1\texttt{)}\ \ldots\ \texttt{(}\mathit{var}_n\ \mathit{term}_n\texttt{))}\ \beta\texttt{)}$$

is an abbreviation for

$$\texttt{((lambda (}x_1\ \ldots\ x_m\ \mathit{var}_1\ \ldots\ \mathit{var}_n\texttt{)}\ \beta\texttt{)}\ x_1\ \ldots\ x_m\ \mathit{term}_1\ \ldots\ \mathit{term}_n\texttt{)},$$

where $x_1, \ldots, x_m$ are the free variables of β besides $\mathit{var}_1, \ldots, \mathit{var}_n$ in lexicographic order. In other words, `let` is like a lambda which implicitly binds all of the unbound variables in β to themselves.

Finally, `let` abbreviations effectively bind $\mathit{var}_1, \ldots, \mathit{var}_n$ simultaneously. That is, the new value for var_1 cannot be used in term_2 to define var_2, and so on. This is sometimes inconvenient, so as an alternative, the abbreviation `let*` introduces a series of lambdas which bind each variable in order. It is convenient to define the abbreviation `let*` in terms of `let`. That is, $\texttt{(let* ()}\ \beta\texttt{)}$ abbreviates β, while

$$\texttt{(let* ((}\mathit{var}_1\ \mathit{term}_1\texttt{)}\ \ldots\ \texttt{(}\mathit{var}_n\ \mathit{term}_n\texttt{))}\ \beta\texttt{)}$$

abbreviates

$$\begin{array}{l}\texttt{(let ((}\mathit{var}_1\ \mathit{term}_1\texttt{))}\\ \quad\texttt{(let* ((}\mathit{var}_2\ \mathit{term}_2\texttt{)}\ \ldots\ \texttt{(}\mathit{var}_n\ \mathit{term}_n\texttt{))}\\ \qquad\beta\texttt{))}.\end{array}$$

2.6 Defining Functions with Axioms

Before proceeding further, we would like to introduce some additional functions in terms of the primitives. In order to reason about these new functions, we add *definitional axioms* which describe their behavior.

The function `not` may be used to negate its argument in the sense of `if`, where NIL is considered to be false and any non-NIL object is considered true. This function provides a term-level notion of equality, `not` provides a term-level notion of negation, whereas `pnot*` is a formula-level concept. The definitional axiom for `not` is:

Axiom 53. `definition-of-not`

```
(pequal* (not x)
         (if x nil t))
```

If $\mathcal{A}$ is any standard structure in which this axiom is valid, then the behavior of $\texttt{not}^{\mathcal{A}}$ has been completely specified and $\texttt{not}^{\mathcal{A}}$ must be a particular function, which we will name NOT. Why is this? If `definition-of-not` is valid, then

$$\forall I, I(\texttt{(not x)}) = I(\texttt{(if x nil t)}),$$

which is the same as

$$\forall I, \texttt{not}^{\mathcal{A}}(I(\texttt{x})) = \textsc{if}(I(\texttt{x}), \text{NIL}, \text{T}).$$

But given any $x \in \mathbb{U}$, we can pick an interpretation which maps `x` to x, and so we may conclude

$$\texttt{not}^{\mathcal{A}}(x) = \textsc{if}(x, \text{NIL}, \text{T}),$$

which, by the definition of IF, leads to:

$$\textsc{not}(x) = \texttt{not}^{\mathcal{A}}(x) = \begin{cases} \text{NIL} & \text{if } x \neq \text{NIL} \\ \text{T} & \text{otherwise.} \end{cases}$$

The behavior of recursive functions may also be captured with definitional axioms. For instance, the function `rank` may be used to count the number of conses in an object. The definitional axiom for `rank` is:

Axiom 54. `definition-of-rank`

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

It is only slightly more complicated to see that $\texttt{rank}^{\mathcal{A}}(x) = \textsc{rank}(x)$, where $\textsc{rank}$ is recursively defined as

$$\textsc{rank}(x) = \begin{cases} 0 & \text{if } x \text{ is an atom} \\ 1 + \textsc{rank}(a) + \textsc{rank}(b) & \text{otherwise, where } x \text{ is } (a\ .\ b). \end{cases}$$

We begin as before. Since `definition-of-rank` is valid,

$$\forall I, I(\texttt{(rank x)}) = I\left(\begin{array}{l}\texttt{(if (consp x)} \\ \texttt{\ \ \ \ (+ 1} \\ \texttt{\ \ \ \ \ \ \ (+ (rank (car x))} \\ \texttt{\ \ \ \ \ \ \ \ \ \ (rank (cdr x))))} \\ \texttt{\ \ 0))}\end{array}\right),$$

which is the same as

$$\forall I, \texttt{rank}^{\mathcal{A}}(I(\texttt{x})) = \textsc{if}\left(\begin{array}{l}\textsc{consp}(I(\texttt{x})), \\ \textsc{plus}\left(1, \textsc{plus}\left(\begin{array}{l}\texttt{rank}^{\mathcal{A}}(\textsc{car}(I(\texttt{x})), \\ \texttt{rank}^{\mathcal{A}}(\textsc{cdr}(I(\texttt{x}))\end{array}\right)\right), \\ 0\end{array}\right).$$

Once again, we can choose an interpretation which maps $I(\texttt{x})$ to x for any choice of x, and so we have

$$\texttt{rank}^{\mathcal{A}}(x) = \textsc{if}\left(\begin{array}{l}\textsc{consp}(x), \\ \textsc{plus}\left(1, \textsc{plus}\left(\begin{array}{l}\texttt{rank}^{\mathcal{A}}(\textsc{car}(x)), \\ \texttt{rank}^{\mathcal{A}}(\textsc{cdr}(x))\end{array}\right)\right), \\ 0\end{array}\right),$$

which, applying definitions, is just

$$\texttt{rank}^{\mathcal{A}}(x) = \begin{cases} 0 & \text{if } x \text{ is an atom} \\ \left(\begin{array}{l} 1 + \textsc{natfix}(\texttt{rank}^{\mathcal{A}}(a)) \\ \qquad + \textsc{natfix}(\texttt{rank}^{\mathcal{A}}(b)) \end{array}\right) & \text{otherwise, where } x = (a\ .\ b). \end{cases}$$

Finally, we proceed by structural induction on $\mathbb{U}$ to show that $\texttt{rank}^{\mathcal{A}}(x) = \textsc{rank}(x)$ for all $x \in \mathbb{U}$. The basis, when x is an atom, is trivial. Otherwise, inductively assuming $\texttt{rank}^{\mathcal{A}}(a) = \textsc{rank}(a)$ and that $\texttt{rank}^{\mathcal{A}}(b) = \textsc{rank}(b)$,

$$\begin{aligned} \textsc{rank}((a\ .\ b)) &= 1 + \textsc{rank}(a) + \textsc{rank}(b) \\ &= 1 + \textsc{natfix}(\textsc{rank}(a)) + \textsc{natfix}(\textsc{rank}(b)) \\ &= 1 + \texttt{rank}^{\mathcal{A}}(a) + \texttt{rank}^{\mathcal{A}}(b) \\ &= \texttt{rank}^{\mathcal{A}}(x). \end{aligned}$$

2.7 Ordinals

Our induction rule relies upon a Cantor normal form encoding of the ordinals under ε_0 into $\mathbb{U}$, and is adapted from the ACL2 implementation of ordinals developed by Manolios and Vroon. [62] These ordinals may be written, uniquely, as sums of the form $k_1\omega^{\alpha_1} + \cdots + k_n\omega^{\alpha_n} + p$, where $k_1, \ldots, k_n$ are non-zero naturals, p is a natural, and the α_i are, recursively, non-zero ordinals, also under ε_0, with $\alpha_1 \succ \alpha_2 \succ \cdots \succ \alpha_n$. When $n = 0$, this sum is simply p and we encode it in $\mathbb{U}$ as p itself. Otherwise, $n > 0$ and we encode it as $((\textsc{ordenc}(\alpha_1)\ .\ k_1)\ \ldots\ (\textsc{ordenc}(\alpha_n)\ .\ k_n)\ .\ p)$, where $\textsc{ordenc}(\alpha_i)$ is the encoding of α_i.

We now introduce definitional axioms for `ord<`, which determines if one encoded ordinal is smaller than another, and `ordp`, which determines if an object is a valid encoding of an ordinal. Per routine, these axioms completely specify $\texttt{ord<}^{\mathcal{A}}$ and $\texttt{ordp}^{\mathcal{A}}$ in any standard structure for which they are valid, and we will name these functions $\textsc{ord<}$ and $\textsc{ordp}$.

Axiom 55. `definition-of-ord<`

```
(pequal* (ord< x y)
         (cond ((not (consp x))
                (if (consp y)
                    t
                  (< x y)))
               ((not (consp y))
                nil)
               ((not (equal (car (car x))
                            (car (car y))))
                (ord< (car (car x))
                      (car (car y))))
               ((not (equal (cdr (car x))
                            (cdr (car y))))
                (< (cdr (car x))
                   (cdr (car y))))
               (t
                (ord< (cdr x) (cdr y)))))
```

Axiom 56. `definition-of-ordp`

```
(pequal* (ordp x)
         (if (not (consp x))
             (natp x)
           (and (consp (car x))
                (ordp (car (car x)))
                (not (equal (car (car x)) 0))
                (< 0 (cdr (car x)))
                (ordp (cdr x))
                (if (consp (cdr x))
                    (ord< (car (car (cdr x)))
                          (car (car x)))
                  t))))
```

Recall a few definitions from mathematics. A *strict partial order*, $\prec$, is a binary relation which is irreflexive, antisymmetric, and transitive over some set, X. When $x \prec y$ we say x is *smaller* than y and that y is *larger* than x, and we may also write $y \succ x$. A *strict total order* is a strict partial order which also satisfies the property of trichotomy, $\forall x, y \in X, (x = y) \vee (x \prec y) \vee (x \succ y)$. A (possibly infinite) sequence of elements from X, $(A_n) = (a_0, a_1, \dots)$ is said to be *strictly decreasing* when $a_n \succ a_{n+1}$ for all n, and *strictly increasing* when $a_n \prec a_{n+1}$ for all n. The relation $\prec$ is *well-founded* on X when every strictly decreasing sequence from X is finite, or equivalently (assuming the axiom of choice) when every subset of X has a $\prec$-minimal element. A *well-ordering* of X is a well-founded, strict total order.

The crucial property of ORD< is that it well-orders the objects recognized by ORDP. To be more precise, it should be clear that ORD<(x, y) always returns either T or NIL, so let us say the relation $x \prec y$ holds when ORD<$(x, y) =$ T. Similarly, ORDP(x) returns T or NIL, so let $\mathcal{O}$ be the set $\{x \in \mathbb{U} : \text{ORDP}(x) = \text{T}\}$. We will now establish that $\prec$ is a well-ordering of $\mathcal{O}$.

Lemma 2.6 (Irreflexivity). *$\forall x \in \mathcal{O}, x \not\prec x$.*

Proof. We will show this holds for all $x \in \mathbb{U}$, which is sufficient since $\mathcal{O} \subseteq \mathbb{U}$. The proof is by structural induction on $\mathbb{U}$. As a basis, if x is an atom, then $x \prec x$ exactly when NATFIX$(x) <$ NATFIX(x), so by the irreflexivity of $<$, $x \not\prec x$. Otherwise, let $x = (a\ .\ b)$. Now $x \prec x$ precisely when $b \prec b$, but we may inductively assume $b \not\prec b$, so $x \not\prec x$. □

Lemma 2.7 (Antisymmetry). *$\forall x, y \in \mathcal{O}, x \prec y \rightarrow y \not\prec x$.*

Proof. Again we will show this holds for any $x, y \in \mathbb{U}$. The proof is by induction on RANK$(x) +$ RANK(y). That is, let n be RANK$(x) +$ RANK(y) and suppose whenever RANK$(a) +$ RANK$(b) < n$, $a \prec b \rightarrow b \not\prec a$.

As a basis, if $n = 0$ then x and y are atoms, so

$$x \prec y = \textsc{natfix}(x) < \textsc{natfix}(y), \text{ while}$$

$$y \prec x = \textsc{natfix}(y) < \textsc{natfix}(x),$$

and by the antisymmetry of $<$ we are done.

Otherwise, $n > 0$ so at least one of x or y is a cons. If only one of x or y is a cons, our goal is trivial, so for the remainder of the proof assume they are both conses.

Case 1: $\textsc{car}(\textsc{car}(x)) \neq \textsc{car}(\textsc{car}(y))$. Let a be $\textsc{car}(\textsc{car}(x))$ and b be $\textsc{car}(\textsc{car}(y))$. Then

$$x \prec y = a \prec b, \text{ while}$$

$$y \prec x = b \prec a,$$

and since we may inductively assume $a \prec b \rightarrow b \not\prec a$, we are done.

Case 2: $\textsc{car}(\textsc{car}(x)) = \textsc{car}(\textsc{car}(y))$, $\textsc{cdr}(\textsc{car}(x)) \neq \textsc{cdr}(\textsc{car}(y))$. Let a be $\textsc{cdr}(\textsc{car}(x))$ and b be $\textsc{cdr}(\textsc{car}(y))$. Then

$$x \prec y = \textsc{natfix}(a) < \textsc{natfix}(b), \text{ while}$$

$$y \prec x = \textsc{natfix}(b) < \textsc{natfix}(a),$$

so by the antisymmetry of $<$ we are done.

Case 3: $\textsc{car}(\textsc{car}(x)) = \textsc{car}(\textsc{car}(y))$, $\textsc{cdr}(\textsc{car}(x)) = \textsc{cdr}(\textsc{car}(y))$ Let a be $\textsc{cdr}(x)$ and b be $\textsc{cdr}(y)$. Then

$$x \prec y = a \prec b, \text{ while}$$

$$y \prec x = b \prec a,$$

and since we may inductively assume $a \prec b \rightarrow b \not\prec a$, we are done. □

Lemma 2.8 (Transitivity). $\forall x, y, z \in \mathcal{O}, x \prec y \wedge y \prec z \rightarrow x \prec z.$

Proof. Again we will show this holds for $x, y, z \in \mathbb{U}$. The proof is by induction on $\textsc{rank}(x) + \textsc{rank}(y) + \textsc{rank}(z)$. That is, suppose $\textsc{rank}(x) + \textsc{rank}(y) + \textsc{rank}(z) = n$ and inductively suppose if $\textsc{rank}(a) + \textsc{rank}(b) + \textsc{rank}(c) < n$, $a \prec b$, and $b \prec c$, then $a \prec c$. Finally, assume $x \prec y$ and $y \prec z$, so our goal is to show $x \prec z$.

As a basis, if $n = 0$, none of x, y, z are conses, and we have:

$$\begin{aligned}
&x \prec y = \textsc{natfix}(x) < \textsc{natfix}(y),\\
&y \prec z = \textsc{natfix}(y) < \textsc{natfix}(z), \text{ and}\\
&x \prec z = \textsc{natfix}(x) < \textsc{natfix}(z),
\end{aligned}$$

so by the transitivity of $<$, we are done.

Otherwise, $n > 0$. First, note that z must be a cons: otherwise $y \prec z$ would imply y is an atom, $x \prec y$ would imply x is an atom, and n would be 0. Furthermore, if x is an atom then $x \prec z$ is trivial, so assume x is a cons. Finally, since $x \prec y$, y must also be a cons. Now, we let

$$\begin{array}{lll}
\alpha_x = \textsc{car}(\textsc{car}(x)), & k_x = \textsc{cdr}(\textsc{car}(x)), & \beta_x = \textsc{cdr}(x),\\
\alpha_y = \textsc{car}(\textsc{car}(y)), & k_y = \textsc{cdr}(\textsc{car}(y)), & \beta_y = \textsc{cdr}(y),\\
\alpha_z = \textsc{car}(\textsc{car}(z)), & k_z = \textsc{cdr}(\textsc{car}(z)), & \beta_z = \textsc{cdr}(z),
\end{array}$$

and proceed by cases.

A. Suppose $\alpha_x = \alpha_y = \alpha_z$ and $k_x = k_y = k_z$.

   By our inductive hypothesis, we may assume $\beta_x \prec \beta_y \wedge \beta_y \prec \beta_z \rightarrow \beta_x \prec \beta_z$. But in this case, $x \prec y = \beta_x \prec \beta_y$, $y \prec z = \beta_y \prec \beta_z$, and $x \prec z = \beta_x \prec \beta_z$, so we are done.

B. Suppose $\alpha_x = \alpha_y = \alpha_z$ but either $k_x \neq k_y$ or $k_y \neq k_z$.

B1. $k_x \neq k_y, k_y \neq k_z$. Since $x \prec y$, $k_x < k_y$. Since $y \prec z$, $k_y < k_z$. By the transitivity of $<$, $k_x < k_z$, and so $x \prec z$.

B2. $k_x \neq k_y, k_y = k_z$. Since $x \prec y$, $k_x < k_y$. By equality substitution, we see $k_x < k_z$. By irreflexivity, $k_x \neq k_z$; hence $x \prec z$.

B3. $k_x = k_y, k_y \neq k_z$. Since $y \prec z$, $k_y < k_z$. By equality substitution, $k_x < k_z$; By irreflexivity, $k_x \neq k_z$; hence $x \prec z$.

C. Suppose either $\alpha_x \neq \alpha_y$ or $\alpha_y \neq \alpha_z$.

C1. $\alpha_x \neq \alpha_y$, $\alpha_y \neq \alpha_z$. By our inductive hypothesis, we may assume $\alpha_x \prec \alpha_y \wedge \alpha_y \prec \alpha_z \rightarrow \alpha_x \prec \alpha_z$. But in this case, $x \prec y = \alpha_x \prec \alpha_y$, $y \prec z = \alpha_y \prec \alpha_z$, and $x \prec z = \alpha_x \prec \alpha_z$, so we are done.

C2. $\alpha_x \neq \alpha_y, \alpha_y = \alpha_z$. Since $x \prec y$, $\alpha_x \prec \alpha_y$. By equality substitution, $\alpha_x \prec \alpha_z$. By irreflexivity, $\alpha_x \neq \alpha_z$; hence $x \prec z$.

C3. $\alpha_x = \alpha_y, \alpha_y \neq \alpha_x$. Since $y \prec z$, $\alpha_y \prec \alpha_z$. By equality substitution, $\alpha_x \prec \alpha_z$. By irreflexivity, $\alpha_x \neq \alpha_z$; hence $x \prec z$. $\square$

Lemma 2.9 (Trichotomy)**.** $\forall x, y \in \mathcal{O}, (x = y) \vee (x \prec y) \vee (x \succ y)$

Proof. We will actually prove the following, equivalent statement,

$$\forall x, y \in \mathbb{U}, (x, y \in \mathcal{O}) \rightarrow (x = y \vee x \prec y \vee x \succ y)\,,$$

by induction on $n = \textsc{rank}(x) + \textsc{rank}(y)$. As a basis, if $n = 0$ then x and y are atoms and $x, y \in \mathcal{O}$ exactly when $x, y \in \mathbb{N}$; furthermore $x \prec y = x < y$ and $y \prec x = y < x$, and by the trichotomy of $<$ over $\mathbb{N}$ we are done.

Otherwise, suppose $n > 0$. If x is an atom then y must be a cons and $x \prec y$ and we are done, and similarly if y is an atom then $y \prec x$ and we are done. So assume x, y are both conses. Since $x, y \in \mathcal{O}$, we may let

$$\begin{aligned} \alpha_x &= \text{CAR}(\text{CAR}(x)), & k_x &= \text{CDR}(\text{CAR}(x)), & \beta_x &= \text{CDR}(x), \\ \alpha_y &= \text{CAR}(\text{CAR}(y)), & k_y &= \text{CDR}(\text{CAR}(y)), & \beta_y &= \text{CDR}(y), \end{aligned}$$

and we can see that $\alpha_x, \alpha_y, \beta_x, \beta_y \in \mathcal{O}$ and $k_x, k_y \in \mathbb{N}$.

A. Suppose $\alpha_x = \alpha_y, k_x = k_y$.

   If $\beta_x = \beta_y$, then $x = y$ and our goal is met. Otherwise, $x \prec y = \beta_x \prec \beta_y$ and $x \succ y = \beta_x \succ \beta_y$. But inductively, $(\beta_x = \beta_y) \vee (\beta_x \prec \beta_y) \vee (\beta_x \succ \beta_y)$, so either $x \prec y$ or $x \succ y$ and we are done.

B. Suppose $\alpha_x = \alpha_y, k_x \neq k_y$.

   Now $x \neq y$, and we see that $x \prec y = k_x < k_y$ and $x \succ y = k_x > k_y$. Since $k_x, k_y \in \mathbb{N}$, we know $(k_x < k_y) \vee (k_x > k_y)$, so either $x \prec y$ or $x \succ y$ and we are done.

C. Suppose $\alpha_x \neq \alpha_y$.

   Now $x \neq y$, $x \prec y = \alpha_x \prec \alpha_y$, and $x \succ y = \alpha_x \succ \alpha_y$. But inductively, we have $(\alpha_x = \alpha_y) \vee (\alpha_x \prec \alpha_y) \vee (\alpha_x \succ \alpha_y)$, and so either $x \prec y$ or $x \succ y$ and we are done. □

For the proof of well-foundedness, it is convenient to define ODEPTH : $\mathcal{O} \rightarrow \mathbb{N}$, as follows:

$$\text{ODEPTH}(x) = \begin{cases} 0 & \text{when } x \in \mathbb{N} \\ 1 + \text{ODEPTH}(\alpha) & \text{when } x = ((\alpha\ .\ k)\ .\ \beta). \end{cases}$$

It should be clear that $\text{ODEPTH}(x) \in \mathbb{N}$ for all $x \in \mathcal{O}$ and that for any $n \in \mathbb{N}$, we can construct an ordinal x with $\text{ODEPTH}(x) = n$.

Lemma 2.10. *$\forall x, y \in \mathcal{O}, \text{ODEPTH}(x) < \text{ODEPTH}(y) \rightarrow x \prec y$.*

Proof. We will actually prove

$$\forall x, y \in \mathbb{U}, ((x, y \in \mathcal{O}) \to (\textsc{odepth}(x) < \textsc{odepth}(y) \to x \prec y))$$

by induction on $n = \textsc{rank}(x) + \textsc{rank}(y)$. As a basis, if $n = 0$ then x, y are atoms and $\textsc{odepth}(x) = \textsc{odepth}(y) = 0$, so our goal is vacuously true.

So suppose $n > 0$. If x is an atom, y must be a cons and the conclusion, $x \prec y$, is trivial. If y is an atom, x must be a cons and the hypothesis $\textsc{odepth}(x) < \textsc{odepth}(y)$ is false. So assume x and y are both conses, and let $\alpha_x = \textsc{car}(\textsc{car}(x))$, and $\alpha_y = \textsc{car}(\textsc{car}(y))$. Now, we may inductively assume:

$$\textsc{odepth}(\alpha_x) < \textsc{odepth}(\alpha_y) \to \alpha_x \prec \alpha_y.$$

But since $\textsc{odepth}(x) < \textsc{odepth}(y)$ we find that $\textsc{odepth}(\alpha_x) < \textsc{odepth}(\alpha_y)$, and so $\alpha_x \prec \alpha_y$. But now, since $\alpha_x \neq \alpha_y$, our goal, $x \prec y$, reduces to $\alpha_x \prec \alpha_y$, which was just established. □

Corollary 2.11. *If $a_0, a_1, \dots \in \mathcal{O}$ with $a_0 \succ a_1 \succ \dots$, then*

$$\textsc{odepth}(a_0) \geq \textsc{odepth}(a_1) \geq \dots.$$

Lemma 2.12. *If x is a cons, $x \in \mathcal{O} \to \textsc{odepth}(\textsc{cdr}(x)) \leq \textsc{odepth}(x)$.*

Proof. Since $x \in \mathcal{O}$ and x is a cons, $\textsc{cdr}(x)$ is also an ordinal. If $\textsc{cdr}(x)$ is an atom then its $\textsc{odepth}$ is 0 and we are trivially done. Otherwise, by the definition of $\textsc{ordp}$, $\textsc{cdr}(x) \prec x$, and by lemma 2.10 we see that $\textsc{odepth}(\textsc{cdr}(x))$ can be at most $\textsc{odepth}(x)$. □

Lemma 2.13 (Well-Foundedness)**.** *If $A = (a_0, a_1, \dots)$ with $a_0, a_1, \dots \in \mathcal{O}$ and $a_0 \succ a_1 \succ \dots$, then A must be finite.*

Proof. First, note that ODEPTH can be used to stratify $\mathcal{O}$, i.e.,

$$\mathcal{O} = \bigcup_{n \in \mathbb{N}} \mathcal{O}_n, \text{ where } \mathcal{O}_n = \{x \in \mathcal{O} : \textsc{odepth}(x) \leq n\},$$

so to prove that some property $P(x)$ holds for all $x \in \mathcal{O}$, it suffices to prove that it holds for all $x \in \mathcal{O}_n$ for every n. Similarly, if we want to prove some property P about all sequences of ordinals, i.e., $(a_0, a_1, \dots)$ where each $a_i \in \mathcal{O}$, then it suffices to prove that P holds for all sequences $(a_0, a_1, \dots)$ where $a_0 \in \mathcal{O}_n$, for all n.

Our proof follows this induction scheme. That is, we will show, for all n, that if $(a_0, a_1, \dots)$ is a sequence of ordinals with $a_0 \in \mathcal{O}_n$, with $a_0 \succ a_1 \succ \dots$, then A must be finite. But by Corollary 2.11, we see that every a_i is in $\mathcal{O}_n$. So, it suffices to show that for all n, if $(a_0, a_1, \dots)$ is a sequence from $\mathcal{O}_n$ with $a_0 \succ a_1 \succ \dots$, then A must be finite.

As a basis, suppose $n = 0$. Now, $\textsc{odepth}(a_i) = 0$ for each a_i, so every a_i is a natural number, and $a_0 \succ a_1 \succ \dots$ is the same as $a_0 > a_1 > \dots$. Since $<$ is well-founded on the natural numbers, A must be finite.

Otherwise, suppose $n > 0$, and that any strictly decreasing sequence from $\mathcal{O}_{n-1}$ is finite. Let D be the set of all infinite, strictly decreasing sequences from $\mathcal{O}_n$, and assume toward contradiction that D is nonempty.

Claim: Any member of any sequence of D has an ODEPTH of n.

Proof: Let $(d_0, d_1, \dots) \in D$, and suppose toward contradiction that there is some d_i so that $\textsc{odepth}(d_i) \neq n$. By Corollary 2.11, $\textsc{odepth}(d_i) < n$, so $(d_i, d_{i+1}, \dots)$ is a strictly decreasing, infinite sequence from $\mathcal{O}_{n-1}$, which by our inductive hypothesis cannot exist. ✓.

Now, let $D_0 = \{d_0 : (d_0, d_1, \dots) \in D\}$. In other words, take all of the first elements of these infinite, strictly decreasing sequences from $\mathcal{O}_n$, and put them into a set. Since D is nonempty, D_0 is also nonempty.

Consider the elements $d \in D_0$. Since $\textsc{odepth}(d) = n$, $\textsc{odepth}(\textsc{car}(\textsc{car}(d)))$ must be $n-1$. In other words, $\textsc{car}(\textsc{car}(d)) \in \mathcal{O}_{n-1}$, and since by our inductive hypothesis $\prec$ is well-founded for $\mathcal{O}_{n-1}$, we know there is a smallest such element. Call this element α, and note that at least some $d \in D_0$ must have $\textsc{car}(\textsc{car}(d)) = \alpha$.

Now consider the elements $d \in D_0$ with $\textsc{car}(\textsc{car}(d)) = \alpha$. Since $d \in \mathcal{O}$, we know $\textsc{cdr}(\textsc{car}(d_0)) \in \mathbb{N}$, and hence there is a smallest such element. Call this element k, and now note that at least some $d \in D_0$ must have $\textsc{car}(d) = (\alpha\ .\ k)$.

We are now ready for the main part of the argument. Choose any fixed $(d_0, d_1, \dots) \in D$ with $\textsc{car}(d_0) = (\alpha\ .\ k)$

Claim: $\textsc{car}(d_i) = (\alpha\ .\ k)$ for all i.

Proof: Suppose toward contradiction that j is the first index so that $\textsc{car}(d_j) \neq (\alpha\ .\ k)$. Since d_j is an ordinal and $\textsc{odepth}(d_j) = n$ and $n > 0$, we know that d_j has the form $((\alpha'\ .\ k')\ .\ \beta)$. Furthermore, we know that $\textsc{odepth}(\alpha') = n-1$, by the definition of $\textsc{odepth}$. We consider two cases.

Case 1: $\alpha \neq \alpha'$. Since $\textsc{car}(d_{j-1}) = (\alpha\ .\ k)$ and $d_j \prec d_{j-1}$, we see that $\alpha' \prec \alpha$. Since $(d_j, d_{j+1}, \dots)$ is a strictly decreasing infinite sequence in $\mathcal{O}_n$, we see that $(d_j, d_{j+1}, \dots) \in D$, and that $\textsc{car}(\textsc{car}(d_j)) \prec \alpha$, but this contradicts the minimality of α.

Case 2: $\alpha = \alpha'$, but $k \neq k'$. Since $\textsc{car}(d_{j-1}) = (\alpha\ .\ k)$ and $d_j \prec d_{j-1}$, we see that $k' < k$. Since $(d_j, d_{j+1}, \dots)$ is a strictly decreasing infinite sequence in $\mathcal{O}_n$, we see that $(d_j, d_{j+1}, \dots) \in D$ with $\textsc{car}(\textsc{car}(d_j)) = \alpha$ and $\textsc{cdr}(\textsc{car}(d_j)) < k$, but this contradicts the minimality of k. ✓

Now, since $\textsc{car}(d_i) = (\alpha\ .\ k)$ for every i and $d_0 \succ d_1 \succ \dots$, we are left with $\textsc{cdr}(d_0) \succ \textsc{cdr}(d_1) \succ \dots$.

Claim: $\textsc{odepth}(\textsc{cdr}(d_i)) = n$ for all i.

Proof: By Lemma 2.12, we see that $\textsc{odepth}(\textsc{cdr}(d_i)) \leq \textsc{odepth}(d_i)$, i.e., $\textsc{odepth}(\textsc{cdr}(d_i)) \leq n$, so we only need to show that $\textsc{odepth}(\textsc{cdr}(d_i))$ cannot be strictly less than n. Suppose toward contradiction that there is some i with $\textsc{odepth}(\textsc{cdr}(d_i)) < n$. By Corollary 2.11, $\textsc{odepth}(\textsc{cdr}(d_j)) < n$ for all $j \geq i$. But now, $\textsc{cdr}(d_i) \succ \textsc{cdr}(d_{i+1}) \succ \ldots$ is an infinite sequence in $\mathcal{O}_{n-1}$, and by our inductive hypothesis no such sequence exists. ✓

As a result, we see that $(\textsc{cdr}(d_0), \textsc{cdr}(d_1), \ldots) \in D$. Furthermore, since $\textsc{odepth}(\textsc{cdr}(d_0)) = n$, and $n > 0$, we see that $\textsc{cdr}(d_0)$ is a cons, and by the definition of $\textsc{ordp}$ we know that $\textsc{car}(\textsc{car}(\textsc{cdr}(d_0))) \prec \textsc{car}(\textsc{car}(d_0))$, or, in other words, $\textsc{car}(\textsc{car}(\textsc{cdr}(d_0))) \prec \alpha$. This contradicts the minimality of α. $\square$

Together, these lemmas establish $\prec$ is a well-ordering of $\mathcal{O}$.

2.8 Induction

Our **induction rule** is as follows. Suppose m is a term, $q_1, \ldots, q_k$ are formulas, and for each $i = 1 \ldots k$ we have a set of substitution lists, $\Sigma_i = \{\sigma_{\langle i,1\rangle}, \ldots, \sigma_{\langle i,h_i\rangle}\}$. Then, we may derive the formula F given proofs of the

basis step,
`(por*` F `(por*` q_1 `(...` `(por*` q_{k-1} q_k`)...))`,

inductive steps, for $i = 1 \ldots k$,
```
(por* F
      (por* (pnot* q_i)
            (por* (pnot* F/σ_⟨i,1⟩)
                  ⋱
                    (por* (pnot* F/σ_⟨i,h_i−1⟩)
                          (pnot* F/σ_⟨i,h_i⟩)) ...))),
```

ordinal step,
`(pequal* (ordp` m`) t)`, and

measure steps, for $i = 1 \dots k$, $j = 1 \dots h_i$,
`(por* (pnot*` q_i`) (pequal* (ord<` $m/\sigma_{\langle i,j\rangle}$ m`) t))`.

We say a first-order structure, $\mathcal{A}$, is a *ground-zero structure* when it is a standard structure that also satisfies the definitional axioms for `not`, `rank`, `ordp`, and `ord<`.

Theorem 2.14 (Soundness of Induction)**.** *Suppose $\mathcal{A}$ is a ground-zero structure and that F is any formula which can be justified by applying the induction rule to valid formulas. Then, F is also valid.*

Proof. Let m be a term, $q_1, \dots, q_k$ be formulas, and $\Sigma_1, \dots, \Sigma_k$ be sets of substitution lists so that F is justified by the induction rule using these choices of m, q_i, and Σ_i, and assume that the basis step, inductive steps, ordinal step, and measure steps described above are all valid in $\mathcal{A}$.

Suppose toward contradiction that F is not valid in $\mathcal{A}$. Let A be the set of all interpretations which invalidate F, i.e., $A = \{J : \neg J(F)\}$. Since F is not valid, A is non-empty. By the validity of the ordinal step, $J(m) \in \mathcal{O}$ for every interpretation, and since $\prec$ is a well-ordering of $\mathcal{O}$, we may let I be an interpretation from A which gives the $\prec$-minimal interpretation to m. That is, I is an interpretation in A which satisfies $\forall J \in A, I(m) \preceq J(m)$.

Since the basis step is valid, we have $I(q_1) \vee \dots \vee I(q_k) \vee I(F)$, and so since $\neg I(F)$, there must be some i for which $I(q_i)$ holds. Furthermore, by the validity of the inductive step for i, along with $I(q_i)$ and $\neg I(F)$, we find there must be some j for which $\neg I(F/\sigma_{\langle i,j\rangle})$. Recall from Lemma 2.3 that $I(F/\sigma_{\langle i,j\rangle}) = I_{\sigma_{\langle i,j\rangle}}(F)$, so $\neg I_{\sigma_{\langle i,j\rangle}}(F)$, i.e., $I_{\sigma_{\langle i,j\rangle}} \in A$.

Now, by the validity of the measure step for i and j, along with $I(q_i)$, we have

$I(m/\sigma_{\langle i,j\rangle}) \prec I(m)$. Recall, from Lemma 2.2, that $I(m/\sigma_{\langle i,j\rangle}) = I_{\sigma_{\langle i,j\rangle}}(m)$, and so we have $I_{\sigma_{\langle i,j\rangle}}(m) \prec I(m)$, contradicting the minimality of $I(m)$. □

2.9 Events

To be useful in modeling programs, our logic needs some mechanism for introducing new concepts. Following the approach of Kaufmann and Moore [53], we say a *history* is a possibly empty, finite sequence of *events*. We associate with every history an arity table and a collection of formulas, called its axioms, each of which must be well-formed with respect to this arity table. For the empty history, we associate the fifty-six numbered axioms listed in the previous sections, and the following arity table.

`if`	3	`natp`	1	`consp`	1	`not`	1
`equal`	2	`<`	2	`cons`	2	`rank`	1
`symbolp`	1	`+`	2	`car`	1	`ordp`	1
`symbol-<`	2	`-`	2	`cdr`	1	`ord<`	2

For pragmatic reasons we also associate with every history a collection of formulas, called its theorems, each of which again must be well-formed with respect to its arity table. The empty history has no theorems. We say that a formula, ϕ, is provable from a history, h, when ϕ may be derived from the axioms and theorems of h using the rules of inference.

We allow three kinds of *events*—theorems, recursive function definitions, and witnessing function definitions. Each of these has certain criteria for *admissibility* in the current history, which ensure the history remains coherent as it is extended. For instance, only a provable formula may be admitted as a theorem, which ensures the theorems of a well-formed history are always provable from its axioms.

If $h = (e_0, \ldots, e_n)$ is a history, h may be extended with a *theorem* event, e, to form a new history, $h' = (e_0, \ldots, e_n, e)$. The event, e, says that some formula,

ϕ, should now be regarded as a theorem. The arity table and axioms of h' are the same as those of h, and the theorems of h' are ϕ and the theorems of h. For e to be admissible, ϕ must be well-formed with respect to the arity table of h and provable from h.

Alternately, h may be extended with a *recursive function definition* event, e. Such an event includes a function name, f; a list of distinct variables, $x_1, \ldots, x_n$, called its *formals*; a term, β, called its *body*; and another term, m, called its *measure*. The arity table of h' is formed by extending the arity table of h by associating f with n, the axioms of h' are the axioms of h along with the *definitional axiom*, `(pequal* (`f x_1 ... x_n`)` β`)`, and the theorems of h' are the theorems of h.

There are many requirements for a definition to be admissible in h. To begin, f must be a new name which is not already in the arity table of h. Furthermore, β and m must be well-formed with respect to the new arity table, and FREEVARS(β) and FREEVARS(m) must be subsets of $\{x_1, \ldots, x_n\}$. Finally, certain formulas called the *termination obligations*, described in a moment, must be provable from h.

The termination obligations are certain formulas whose provability ensures f describes a terminating computation, and arise when β contains recursive calls of f. The *ordinal obligation*,

`(pequal* (ordp` m`) t)`,

ensures that the measure is an ordinal, while the *progress obligations* ensure that during each recursive call, `(`f a_1 ... a_n`)`, this measure is being decreased. The basic idea is to show

`(pequal* (ord<` $m/[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n]$ m`) t)`,

but generally each recursive call only occurs under certain conditions—for instance, in the definition of `rank`,

```
(pequal* (rank x)
         (if (consp x)
             (+ 1
                (+ (rank (car x))
                   (rank (cdr x))))
           0)),
```

the recursive calls, `(rank (car x))` and `(rank (cdr x))`, only play a role in the case where `(consp x)` holds—so we really only need to ensure progress is made when these conditions are met.

We use *call maps* to explain when f is called recursively and which conditions hold during each of these calls. That is, $\textsc{callmap}(f, x)$ takes a function name, f, and a term, x, and computes a table associating each recursive call of f in x to a list of the terms which are said to *rule* that recursive call. In particular,

– When x is a constant or a variable, there are no recursive calls of f within x, so $\textsc{callmap}(f, x)$ is empty.

– When x is `(if` a b c`)`, then $\textsc{callmap}(f, x)$ includes the calls from a, verbatim; the calls from b, but modified so that a is also a ruler of each call; and the calls of c, modified so `(not` a`)` is also a ruler of each call.

– When x is `(`f t_1 $\ldots$ t_n`)`, $\textsc{callmap}(f, x)$ associates `(`f t_1 $\ldots$ t_n`)` with no rulers, and also includes the calls from $\textsc{callmap}(f, t_i)$.

– When x is any other function call, `(`g t_1 $\ldots$ t_m`)`, $\textsc{callmap}(f, x)$ is the union of $\textsc{callmap}(f, t_i)$.

– When x is `((lambda (`x_1 $\ldots$ x_n`)` β`)` t_1 $\ldots$ t_n`)`, its call map includes all calls in the actuals, i.e., $\textsc{callmap}(f, t_i)$, and also includes the modified call map of β, formed by substituting $\sigma = [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ into each call and all rulers.

The progress obligations are determined by examining the callmap for f and its body. For each recursive call, (*f* a_1 ... a_n), associated with the rulers $r_1, \ldots, r_m$, we have the obligation

```
(por* (pequal* (ord< m/σ m) t)
      (por* (pequal* r_1 nil)
            ⋱
              (por* (pequal* r_{m-1} nil)
                    (pequal* r_m nil))) ... )),
```

where $\sigma = [x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n]$. In other words, we must show either that progress is made and the measure is reduced, or that some ruler is false (and hence this recursive call does not occur).

Finally, h may be extended with a *witnessing function definition* event, e. Such an event includes a function name, f, a variable, v, called the *bound variable*, a list of distinct variables, $x_1, \ldots, x_n$ called the *free variables*, and a term, β, called its body. The arity table of h' is formed by extending the arity table of h by associating f with n, the theorems of h' are the theorems of h, and the axioms of h' are the axioms of h along with

```
(por* (pequal* β nil)
      (pnot* (pequal* ((lambda (v x_1 ... x_n) β)
                        (f x_1 ... x_n) x_1 ... x_n)
                       nil))).
```

To be admissible in h, f must be a new name which is not already in the arity table of h, v must not be any of the free variables, β must be well-formed with respect to the arity table of h and FREEVARS(β) must be a subset of $\{v, x_1, \ldots, x_n\}$.

Chapter 3

The Proof Checker

We now turn our attention to the development of our proof-checking function. This function is important in two ways. First, as a mathematical model, it forms the basis for our notion of provability, and hence it is fundamental to the statement of our theorem prover's fidelity. Second, as a program, it is used in the proof-checking system we develop in the next chapter, and is responsible for checking the proof of fidelity for the Level 2 proof checker.

The definitions provided in this chapter are admissible events when processed in order from the empty history. However, we do not wish to complicate this chapter with termination arguments, so we only refer the reader to the discussion at the end of Section 4.3.

3.1 Utilities

Our proof checker relies upon a number of auxiliary definitions. Many of these are general-purpose utilities about arithmetic, lists, etc., which have little to do with our logic in particular.

We begin with some simple arithmetic functions. Recall from page 23 that functions in our logic are not typed but are instead defined for all inputs from $\mathbb{U}$. When non-numeric inputs are encountered by the primitive functions `+`, `-`, and `<`, they are interpreted as zero, e.g., `(+ 1 nil)` is provably equal to `1`. We call this the *zero convention*. Our first function, `nfix`, performs this interpretation, viz. natural numbers are interpreted as themselves, while other objects from $\mathbb{U}$ are coerced to 0.

Definition 1: `nfix`
```
(pequal* (nfix x)
         (if (natp x)
             x
           0))
```

Next, `zp`, the *zero predicate*, determines whether its argument is zero when interpreted as a natural number.

Definition 2: `zp`
```
(pequal* (zp x)
         (if (natp x)
             (equal x 0)
           t))
```

Finally, `<=` can be used to determine whether $a \leq b$. Like our other arithmetic functions, non-numeric inputs are treated as zeroes.

Definition 3: `<=`
```
(pequal* (<= a b)
         (not (< b a)))
```

Now we move on to some list operations. In most typed, functional languages like ML, there is only one representation of the empty list, *nil*, and the fundamental operation, *cons*, has the signature $A \times A\ list \rightarrow A\ list$. In our logic, as in Lisp, `cons` is untyped and can take any arguments from $\mathbb{U}$, so when we talk about lists we usually mean a particular subset of $\mathbb{U}$, called the *true lists*: NIL is a true list, and intuitively represents the empty list; meanwhile $(a\ \ .\ \ b)$ is a true list whenever b is a true list, and intuitively represents the list where the element a precedes the elements of b. We can determine whether an object is a true list with `true-listp`, the *true-list predicate*.

Definition 4: `true-listp`
```
(pequal* (true-listp x)
         (if (consp x)
             (true-listp (cdr x))
           (equal x nil)))
```

In a moment, we will introduce some basic list operations, such as taking the length of a list and reversing a list. But first, what should such functions do when given inputs from $\mathbb{U}$ which are not true lists? Earlier, we faced a similar question in our arithmetic functions, and dealt with it by adopting the zero convention, which treats non-naturals as zero. For our list functions, we adopt a *list-fix convention*, wherein the "cdr-most" position of any object in $\mathbb{U}$ is coerced to NIL to produce a true list. For example, the `list-fix` of `(1 2 3 . 4)` is `(1 2 3)`. Just as `nfix` is the identity for natural numbers, `list-fix` leaves true-lists unchanged.

Definition 5: `list-fix`
```
(pequal* (list-fix x)
         (if (consp x)
             (cons (car x)
                   (list-fix (cdr x)))
           nil))
```

We now introduce a number of familiar list operations that respect the list-fix convention. The function `len` computes the number of elements in a list.

Definition 6: `len`
```
(pequal* (len x)
         (if (consp x)
             (+ 1 (len (cdr x)))
           0))
```

We can ask whether some object is an element of a list with `memberp`, the *membership predicate*.

Definition 7: memberp

```
(pequal* (memberp a x)
         (if (consp x)
             (or (equal a (car x))
                 (memberp a (cdr x)))
           nil))
```

We can ask whether every member of a list, x, is also a member of another list, y, using subsetp, the *subset predicate*.

Definition 8: subsetp

```
(pequal* (subsetp x y)
         (if (consp x)
             (and (memberp (car x) y)
                  (subsetp (cdr x) y))
           nil))
```

We can ask whether a list has distinct members using uniquep.

Definition 9: uniquep

```
(pequal* (uniquep x)
         (if (consp x)
             (and (not (memberp (car x) (cdr x)))
                  (uniquep (cdr x)))
           t))
```

We can append lists together with app. Note that we explicitly list-fix y in the base case to ensure app always produces a true list.

Definition 10: app

```
(pequal* (app x y)
         (if (consp x)
             (cons (car x)
                   (app (cdr x) y))
           (list-fix y)))
```

Finally, we can reverse a list with `rev`.

Definition 11: `rev`

```
(pequal* (rev x)
         (if (consp x)
             (app (rev (cdr x))
                  (list (car x)))
           nil))
```

We say an n-tuple is a true list of length n, and the function `tuplep` may be used to determine if its argument is an n-tuple.

Definition 12: `tuplep`

```
(pequal* (tuplep n x)
         (if (zp n)
             (equal x nil)
           (and (consp x)
                (tuplep (- n 1) (cdr x)))))
```

Another commonly useful data structure in functional programming is the association list, where associations of keys with values are represented using a list of (*key* . *value*) pairs. Given equal-length lists $x = (x_1 \ \ldots \ x_n)$ and $y = (y_1 \ \ldots \ y_n)$, the function `pair-lists` creates an association list where each x_i is associated with the corresponding y_i. When x is shorter than y, the extra elements of y are ignored, and when x is longer than y, the extra elements of x are paired with NIL.

Definition 13: `pair-lists`

```
(pequal* (pair-lists x y)
         (if (consp x)
             (cons (cons (car x) (car y))
                   (pair-lists (cdr x) (cdr y)))
           nil))
```

Finally, the function lookup retrieves the first (*key* . *value*) pair from the association list, x, whose key is a, or returns NIL if there is no such pair. Since not every object $x \in \mathbb{U}$ is a well-formed association list, we interpret x as follows. First, following the list-fix convention, we treat x as a list of elements. Then, we interpret each element of x, say e, as a pair, by following the *cons-fix convention*: if e is already a cons, it is interpreted as itself; otherwise it is interpreted as (NIL . NIL). This same convention is followed by car and cdr.

Definition 14: lookup

```
(pequal* (lookup 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 a (cdr x)))
           nil))
```

3.2 Terms

We now develop a way to represent terms. Recall from page 22 that terms are a subset of the token trees, and that token trees are defined recursively as the closure of the numeric and symbolic tokens under ordered pairing. Since there is a numeric token for each member of $\mathbb{N}$ and a symbolic token for each member of $\mathbb{S}$, there is a natural isomorphism which relates the token trees to $\mathbb{U}$. This isomorphism provides a straightforward way to represent terms as objects: to represent any particular term, x, we simply use the object to which x corresponds.

Recall that the variables are any symbolic tokens except for t and nil. The function logic.variablep determines whether some object of $\mathbb{U}$ is the representation of a variable. The "logic." prefix is only used as a naming convention to indicate this function deals with something from our logic, and has no other special significance.

Definition 15: `logic.variablep`

```
(pequal* (logic.variablep x)
         (and (symbolp x)
              (not (equal x t))
              (not (equal x nil))))
```

We can also determine if every element of a list is a variable, using the function `logic.variable-listp`. We again respect the list-fix convention by not requiring that the list is properly terminated with NIL.

Definition 16: `logic.variable-listp`

```
(pequal* (logic.variable-listp x)
         (if (consp x)
             (and (logic.variablep (car x))
                  (logic.variable-listp (cdr x)))
           t))
```

Recall that for every token tree, a, we have a constant, `(quote` a`)`. The function `logic.constantp` determines if some object, $x \in \mathbb{U}$, is the representation of a constant; the function `logic.constant-listp` determines if every element of a list is a constant.

Definition 17: `logic.constantp`

```
(pequal* (logic.constantp x)
         (and (tuplep 2 x)
              (equal (first x) 'quote)))
```

Definition 18: `logic.constant-listp`

```
(pequal* (logic.constant-listp x)
         (if (consp x)
             (and (logic.constantp (car x))
                  (logic.constant-listp (cdr x)))
              t))
```

Recall that function names are symbolic tokens besides `nil`, `quote`, `pequal*`, `pnot*`, `por*`, `first`, `second`, `third`, `fourth`, `fifth`, `and`, `or`, `list`, `cond`, `let`, and `let*`. The function `logic.function-namep` determines if an object in $\mathbb{U}$ is the representation of a function name.

Definition 19: `logic.function-namep`

```
(pequal* (logic.function-namep x)
         (and (symbolp x)
              (not (memberp x '(nil quote pequal* pnot* por* first
                                 second third fourth fifth and or
                                 list cond let let*))))))
```

Next, we will develop a way to compute the free variables of a term. This is somewhat tricky. To determine the free variables of a function application or lambda abbreviation, we must compute the free variables of the arguments and then union them all together. But this means we must simultaneously introduce a way to gather the free variables from a term, and a way to gather the free variables from a list of terms. We can accomplish this with the standard *flag function* approach. Such a function uses an additional argument, typically called the flag, to specify a mode of operation. Our function, `logic.flag-term-vars`, can operate in two modes: when the flag is TERM, it gathers the free variables from a term, and otherwise it gathers the free variables from a list of terms. For efficient execution on Lisp systems, the function is also written in a tail-recursive style using an accumulator.

Definition 20: `logic.flag-term-vars`

```
(pequal* (logic.flag-term-vars flag x acc)
         (if (equal flag 'term)
             (cond ((logic.constantp x) acc)
                   ((logic.variablep x) (cons x acc))
                   ((not (consp x)) acc)
                   (t
                    (logic.flag-term-vars 'list (cdr x) acc)))
           (if (consp x)
               (logic.flag-term-vars 'term (car x)
                (logic.flag-term-vars 'list (cdr x) acc))
             acc)))
```

With the flag function in place, we introduce a simple wrapper, `logic.-term-vars`, which computes the free variables of a term without needing a flag or accumulator as arguments.

Definition 21: `logic.term-vars`

```
(pequal* (logic.term-vars x)
         (logic.flag-term-vars 'term x nil))
```

We use another flag function, `logic.flag-termp`, which determines either (1) when an object in $\mathbb{U}$ represents a term, or (2) when an object in $\mathbb{U}$ represents a list of terms, depending upon the mode of operation specified by its flag.

Definition 22: `logic.flag-termp`

```
(pequal*
 (logic.flag-termp flag x)
 (if (equal flag 'term)
     (or (logic.variablep x)
         (logic.constantp x)
         (and (consp x)
              (if (logic.function-namep (car x))
                  (let ((args (cdr x)))
                    (and (true-listp args)
                         (logic.flag-termp 'list args)))
                (and (tuplep 3 (car x))
                     (let ((lambda-symbol (first (car x)))
                           (formals       (second (car x)))
```

```
                                (body          (third (car x)))
                                (actuals       (cdr x)))
                           (and (equal lambda-symbol 'lambda)
                                (true-listp formals)
                                (logic.variable-listp formals)
                                (uniquep formals)
                                (logic.flag-termp 'term body)
                                (subsetp (logic.term-vars body) formals)
                                (equal (len formals) (len actuals))
                                (true-listp actuals)
                                (logic.flag-termp 'list actuals)))))))
    (if (consp x)
        (and (logic.flag-termp 'term (car x))
             (logic.flag-termp 'list (cdr x)))
      t)))
```

We also introduce another wrapper function, `logic.termp`, which determines if its argument is a term without needing a flag parameter.

Definition 23: `logic.termp`

```
(pequal* (logic.termp x)
         (logic.flag-termp 'term x))
```

For readability and for reasoning, it is useful to define functions to construct and inspect terms. We can retrieve the value of a constant using `logic.unquote`, and the values from a list of constants using `logic.unquote-list`.

Definition 24: `logic.unquote`

```
(pequal* (logic.unquote x)
         (second x))
```

Definition 25: `logic.unquote-list`

```
(pequal* (logic.unquote-list x)
         (if (consp x)
             (cons (logic.unquote (car x))
                   (logic.unquote-list (cdr x)))
           nil))
```

Given a term, `logic.functionp` determines if it is a function application. We can access the name and arguments of a function application term using `logic.function-name` and `logic.function-args`, respectively. Finally, we may construct a function application from a function name and a list of arguments via `logic.function`.

Definition 26: `logic.functionp`

```
(pequal* (logic.functionp x)
         (logic.function-namep (car x)))
```

Definition 27: `logic.function-name`

```
(pequal* (logic.function-name x)
         (car x))
```

Definition 28: `logic.function-args`

```
(pequal* (logic.function-args x)
         (cdr x))
```

Definition 29: `logic.function`

```
(pequal* (logic.function name args)
         (cons name args))
```

Similarly, given a term, `logic.lambdap` determines if it is a lambda abbreviation. We can access the formal parameters, body, and actuals from a lambda abbreviation using `logic.lambda-formals`, `logic.lambda-body`, and `logic.lambda--actuals`, respectively. Finally, we can construct a lambda abbreviation from a list of formals, a body, and a list of actuals using the `logic.lambda`.

Definition 30: `logic.lambdap`
```
(pequal* (logic.lambdap x)
         (consp (car x)))
```

Definition 31: `logic.lambda-formals`
```
(pequal* (logic.lambda-formals x)
         (second (car x)))
```

Definition 32: `logic.lambda-body`
```
(pequal* (logic.lambda-body x)
         (third (car x)))
```

Definition 33: `logic.lambda-actuals`
```
(pequal* (logic.lambda-actuals x)
         (cdr x))
```

Definition 34: `logic.lambda`
```
(pequal* (logic.lambda xs b ts)
         (cons (list 'lambda xs b) ts))
```

We also need a way to determine if terms are well-formed with respect to an arity table. We represent arity tables as association lists whose keys are function names and whose values are the corresponding arities. We then use a flag function to determine (1) whether a term is well-formed with respect to an arity table, or (2) whether a list of terms are all well-formed with respect to an arity table.

Definition 35: `logic.flag-term-atblp`
```
(pequal*
 (logic.flag-term-atblp flag x atbl)
 (if (equal flag 'term)
     (cond ((logic.constantp x) t)
           ((logic.variablep x) t)
           ((logic.functionp x)
            (let ((name (logic.function-name x))
                  (args (logic.function-args x)))
              (and (equal (len args) (cdr (lookup name atbl)))
                   (logic.flag-term-atblp 'list args atbl))))
           ((logic.lambdap x)
            (let ((body (logic.lambda-body x))
                  (actuals (logic.lambda-actuals x)))
              (and (logic.flag-term-atblp 'term body atbl)
                   (logic.flag-term-atblp 'list actuals atbl))))
           (t nil))
   (if (consp x)
       (and (logic.flag-term-atblp 'term (car x) atbl)
            (logic.flag-term-atblp 'list (cdr x) atbl))
     t)))
```

As usual, we introduce a wrapper, `logic.term-atblp`, which can determine if a term is well-formed with respect to an arity table without a flag parameter.

Definition 36: `logic.term-atblp`

```
(pequal* (logic.term-atblp x atbl)
         (logic.flag-term-atblp 'term x atbl))
```

3.3 Formulas

Formulas, like terms, are certain token trees, and we again use the isomorphism between token trees and $\mathbb{U}$ to represent formulas. The function `logic.formulap` determines whether some object in $\mathbb{U}$ represents a formula.

Definition 37: `logic.formulap`

```
(pequal* (logic.formulap x)
         (cond ((equal (first x) 'pequal*)
                (and (tuplep 3 x)
                     (logic.termp (second x))
                     (logic.termp (third x))))
               ((equal (first x) 'pnot*)
                (and (tuplep 2 x)
                     (logic.formulap (second x))))
               ((equal (first x) 'por*)
                (and (tuplep 3 x)
                     (logic.formulap (second x))
                     (logic.formulap (third x))))
               (t nil)))
```

We can determine if every element of a list represents a formula using the function `logic.formula-listp`.

Definition 38: `logic.formula-listp`

```
(pequal* (logic.formula-listp x)
         (if (consp x)
             (and (logic.formulap (car x))
                  (logic.formula-listp (cdr x)))
           t))
```

We provide some simple accessors for inspecting formulas. Given a formula, the function `logic.fmtype` determines its type, returning PEQUAL*, PNOT*, or POR*. Given an equality, `(pequal*` *lhs* *rhs*`)`, we can retrieve the *lhs* and *rhs* using `logic.=lhs` and `logic.=rhs`, respectively. Given a negation, `(pnot*` *arg*`)`, we can retrieve *arg* using `logic.~arg`. Finally, given a disjunction, `(por*` *lhs* *rhs*`)`, we can obtain the *lhs* and *rhs* using `logic.vlhs` and `logic.vrhs`. (These odd names are intended to convey the type of the formula being accessed. That is, the `=` character in `logic.=lhs` and `logic.=rhs` indicates that we are accessing the left or right hand sides of an equality, the ∼ in `logic.~arg` is intended to suggest that we are accessing the argument of a negation, and the `v` in `logic.vlhs` and `logic.vrhs` is intended as an ASCII approximation of the conventional disjunction symbol, $\vee$.)

Definition 39: `logic.fmtype`

```
(pequal* (logic.fmtype x)
         (first x))
```

Definition 40: `logic.=lhs`

```
(pequal* (logic.=lhs x)
         (second x))
```

Definition 41: `logic.=rhs`

```
(pequal* (logic.=rhs x)
         (third x))
```

Definition 42: `logic.∼arg`

```
(pequal* (logic.∼arg x)
         (second x))
```

Definition 43: `logic.vlhs`

```
(pequal* (logic.vlhs x)
         (second x))
```

Definition 44: `logic.vrhs`

```
(pequal* (logic.vrhs x)
         (third x))
```

Similarly, we provide some simple functions for constructing formulas. Given terms a and b, `logic.pequal` constructs the equality `(pequal*` a b`)`. Given a formula a, `logic.pnot` constructs the negation `(pnot*` a`)`. Given formula a and b, `logic.por` constructs the disjunction `(por*` a b`)`.

Definition 45: `logic.pequal`

```
(pequal* (logic.pequal a b)
         (list 'pequal* a b))
```

Definition 46: `logic.pnot`

```
(pequal* (logic.pnot a)
         (list 'pnot* a))
```

Definition 47: `logic.por`

```
(pequal* (logic.por a b)
         (list 'por* a b))
```

We can determine whether a formula is well-formed with respect to an arity table using the function `logic.formula-atblp`.

Definition 48: `logic.formula-atblp`

```
(pequal* (logic.formula-atblp x atbl)
         (let ((type (logic.fmtype x)))
           (cond ((equal type 'por*)
                  (and (logic.formula-atblp (logic.vlhs x) atbl)
                       (logic.formula-atblp (logic.vrhs x) atbl)))
                 ((equal type 'pnot*)
                  (logic.formula-atblp (logic.~arg x) atbl))
                 ((equal type 'pequal*)
                  (and (logic.term-atblp (logic.=lhs x) atbl)
                       (logic.term-atblp (logic.=rhs x) atbl)))
                 (t nil))))
```

Finally, given a non-empty list of formulas $x_1, \ldots, x_n$, we can create the right-associative disjunction, `(por*` x_1 `(por*` $\ldots$`(por*` x_{n-1} x_n`)`$\ldots$`))`, using the function `logic.disjoin-formulas`.

Definition 49: `logic.disjoin-formulas`

```
(pequal* (logic.disjoin-formulas x)
         (if (consp x)
             (if (consp (cdr x))
                 (logic.por (car x)
                            (logic.disjoin-formulas (cdr x)))
```

```
            (car x))
        nil))
```

3.4 Appeals

We now turn our attention to representing and checking proofs. Each step in a proof will be represented by an *appeal*. An appeal is either a 2-tuple, 3-tuple, or 4-tuple of the form

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

where the brackets indicate the *subproofs* and *extras* may be omitted when they are not needed. The *method* of an appeal is a symbol which indicates which rule of inference is being used in this proof step, the *conclusion* is the formula being proven by this step, the *subproofs* are a list of appeals which should justify any premises needed by this rule, and the *extras* are any additional, non-proof information being used by this step. For instance, in an instantiation step the *extras* will contain the substitution list being used.

The flag function `logic.flag-appealp` may be used to determine if an object is (1) a valid appeal, or (2) a valid list of appeals, based upon the mode of operation specified by its flag parameter. We also provide two wrapper functions, `logic.appealp` and `logic.appeal-listp`, which determine if their argument is a valid appeal or list of appeals, respectively, without a flag.

Definition 50: `logic.flag-appealp`
```
(pequal* (logic.flag-appealp flag x)
         (if (equal flag 'proof)
             (and (true-listp x)
                  (<= (len x) 4)
                  (symbolp (first x))
```

```
                (logic.formulap (second x))
                (true-listp (third x))
                (logic.flag-appealp 'list (third x)))
         (if (consp x)
             (and (logic.flag-appealp 'proof (car x))
                  (logic.flag-appealp 'list (cdr x)))
           t)))
```

Definition 51: `logic.appealp`

```
(pequal* (logic.appealp x)
         (logic.flag-appealp 'proof x))
```

Definition 52: `logic.appeal-listp`

```
(pequal* (logic.appeal-listp x)
         (logic.flag-appealp 'list x))
```

We also introduce simple accessors for appeals. Note that `logic.subproofs` and `logic.extras` will return NIL when these components have been omitted from an appeal.

Definition 53: `logic.method`

```
(pequal* (logic.method x)
         (first x))
```

Definition 54: `logic.conclusion`

```
(pequal* (logic.conclusion x)
         (second x))
```

Definition 55: `logic.subproofs`

```
(pequal* (logic.subproofs x)
         (third x))
```

Definition 56: `logic.extras`

```
(pequal* (logic.extras x)
         (fourth x))
```

Given a list of appeals, the function `logic.strip-conclusions` can be used to extract a list of their conclusions.

Definition 57: `logic.strip-conclusions`

```
(pequal* (logic.strip-conclusions x)
         (if (consp x)
             (cons (logic.conclusion (car x))
                   (logic.strip-conclusions (cdr x)))
           nil))
```

3.5 Step Checking

We now introduce several functions which determine whether an appeal is a valid proof step according to the rules of our logic. Each step-checker is also responsible for ensuring that its conclusion is well-formed with respect to an arity table, given that the conclusions of its subproofs are similarly well-formed.

To begin, the function `logic.axiom-okp` checks whether an appeal is a valid use of an axiom. Since the set of axioms may grow as we extend the history, it takes a list of the current axioms as a parameter. We say an appeal is a valid use of an axiom when its method is AXIOM, its conclusion is among the given axioms, and it has no subproofs or extras. We explicitly check to ensure that the formula is well-formed with respect to an arity table.

Definition 58: `logic.axiom-okp`

```
(pequal* (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))))
```

The function `logic.theorem-okp` is quite similar, and checks whether an appeal is the valid use of a theorem. Such an appeal is valid when its method is THEOREM, its conclusion is in the list of theorems, it has no subproofs or extras, and its conclusion is well-formed with respect to the arity table.

Definition 59: `logic.theorem-okp`

```
(pequal* (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))))
```

Recall from page 25 the associativity rule of inference, which allows us to derive `(por* (por*` A B`)` C`)` from the premise `(por*` A `(por*` B C`))`. We say an appeal is a valid use of this rule when its method is ASSOCIATIVITY, it has a single subproof which is appropriately related to its conclusion, and it has no extras. The function `logic.associativity-okp` determines if an appeal satisfies these conditions. We do not check the formula against an arity table, since if the subproof's conclusion is well-formed, then the new conclusion must also be well-formed.

Definition 60: `logic.associativity-okp`

```
(pequal*
 (logic.associativity-okp x)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'associativity)
        (equal extras nil)
        (tuplep 1 subproofs)
        (let ((sub-or-a-b-c (logic.conclusion (first subproofs))))
          (and (equal (logic.fmtype conclusion) 'por*)
               (equal (logic.fmtype sub-or-a-b-c) 'por*)
               (let ((conc-or-a-b (logic.vlhs conclusion))
                     (conc-c      (logic.vrhs conclusion))
                     (sub-a       (logic.vlhs sub-or-a-b-c))
                     (sub-or-b-c  (logic.vrhs sub-or-a-b-c)))
                 (and (equal (logic.fmtype conc-or-a-b) 'por*)
                      (equal (logic.fmtype sub-or-b-c) 'por*)
                      (let ((conc-a (logic.vlhs conc-or-a-b))
                            (conc-b (logic.vrhs conc-or-a-b))
                            (sub-b  (logic.vlhs sub-or-b-c))
                            (sub-c  (logic.vrhs sub-or-b-c)))
                        (and (equal conc-a sub-a)
                             (equal conc-b sub-b)
                             (equal conc-c sub-c))))))))))
```

Recall from page 25 the contraction rule, which allows us to derive A from the premise `(por*` A A`)`. We say an appeal is a valid use of this rule when its method is CONTRACTION, it has a single subproof of the form `(por*` A A`)`, its conclusion is A, and it has no extras. The function `logic.contraction-okp` checks whether an appeal satisfies these conditions. As for associativity steps, no arity checking is needed for contraction steps.

Definition 61: `logic.contraction-okp`

```
(pequal* (logic.contraction-okp x)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'contraction)
                (equal extras nil)
                (tuplep 1 subproofs)
                (let ((or-a-a (logic.conclusion (first subproofs))))
                  (and (equal (logic.fmtype or-a-a) 'por*)
                       (equal (logic.vlhs or-a-a) conclusion)
                       (equal (logic.vrhs or-a-a) conclusion))))))
```

Recall from page 25 the cut rule, which allows us to conclude (por* B C) from proofs of the premises (por* A B) and (por* (pnot* A) C). An appeal is valid according to this rule when its method is CUT, it has two subproofs which match its conclusion in the appropriate way, and it has no extras. Again, no arity checking is needed.

Definition 62: `logic.cut-okp`

```
(pequal*
 (logic.cut-okp x)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'cut)
        (equal extras nil)
        (tuplep 2 subproofs)
        (let ((or-a-b     (logic.conclusion (first subproofs)))
              (or-not-a-c (logic.conclusion (second subproofs))))
          (and (equal (logic.fmtype or-a-b) 'por*)
               (equal (logic.fmtype or-not-a-c) 'por*)
               (let ((a     (logic.vlhs or-a-b))
                     (b     (logic.vrhs or-a-b))
                     (not-a (logic.vlhs or-not-a-c))
                     (c     (logic.vrhs or-not-a-c)))
                 (and (equal (logic.fmtype not-a) 'pnot*)
                      (equal (logic.~arg not-a) a)
                      (equal (logic.fmtype conclusion) 'por*)
                      (equal (logic.vlhs conclusion) b)
                      (equal (logic.vrhs conclusion) c))))))))
```

Recall from page 25 the expansion rule, which given the premise B allows us to derive (por* A B). An appeal is valid under this rule when its method is EXPANSION, it has a single subproof which is appropriately related to its conclusion, and it has no extras. Since the A portion of the conclusion is new, we check to ensure it is well-formed with respect to the arity table.

Definition 63: `logic.expansion-okp`

```
(pequal* (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)
                (let ((b (logic.conclusion (first subproofs))))
                  (and (equal (logic.fmtype conclusion) 'por*)
                       (equal (logic.vrhs conclusion) b)
                       (logic.formula-atblp (logic.vlhs conclusion)
                                            atbl))))))
```

Recall from page 25 the propositional schema, which has no premises and allows us to derive (por* (pnot* A) A). An appeal is a a valid use of this rule when its method is PROPOSITIONAL-SCHEMA, its conclusion has the proper shape, and it has no subproofs or extras. We also check that A is well-formed with respect to the arity table.

Definition 64: `logic.propositional-schema-okp`

```
(pequal* (logic.propositional-schema-okp x atbl)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'propositional-schema)
                (equal subproofs nil)
                (equal extras nil)
                (equal (logic.fmtype conclusion) 'por*)
                (let ((not-a (logic.vlhs conclusion))
                      (a     (logic.vrhs conclusion)))
                  (and (equal (logic.fmtype not-a) 'pnot*)
                       (equal (logic.~arg not-a) a)
                       (logic.formula-atblp a atbl))))))
```

Recall from page 25 the functional equality rule, which allows us to derive, from no premises, a formula of the form

```
(por* (pnot* (pequal* t₁ s₁))
      (por* (pnot* (pequal* t₂ s₂))
            ⋱
              (por* (pnot* (pequal* tₙ sₙ))
                    (pequal* (f t₁ t₂ ... tₙ)
                             (f s₁ s₂ ... sₙ))) ... )).
```

where f is a function name, $t_1, \ldots, t_n$ and $s_1, \ldots, s_n$ are terms.

We use the function `logic.check-functional-axiom` to determine if a formulas has this shape. We process one `(pnot* (pequal*` t_i s_i`))` term at each step. That is, if we find a `por*`-type formula, its left-hand side must be of the form `(pnot* (pequal*` t_i s_i`))`; if so, we record the particular terms, t_i and s_i, which have been encountered, and recursively check the right-hand side. Eventually, we must reach a `pequal*`-type formula, which must be

`(pequal* (`f t_1 ... t_n`) (`f s_1 ... s_n`))`,

where the t_i and s_i are the terms we have recorded along the way.

Definition 65: `logic.check-functional-axiom`

```
(pequal*
 (logic.check-functional-axiom x ti si)
 (if (equal (logic.fmtype x) 'pequal*)
     (and (logic.functionp (logic.=lhs x))
          (logic.functionp (logic.=rhs x))
          (equal (logic.function-name (logic.=lhs x))
                 (logic.function-name (logic.=rhs x)))
          (equal (logic.function-args (logic.=lhs x)) (rev ti))
          (equal (logic.function-args (logic.=rhs x)) (rev si)))
   (and (equal (logic.fmtype x) 'por*)
        (equal (logic.fmtype (logic.vlhs x)) 'pnot*)
        (equal (logic.fmtype (logic.~arg (logic.vlhs x))) 'pequal*)
        (logic.check-functional-axiom
         (logic.vrhs x)
         (cons (logic.=lhs (logic.~arg (logic.vlhs x))) ti)
         (cons (logic.=rhs (logic.~arg (logic.vlhs x))) si)))))
```

An appeal is a valid use of the functional equality rule when its method is FUNCTIONAL-EQUALITY, its conclusion has the proper shape, has no subproofs or extras, and has the proper arity.

Definition 66: `logic.functional-equality-okp`

```
(pequal* (logic.functional-equality-okp x)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
```

```
           (and (equal method 'functional-equality)
                (equal subproofs nil)
                (equal extras nil)
                (logic.check-functional-axiom conclusion nil nil)
                (logic.formula-atblp conclusion atbl))))
```

Before we can check instantiation and β-reduction steps, we will need to define substitution. We represent substitution lists using association lists whose keys are variables and whose values are terms. We can determine whether some object in $\mathbb{U}$ is a substitution list with `logic.sigmap`, the *sigma predicate*. We can also determine if an object is a list of substitution lists using `logic.sigma-listp`. Finally, we can determine if an object is a list of lists of substitution lists with `logic.sigma-list-listp`.

Definition 67: `logic.sigmap`
```
(pequal* (logic.sigmap x)
         (if (consp x)
             (and (consp (car x))
                  (logic.variablep (car (car x)))
                  (logic.termp (cdr (car x)))
                  (logic.sigmap (cdr x)))
           t))
```

Definition 68: `logic.sigma-listp`
```
(pequal* (logic.sigma-listp x)
         (if (consp x)
             (and (logic.sigmap (car x))
                  (logic.sigma-listp (cdr x)))
           nil))
```

Definition 69: `logic.sigma-list-listp`
```
(pequal* (logic.sigma-list-listp x)
         (if (consp x)
              (and (logic.sigma-listp (car x))
                   (logic.sigma-list-listp (cdr x)))
            nil))
```

The flag function `logic.flag-substitute` may be used to (1) apply a substitution list to a term, or (2) apply a substitution list to a list of terms, depending upon the mode of operation specified by its flag parameter.

Definition 70: `logic.flag-substitute`

```
(pequal*
 (logic.flag-substitute flag x sigma)
 (if (equal flag 'term)
     (cond ((logic.variablep x)
            (if (lookup x sigma)
                (cdr (lookup x sigma))
              x))
           ((logic.constantp x)
            x)
           ((logic.functionp x)
            (let ((fn   (logic.function-name x))
                  (args (logic.function-args x)))
              (logic.function fn (logic.flag-substitute 'list
                                                        args
                                                        sigma))))
           ((logic.lambdap x)
            (let ((formals (logic.lambda-formals x))
                  (body    (logic.lambda-body x))
                  (actuals (logic.lambda-actuals x)))
              (logic.lambda formals body
                            (logic.flag-substitute 'list
                                                   actuals
                                                   sigma))))
           (t nil))
   (if (consp x)
       (cons (logic.flag-substitute 'term (car x) sigma)
             (logic.flag-substitute 'list (cdr x) sigma))
```

```
      nil)))
```

As usual, we define a wrapper, `logic.substitute`, which can apply a substitution list to a term without an extra flag parameter. We also define `logic.substitute-list`, which can apply a substitution list to a term list without a flag parameter.

Definition 71: `logic.substitute`

```
(pequal* (logic.substitute x sigma)
         (logic.flag-substitute 'term x sigma))
```

Definition 72: `logic.substitute-list`

```
(pequal* (logic.substitute-list x sigma)
         (logic.flag-substitute 'list x sigma))
```

The function `logic.substitute-formula` extends our substitution operation to formulas.

Definition 73: `logic.substitute-formula`

```
(pequal*
 (logic.substitute-formula formula sigma)
 (let ((type (logic.fmtype formula)))
   (cond ((equal type 'por*)
          (logic.por
           (logic.substitute-formula (logic.vlhs formula) sigma)
           (logic.substitute-formula (logic.vrhs formula) sigma)))
         ((equal type 'pnot*)
          (logic.pnot
           (logic.substitute-formula (logic.~arg formula) sigma)))
         ((equal type 'pequal*)
          (logic.pequal
           (logic.substitute (logic.=lhs formula) sigma)
           (logic.substitute (logic.=rhs formula) sigma)))
         (t nil))))
```

Recall from page 27 the instantiation rule, which allows us to derive A/σ from a proof of A. An appeal is a valid use of the instantiation rule when (1) its method is INSTANTIATION, (2) it has a single subproof, call its conclusion A, (3) its extras are a substitution list, call it σ, and (4) its conclusion is A/σ. We also ensure that the resulting formula is valid with respect to the arity table.

Definition 74: `logic.instantiation-okp`

```
(pequal* (logic.instantiation-okp x)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'instantiation)
                (logic.sigmap extras)
                (tuplep 1 subproofs)
                (equal (logic.substitute-formula
                        (logic.conclusion (first subproofs))
                        extras)
                       conclusion)
                (logic.formula-atblp conclusion atbl))))
```

Recall from page 27 the β-reduction rule, which allows us to derive, from no premises,

$$\texttt{(pequal* ((lambda (}x_1 \ \ldots \ x_n\texttt{)}\ \beta\texttt{)}\ t_1 \ \ldots \ t_n\texttt{)}$$
$$\beta/[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]\texttt{)}.$$

We say an appeal is a valid use of the β-reduction rule when its method is BETA-REDUCTION, its conclusion has the appropriate form, and it has no subproofs or extras.

Definition 75: `logic.beta-reduction-okp`

```
(pequal* (logic.beta-reduction-okp x atbl)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'beta-reduction)
                (equal subproofs nil)
                (equal extras nil)
                (logic.formula-atblp conclusion atbl)
                (equal (logic.fmtype conclusion) 'pequal*)
                (let ((lhs (logic.=lhs conclusion))
                      (rhs (logic.=rhs conclusion)))
                  (and (logic.lambdap lhs)
                       (let ((formals (logic.lambda-formals lhs))
                             (body    (logic.lambda-body lhs))
                             (actuals (logic.lambda-actuals lhs)))
                         (equal (logic.substitute
                                 body
                                 (pair-lists formals actuals))
                                rhs)))))))
```

We introduce the arity table for our primitive functions with a zero-ary function, `logic.initial-arity-table`.

Definition 76: `logic.initial-arity-table`

```
(pequal* (logic.initial-arity-table)
         '((if . 3)
           (equal . 2)
           (consp . 1)
           (cons . 2)
           (car . 1)
           (cdr . 1)
           (symbolp . 1)
           (symbol-< . 2)
           (natp . 1)
           (< . 2)
           (+ . 2)
           (- . 2)))
```

Recall from page 41 the base evaluation rule, which allows us to evaluate the application of a primitive function on constants. To determine if a term has this form, we use the function `logic.base-evaluablep`.

Definition 77: `logic.base-evaluablep`

```
(pequal*
 (logic.base-evaluablep x)
 (and (logic.functionp x)
      (let ((fn   (logic.function-name x))
            (args (logic.function-args x)))
        (let ((entry (lookup fn (logic.initial-arity-table))))
          (and entry
               (logic.constant-listp args)
               (tuplep (cdr entry) args))))))
```

Given a base-evaluable term, the function `logic.base-evaluator` produces the constant which it evaluates to.

Definition 78: `logic.base-evaluator`

```
(pequal* (logic.base-evaluator x)
         (let ((fn   (logic.function-name x))
               (vals (logic.unquote-list (logic.function-args x))))
           (list 'quote
                 (cond ((equal fn 'if)
                        (if (first vals)
                            (second vals)
                          (third vals)))
                       ((equal fn 'equal)
                        (equal (first vals) (second vals)))
                       ((equal fn 'consp)
                        (consp (first vals)))
                       ((equal fn 'cons)
                        (cons (first vals) (second vals)))
                       ((equal fn 'car)
                        (car (first vals)))
                       ((equal fn 'cdr)
                        (cdr (first vals)))
                       ((equal fn 'symbolp)
                        (symbolp (first vals)))
                       ((equal fn 'symbol-<)
                        (symbol-< (first vals) (second vals)))
                       ((equal fn 'natp)
                        (natp (first vals)))
                       ((equal fn '<)
                        (< (first vals) (second vals)))
                       ((equal fn '+)
                        (+ (first vals) (second vals)))
                       ((equal fn '-)
                        (- (first vals) (second vals)))))))
```

Finally, `logic.base-eval-okp` determines if an appeal is a valid use of the base evaluation rule: the method must be BASE-EVAL, the conclusion must have the form `(pequal*` *lhs rhs*`)` where *lhs* is a base-evaluable term which evaluates to *rhs*, and there must be no subproofs or extras. We also ensure that the *lhs* is well-formed with respect to the current arity table; there is no need to check the *rhs* since it is a constant.

Definition 79: `logic.base-eval-okp`

```
(pequal* (logic.base-eval-okp x atbl)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'base-eval)
                (equal subproofs nil)
                (equal extras nil)
                (equal (logic.fmtype conclusion) 'pequal*)
                (let ((lhs (logic.=lhs conclusion))
                      (rhs (logic.=rhs conclusion)))
```

```
                    (and (logic.base-evaluablep lhs)
                         (equal (logic.base-evaluator lhs) rhs)
                         (logic.term-atblp lhs atbl))))))
```

We now review the induction rule from page 56. Suppose m is a term, $q_1, \ldots, q_k$ are formulas, and for each $i = 1 \ldots k$ we have a set of substitution lists, $\Sigma_i = \{\sigma_{\langle i,1\rangle}, \ldots, \sigma_{\langle i,h_i\rangle}\}$. Then, we may derive the formula F given proofs of the

basis step,
(por* F (por* q_1 (... (por* q_{k-1} q_k) ...)),

inductive steps, for $i = 1 \ldots k$,

```
(por* F
      (por* (pnot* q_i)
            (por* (pnot* F/σ⟨i,1⟩)
                  ⋱
                     (por* (pnot* F/σ⟨i,h_i−1⟩)
                           (pnot* F/σ⟨i,h_i⟩)) ...))),
```

ordinal step,
(pequal* (ordp m) t), and

measure steps, for $i = 1 \ldots k$, $j = 1 \ldots h_i$,
(por* (pnot* q_i) (pequal* (ord< $m/\sigma_{\langle i,j\rangle}$ m) t)).

To check whether an appeal is a valid use of the induction rule, we will need to ensure that its subproofs establish each of these obligations. The function logic.make-basis-step creates the formula required for the basis step, given the formula F and the list of formulas (q_1 ... q_k).

Definition 80: logic.make-basis-step

```
(pequal* (logic.make-basis-step f qs)
         (logic.disjoin-formulas (cons f qs)))
```

For the induction steps, we begin with an auxiliary function. Given a formula, F, and a list of substitutions, (σ_1 ... σ_n), logic.substitute-each-sigma-into-formula produces the list of formulas (F/σ_1 ... F/σ_n).

Definition 81: `logic.substitute-each-sigma-into-formula`

```
(pequal*
 (logic.substitute-each-sigma-into-formula f x)
 (if (consp x)
     (cons (logic.substitute-formula f (car x))
           (logic.substitute-each-sigma-into-formula f (cdr x)))
   nil))
```

We use this in `logic.make-induction-step`, which creates the induction step for a particular i when given the formula F, the formula q_i, and the corresponding list of substitution lists, $(\sigma_{\langle i,1\rangle} \ \ldots \ \sigma_{\langle i,h_i\rangle})$.

Definition 82: `logic.make-induction-step`

```
(pequal* (logic.make-induction-step f q-i sigmas-i)
         (logic.disjoin-formulas
          (cons f (cons (logic.pnot q-i)
                        (logic.substitute-each-sigma-into-formula
                         (logic.pnot f)
                         sigmas-i)))))
```

Finally, `logic.make-induction-steps` forms the list of all required induction steps when given the list of formulas $(q_1 \ \ldots \ q_k)$ and the list of lists of substitution lists, $(\Sigma_1 \ \ldots \ \Sigma_k)$, where each Σ_i is the list $(\sigma_{\langle i,1\rangle} \ \ldots \ \sigma_{\langle i,h_i\rangle})$.

Definition 83: `logic.make-induction-steps`

```
(pequal* (logic.make-induction-steps f qs all-sigmas)
         (if (consp qs)
             (cons (logic.make-induction-step f
                                              (car qs)
                                              (car all-sigmas))
                   (logic.make-induction-steps f
                                               (cdr qs)
                                               (cdr all-sigmas)))
           nil))
```

The ordinal step is simpler to construct; given the term m, the function `logic.make-ordinal-step` produces it.

Definition 84: `logic.make-ordinal-step`

```
(pequal* (logic.make-ordinal-step m)
         (logic.pequal (logic.function 'ordp (list m)) ''t))
```

For the measure steps, we begin with `logic.make-measure-step`, which constructs the measure step for a particular i and j, given the term m, the formula q_i, and the substitution list $\sigma_{\langle i,j \rangle}$.

Definition 85: `logic.make-measure-step`

```
(pequal* (logic.make-measure-step m q-i sigma-i-j)
         (logic.por
          (logic.pnot q-i)
          (logic.pequal
           (logic.function 'ord<
                           (list (logic.substitute m sigma-i-j) m))
           ''t)))
```

The function `logic.make-measure-steps` extends this to construct all of the measure steps for a particular i, given the term m, the formula q_i, and the list of substitution lists $(\sigma_{\langle i,1 \rangle} \ \ldots \ \sigma_{\langle i,h_i \rangle})$.

Definition 86: `logic.make-measure-steps`

```
(pequal* (logic.make-measure-steps m q-i sigmas-i)
         (if (consp sigmas-i)
             (cons (logic.make-measure-step m q-i (car sigmas-i))
                   (logic.make-measure-steps m q-i (cdr sigmas-i)))
           nil))
```

Finally, `logic.make-all-measure-steps` constructs all the measure steps for all i and j, given the term m, the list of formulas $(q_1 \ \ldots \ q_k)$, and the list of lists of substitution lists, $(\Sigma_1 \ \ldots \ \Sigma_k)$, where each Σ_i is the list $(\sigma_{\langle i,1\rangle} \ \ldots \ \sigma_{\langle i,h_i\rangle})$.

Definition 87: `logic.make-all-measure-steps`

```
(pequal* (logic.make-all-measure-steps m qs all-sigmas)
         (if (consp all-sigmas)
             (app (logic.make-measure-steps m
                                            (car qs)
                                            (car all-sigmas))
                  (logic.make-all-measure-steps m
                                                (cdr qs)
                                                (cdr all-sigmas)))
           nil))
```

We are now ready to introduce `logic.induction-okp`, which checks whether an appeal is a valid use of the induction rule. The method must be INDUCTION, and the extras are expected to be a three-tuple containing the term m, the list of formulas $qs = (q_1 \ \ldots \ q_k)$, and the list of lists of substitution lists, $\mathit{all\text{-}sigmas} = (\Sigma_1 \ \ldots \ \Sigma_k)$, where each Σ_i is the list $(\sigma_{\langle i,1\rangle} \ \ldots \ \sigma_{\langle i,h_i\rangle})$. Then, taking the conclusion as the formula F, the subproofs must include the basis step, induction steps, ordinal step, and measure steps. There is no need to check the arity of the conclusion since it occurs in the conclusions of the basis and inductive steps.

Definition 88: `logic.induction-okp`

```
(pequal*
 (logic.induction-okp x)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'induction)
        (tuplep 3 extras)
        (let ((m          (first extras))
              (qs         (second extras))
              (all-sigmas (third extras))
              (subconcs   (logic.strip-conclusions subproofs)))
          (and (logic.termp m)
               (logic.formula-listp qs)
               (logic.sigma-list-listp all-sigmas)
               (equal (len qs) (len all-sigmas))
               (memberp (logic.make-basis-step conclusion qs)
                        subconcs)
               (subsetp (logic.make-induction-steps conclusion
                                                    qs
                                                    all-sigmas)
                        subconcs)
               (memberp (logic.make-ordinal-step m) subconcs)
               (subsetp (logic.make-all-measure-steps m
                                                      qs
                                                      all-sigmas)
                        subconcs))))))
```

3.6 Proof Checking

Now that we have functions for checking each kind of proof step, we can create a function to check "any step." That is, logic.appeal-step-okp checks whether its argument, x, is a valid step, with respect to a list of axioms and a list of theorems.

Definition 89: logic.appeal-step-okp

```
(pequal* (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 'propositional-schema)
                  (logic.propositional-schema-okp x atbl))
                 ((equal how 'functional-equality)
                  (logic.functional-equality-okp x atbl))
                 ((equal how 'beta-reduction)
                  (logic.beta-reduction-okp x atbl))
                 ((equal how 'expansion)
                  (logic.expansion-okp x atbl))
                 ((equal how 'contraction)
                  (logic.contraction-okp x))
                 ((equal how 'associativity)
                  (logic.associativity-okp x))
                 ((equal how 'cut)
                  (logic.cut-okp x))
                 ((equal how 'instantiation)
                  (logic.instantiation-okp x atbl))
                 ((equal how 'induction)
                  (logic.induction-okp x))
                 ((equal how 'base-eval)
                  (logic.base-eval-okp x atbl))
                 (t nil))))
```

Our proof checker, `logic.proofp`, is formed by extending this single-step checking function across the proof, recursively. Since we need to check both proofs and lists of proofs, we use a flag function and, as usual, introduce a wrapper to hide the flag.

Definition 90: `logic.flag-proofp`

```
(pequal*
 (logic.flag-proofp flag x axioms thms atbl)
 (if (equal flag 'proof)
     (and (logic.appeal-step-okp x axioms thms atbl)
          (logic.flag-proofp 'list (logic.subproofs x)
                             axioms thms atbl))
   (if (consp x)
       (and (logic.flag-proofp 'proof (car x) axioms thms atbl)
            (logic.flag-proofp 'list (cdr x) axioms thms atbl))
```

```
      t)))
```

Definition 91: `logic.proofp`

```
(pequal* (logic.proofp x axioms thms atbl)
         (logic.flag-proofp 'proof x axioms thms atbl))
```

Recall that every step checking function ensures that if the conclusions of its subproofs are well-formed with respect to an arity table, then its conclusion is also well-formed. Hence, by induction, when `logic.proofp` accepts a proof, all of the formulas in every conclusion throughout the proof must be well-formed with respect to the arity table.

3.7 Provability

Now that we have a proof checker, we can use existential quantification to decide whether a particular formula is provable. Recall from page 61 the notion of a witnessing (Skolem) function. We begin by introducing a witnessing function, `logic.provable-witness`, whose defining axiom is as follows.

Definition 92: `logic.provable-witness`

```
(por* (pequal* (and (logic.appealp proof)
                    (logic.proofp proof axioms thms atbl)
                    (equal (logic.conclusion proof) x))
               nil)
      (pnot*
       (pequal* ((lambda (proof x axioms thms atbl)
                    (and (logic.appealp proof)
                         (logic.proofp proof axioms thms atbl)
                         (equal (logic.conclusion proof) x)))
                  (logic.provable-witness x axioms thms atbl)
                  x axioms thms atbl)
                nil)))
```

Intuitively, this axiom can be understood as: if there exists an appeal which is a valid proof of x, then `(logic.provable-witness x axioms thms atbl)` is such an appeal. Accordingly, it is straightforward to introduce a function that determines if a formula is provable.

Definition 93: `logic.provablep`

```
(pequal* (logic.provablep x axioms thms atbl)
         (let ((proof (provable-witness x axioms thms atbl)))
           (and (logic.appealp proof)
                (logic.proofp proof axiom thms atbl)
                (equal (logic.conclusion proof) x))))
```

Chapter 4

System Implementation

In this chapter, we introduce a Common Lisp program that allows us to run `logic.proofp` to check proofs. This is the program that will be used to check the proofs of Milawa's fidelity (Section 12.12), and hence it must be trusted if we are to have confidence in Milawa. Because of this, we have kept the program intentionally primitive by the standards of interactive theorem provers: it has no automation for finding proofs, no ability to recover from errors, and no ability to interact with other tools and programs. In practice, the user constructs all of his proofs ahead of time, then submits them to the program to be checked.

We begin by introducing a Common Lisp representation of the Milawa objects, and a way to implement functions in the Milawa logic as Common Lisp functions. We use this mechanism to introduce the Common Lisp counterpart of `logic.proofp`. Our program allows the user to manage an evolving history of events. It keeps track of the current axioms, theorems, and arity table, and allows the user to submit commands to extend the history with admissible events. To check that these events are admissible, our program initially requires the user to provide proofs of theorems and admission obligations, and initially these proofs must be checked by `logic.proofp`.

To support our bootstrapping process (Chapter 12), the user can also instruct the system to begin using a new, higher-level proof checker to check proofs. But to do this, he must first prove the fidelity of his new proof checker, and check this proof with the currently trusted proof checker.

Our use of Lisp features is fairly minimal, but there are still some nuances such as creating packages, configuring the Common Lisp reader, and using hash tables, which are best left to Lisp manuals such as Seibel's *Practical Common Lisp* [81] and Steele's *Common LISP: The Language* [86].

4.1 Milawa Functions as Programs

Common Lisp has a package system which allows the programmer to put symbols into different namespaces. We use packages to keep logical definitions separated from our system functions and also from the functions which are built-in to Common Lisp systems. Our program begins in the predefined CL-USER package, which is a starting place for a Lisp user's code. We will use this package for various functions and objects which are not intended to be accessible from the logic. We also instruct the Lisp compiler to optimize for execution speed.

Lisp Code
```
(in-package "CL-USER")
(declaim (optimize (speed 3) (safety 0) (space 0)))
```

We now create a new package which will be used for all the functions in our logic. This is done with the `defpackage` command, which takes (1) a name for the new package being created, and (2) instructions about which symbols should be imported from other packages. We name this package MILAWA, and the instruction `(:use)` ensures that no symbols are imported into the new package as it is created—think of it as "use nothing."

Lisp Code
```
(defpackage "MILAWA" (:use))
```

Next, we use the import command to bring a few symbols from Common Lisp into the MILAWA package. This causes the MILAWA-package symbols such as MILAWA::quote to become aliases for the COMMON-LISP-package symbols of the same name, e.g., COMMON-LISP::quote. It is particularly important that we import quote so that constants are interpreted correctly. The other symbols have definitions which are compatible with our object representation (which we will discuss momentarily), so it is convenient to import them rather than redefine them.

Lisp Code
```
(import '(COMMON-LISP::nil
          COMMON-LISP::t
          COMMON-LISP::quote
          COMMON-LISP::if
          COMMON-LISP::equal
          COMMON-LISP::consp
          COMMON-LISP::cons
          COMMON-LISP::symbolp
          COMMON-LISP::let
          COMMON-LISP::let*
          COMMON-LISP::list
          COMMON-LISP::and
          COMMON-LISP::or
          COMMON-LISP::cond
          COMMON-LISP::lambda)
        "MILAWA")
```

Finally, the Lisp function find-package takes a package's name and returns a reference to that package. We define a constant, milawa-package, so we can refer to this package many times without having to search for it.

Lisp Code
```
(defconstant milawa-package (find-package "MILAWA"))
```

Many Lisp implementations do not automatically compile functions when they are submitted with `defun`. To ensure our system functions are compiled, we introduce them with a new macro, `defun-comp`.

Lisp Code

```
(defmacro defun-comp (&rest args)
  `(compile (defun ,@args)))
```

How can we represent the objects of $\mathbb{U}$? Common Lisp provides a data type for representing arbitrary-precision integers, so we will use the non-negative Lisp integers to represent $\mathbb{N}$. The Common Lisp function `integerp` determines if its argument is a Lisp integer, and its function `<=` performs a less-than-or-equal comparison, so we can determine if `x` is such an object by writing `(and (integerp x) (<= 0 x))`.

Common Lisp also provides a symbol data type. Each Lisp symbol has a name, which is an ASCII string, and a package which it belongs to. In our logic, the symbols, $\mathbb{S}$, also are named using ASCII strings, but are not organized into any package system. The Common Lisp function `symbolp` determines if its argument is a Lisp symbol, and `symbol-name` returns the name of a symbol as a string.

To represent $\mathbb{S}$, we might try to use the subset of Lisp symbols whose package is `MILAWA`. But because of our earlier import statement, symbols like `MILAWA::quote` are actually aliases to symbols in the `COMMON-LISP` package. To correct for this, we use a slightly different subset of Common Lisp symbols—namely, the symbols which can be referred to by names in the `MILAWA` package. The Common Lisp function `intern` takes a name and a package as arguments, and returns the symbol to which *package::name* refers. For instance, if we intern `"NATP"` into `milawa-package`, we obtain `MILAWA::natp`, but if we intern `"QUOTE"` into `milawa-package` we obtain `COMMON-LISP::quote` because `MILAWA::quote` is just an alias to this symbol. So, to determine if a particular symbol, `x`, is in our desired subset, we can write `(equal x (intern (symbol-name x) milawa-package)))`.

Common Lisp also provides a cons data type for representing ordered pairs. The Lisp function `cons` constructs a pair when given the first and second components as arguments, and `consp` determines if its argument is a pair.

We represent the conses of $\mathbb{U}$ using a subset of Lisp's conses. Why do we not simply use all Lisp conses? One reason is that there are "bad" Lisp objects such as negative integers, characters, arrays, structures, and so on, which can be put into a Lisp cons, yet which do not represent any object in $\mathbb{U}$. Furthermore, Lisp conses can contain circular references which would appear to be "infinite" objects, while every object in $\mathbb{U}$ is finite. Accordingly, we represent the conses of $\mathbb{U}$ with Lisp cons trees which are entirely free of bad objects and circular references.

We can recognize whether a Lisp object is free from bad objects and circular references using `acceptable-objectp`. The algorithm is straightforward and makes use of an `EQ` hash table, which can be thought of as a mapping from pointers to values. This table describes the status of each cons: any cons that is unbound has not been seen before, any cons bound to `t` has already been "fully explored" and is known to be acceptable, and any cons bound to `'exploring` is currently being explored. When we encounter a new cons, we bind it to `'exploring` as we explore its car and cdr; after it has been fully explored, we rebind it to `t`. Accordingly, if we ever encounter a cons which we are already `'exploring`, we have found a circular reference.

Lisp Code
```
(defvar *acceptable-object-tbl*)
(declaim (type hash-table *acceptable-object-tbl*))

(defun-comp aux-acceptable-objectp (x)
  (or (and (integerp x)
           (<= 0 x))
```

```
       (and (symbolp x)
            (equal x (intern (symbol-name x) milawa-package)))
       (and (consp x)
            (let ((status (gethash x *acceptable-object-tbl*)))
              (cond ((eq status t)
                     t)
                    ((eq status nil)
                     (progn
                       (setf (gethash x *acceptable-object-tbl*)
                             'exploring)
                       (and (aux-acceptable-objectp (car x))
                            (aux-acceptable-objectp (cdr x))
                            (setf (gethash x *acceptable-object-tbl*)
                                  t))))
                    (t
                     nil))))))

(defun-comp acceptable-objectp (x)
  (let ((*acceptable-object-tbl* (make-hash-table :test 'eq)))
    (aux-acceptable-objectp x)))
```

Throughout our system, we rely upon the *acceptable-object invariant*: only acceptable objects shall be given as arguments to functions in the `MILAWA` package, and all `MILAWA`-package functions shall produce acceptable objects for all such inputs.

Given this invariant, the Common Lisp functions `if`, `equal`, `cons`, `consp`, and `symbolp` implement the semantics of IF, EQUAL, CONS, CONSP, and SYMBOLP, respectively, so above we imported them directly into the `MILAWA` package. The other primitives need to be defined, and for efficiency we suggest that the Lisp system inline calls to each of these.

Lisp Code

```
(declaim (inline MILAWA::natp
                 MILAWA::symbol-<
                 MILAWA::<
```

```
                  MILAWA::+
                  MILAWA::-
                  MILAWA::car
                  MILAWA::cdr))
```

We can implement NATP by defining the function `MILAWA::natp` as a simple alias for `integerp`; this is sufficient since the only acceptable objects which satisfy `integerp` are naturals.

Lisp Code

```
(defun-comp MILAWA::natp (x)
  (integerp x))
```

We implement SYMBOL-< with `MILAWA::symbol-<`. The Common Lisp function `string<` implements a lexicographic ordering on ASCII strings. When the relation is satisfied, it returns a number indicating where the strings differ, rather than `t` which SYMBOL-< is to return. We correct for this using the `if` expression.

Lisp Code

```
(defun-comp MILAWA::symbol-< (x y)
  (let ((x-fix (if (symbolp x) x nil))
        (y-fix (if (symbolp y) y nil)))
    (if (string< (symbol-name x-fix) (symbol-name y-fix))
        t
      nil)))
```

The arithmetic operations are straightforward to implement. Common Lisp provides `<`, `+`, and `-` operations for its unbounded integers. In the case of subtraction, we must be careful to return `0` when integer-subtraction produces a negative result.

Lisp Code

```
(defun-comp MILAWA::< (x y)
  (let ((x-fix (if (integerp x) x 0))
        (y-fix (if (integerp y) y 0)))
```

```
    (< x-fix y-fix)))

(defun-comp MILAWA::+ (x y)
  (let ((x-fix (if (integerp x) x 0))
        (y-fix (if (integerp y) y 0)))
    (+ x-fix y-fix)))

(defun-comp MILAWA::- (x y)
  (let* ((x-fix (if (integerp x) x 0))
         (y-fix (if (integerp y) y 0))
         (ans   (- x-fix y-fix)))
    (if (< ans 0) 0 ans)))
```

Finally, there are the primitive operations on conses, CAR and CDR. It is an error to call the Common Lisp functions car and cdr on non-consp arguments other than nil, so we are careful to handle this case separately.

Lisp Code
```
(defun-comp MILAWA::car (x)
  (if (consp x) (car x) nil))

(defun-comp MILAWA::cdr (x)
  (if (consp x) (cdr x) nil))
```

Lisp's abbreviations, let, let*, list, and, or, cond, and lambda, are also compatible with our definitions, so they were directly imported. The only remaining abbreviations are first, second, third, fourth, and fifth, and these are easy to define using the Lisp macro system.

Lisp Code
```
(defmacro MILAWA::first  (x) `(MILAWA::car ,x))
(defmacro MILAWA::second (x) `(MILAWA::first (MILAWA::cdr ,x)))
(defmacro MILAWA::third  (x) `(MILAWA::second (MILAWA::cdr ,x)))
(defmacro MILAWA::fourth (x) `(MILAWA::third (MILAWA::cdr ,x)))
(defmacro MILAWA::fifth  (x) `(MILAWA::fourth (MILAWA::cdr ,x)))
```

With that, the definitions of not, rank, ord<, ordp, and the definitions from Chapter 3 leading up to logic.proofp may be submitted as Lisp functions in the MILAWA package.

But first, an aside.

We will eventually explain how our system implements recursive and witnessing function definition events. When such events are processed, new Common Lisp functions will be defined in the MILAWA package. Since Common Lisp allows its functions to be redefined at run-time, we want to ensure our proof checker's functions are not overridden by user-submitted events. Toward this purpose, we introduce defun-safe and definline-safe. These commands act like defun, but also prevent functions from being redefined. To do this, a new global variable, *defined-functions-table*, is used to store tuples of the form

$$(\textit{name}\ \ \textit{formals}\ \ \textit{body}\ \ \textit{inlinep}).$$

Whenever defun-safe is used to define a function, the table is updated with the function's information. And before defun-safe will accept a definition, it first consults the table to ensure that if the function has been defined before, then the newly proposed definition is identical to the previous definition.

Lisp Code

```
(defvar *defined-functions-table* nil)

(defun-comp defun-safe-fn (name formals body inlinep)
  (let ((this-defun (list name formals body inlinep))
        (prev-defun (assoc name *defined-functions-table*)))
    (if prev-defun
        (unless (equal this-defun prev-defun)
          (error "Attempted redefinition of ~A.~%
                 Prev: ~A.~%
                 New: ~A~%"
                name prev-defun this-defun))
      (progn
        (push this-defun *defined-functions-table*)

        (when inlinep
          (eval `(declaim (inline ,name))))

        (eval `(compile (defun ,name ,formals
                          (declare (ignorable ,@formals))
                          ,body)))
        ))))

(defmacro defun-safe (name formals body)
  `(defun-safe-fn ',name ',formals ',body nil))

(defmacro definline-safe (name formals body)
  `(defun-safe-fn ',name ',formals ',body t))
```

We use `defun-safe` and `definline-safe` to submit the definitions leading up to `logic.proofp`. For efficiency, we suggest the Common Lisp system inline simple non-recursive functions such as accessors and constructors. In the end, we have a Common Lisp function, `MILAWA::logic.proofp`, which may be run on a computer to check proofs.

Lisp Code

```
(in-package "MILAWA")

(CL-USER::definline-safe not (x)
 (if x nil t))

(CL-USER::defun-safe rank (x)
  (if (consp x)
      (+ 1
         (+ (rank (car x))
            (rank (cdr x))))
    0))
```

... *and so on* ...

```
(CL-USER::definline-safe logic.proofp
  (x axioms thms atbl)
  (logic.flag-proofp 'proof x axioms thms atbl))
```

Unlike recursive function definitions, witnessing function definitions such as `logic.provable-witness` do not explain how to compute their values for arbitrary, concrete inputs. Nevertheless, we still define Common Lisp functions for witnessing function definitions. Such functions simply cause an error that indicates a witnessing function was called.

We show the definition of `logic.provable-witness`, below. Recall from page 2.9 that a witnessing function event involves a name, a bound variable, free variables, and a body. We ensure that each of these components appears in our error message, so that the `defun-safe` mechanism will prohibit any redefinition of witnessing functions.

Lisp Code
```
(CL-USER::defun-safe logic.provable-witness (x axioms thms atbl)
   (CL-USER::error "Called witnessing function ~A.~%"
                   '(logic.provable-witness
                     proof
                     (x axioms thms atbl)
                     (and (logic.appealp proof)
                          (logic.proofp proof axioms thms atbl)
                          (equal (logic.conclusion proof) x)))))
```

Finally, we define `logic.provablep` in the same way we introduced the other proof-checker functions, so that all the definitions from Chapter 3 have been included. Of course, attempts to run the Common Lisp function `logic.provablep` will simply result in an error being caused by `logic.provable-witness`.

Lisp Code
```
(CL-USER::defun-safe logic.provablep (x axioms thms atbl)
  (let ((proof (logic.provable-witness x axioms thms atbl)))
    (and (logic.appealp proof)
         (logic.proofp proof axioms thms atbl)
         (equal (logic.conclusion proof) x))))
```

4.2 Supporting Abbreviations

Our system allows its user to submit events, including definitions of recursive functions. For these definitions to be understandable, we would like the user to be able to use all of the abbreviations mentioned in Section 2.5. Accordingly, we implemented macros in the `MILAWA` package for `first` through `fifth`, and we imported `list`, `and`, `or`, `cond`, `let`, and `let*` from Common Lisp. Together, this allows for the use of abbreviations when we define Common Lisp functions with `defun`.

But to process a definition event, we will need to do more than define a new Common Lisp function. In particular, we must ensure the termination obligations are provable and add the definitional axiom. For these tasks, we will need to *translate* away any abbreviations and produce the actual term which should be used for the function's body. We could implement our translator as a regular Common Lisp function, but it is not difficult to define it as a function in our logic. Then, as with the proof checker, we can use `defun-safe` to produce a `MILAWA`-package Common Lisp function that can perform the translation.

We begin with handful of general-purpose utility functions. The function `remove-all` eliminates all occurrences of some element from a list.

Translator Definition 1: `remove-all`
```
(pequal* (remove-all a x)
         (if (consp x)
             (if (equal a (car x))
                 (remove-all a (cdr x))
               (cons (car x) (remove-all a (cdr x))))
```

```
           nil))
```

The function `remove-duplicates` eliminates repeated occurrences of elements from a list.

Translator Definition 2: `remove-duplicates`

```
(pequal* (remove-duplicates x)
         (if (consp x)
             (if (memberp (car x) (cdr x))
                 (remove-duplicates (cdr x))
               (cons (car x) (remove-duplicates (cdr x))))
           nil))
```

The function `difference` acts like a set-difference operation on lists, removing all elements from x which are not in y.

Translator Definition 3: `difference`

```
(pequal* (difference x y)
         (if (consp x)
             (if (memberp (car x) y)
                 (difference (cdr x) y)
               (cons (car x) (difference (cdr x) y)))
           nil))
```

We can apply `first` to every element in a list using `strip-firsts`, and similarly we can apply `second` with `strip-seconds`.

Translator Definition 4: `strip-firsts`

```
(pequal* (strip-firsts x)
         (if (consp x)
             (cons (first (car x))
                   (strip-firsts (cdr x)))
           nil))
```

Translator Definition 5: `strip-seconds`

```
(pequal* (strip-seconds x)
         (if (consp x)
             (cons (second (car x))
                   (strip-seconds (cdr x)))
           nil))
```

We can ask whether every element in a list is an n-tuple with `tuple-listp`.

Translator Definition 6: `tuple-listp`

```
(pequal* (tuple-listp n x)
         (if (consp x)
             (and (tuplep n (car x))
                  (tuple-listp n (cdr x)))
           t))
```

Finally, we implement `sort-symbols`, a simple insertion sort for lists of symbols, using the auxiliary function `sort-symbols-insert`, which performs a single insert.

Translator Definition 7: `sort-symbols-insert`

```
(pequal* (sort-symbols-insert a x)
         (if (consp x)
             (if (symbol-< a (car x))
                 (cons a x)
               (cons (car x) (sort-symbols-insert a (cdr x))))
           (list a)))
```

Translator Definition 8: `sort-symbols`

```
(pequal* (sort-symbols x)
         (if (consp x)
             (sort-symbols-insert (car x) (sort-symbols (cdr x)))
           nil))
```

Now we develop some routines to assist in the translation. Recall from page 42 the meaning of the abbreviation `list`,

Abbreviation	Meaning
`(list)`	`nil`
`(list` x_1`)`	`(cons` x_1 `nil)`
`(list` x_1 ... x_n`)`	`(cons` x_1 `(list` x_2 ... x_n`))`.

Given *args* $= (x_1 \ \ \ldots \ \ x_n)$, where each x_i is a term—or in other words, where each x_i has already been translated—the function `logic.translate-list-term` produces the translation of `(list` x_1 ... x_n`)`.

Translator Definition 9: `logic.translate-list-term`

```
(pequal* (logic.translate-list-term args)
         (if (consp args)
             (logic.function
              'cons
              (list (car args)
                    (logic.translate-list-term (cdr args))))
           ''nil))
```

Next, recall the meanings of the abbreviations `and` and `or`,

Abbreviation	Meaning
`(and)`	`t`
`(and` x_1`)`	x_1
`(and` x_1 ... x_n`)`	`(if` x_1 `(and` x_2 ... x_n`)` `nil)`
`(or)`	`nil`
`(or` x_1`)`	x_1
`(or` x_1 ... x_n`)`	`(if` x_1 x_1 `(or` x_2 ... x_n`))`.

Given *args* $= (x_1 \ \ \ldots \ \ x_n)$, where each x_i is a term, we may produce the translations of `(and` x_1 ... x_n`)` and `(or` x_1 ... x_n`)` using the functions `logic.translate-and-term` and `logic.translate-or-term`, respectively.

Translator Definition 10: `logic.translate-and-term`

```
(pequal* (logic.translate-and-term args)
         (if (consp args)
             (if (consp (cdr args))
                 (logic.function
                  'if
                  (list (first args)
                        (logic.translate-and-term (cdr args))
                        ''nil))
               (first args))
           ''t))
```

Translator Definition 11: `logic.translate-or-term`

```
(pequal* (logic.translate-or-term args)
         (if (consp args)
             (if (consp (cdr args))
                 (logic.function
                  'if
                  (list (first args)
                        (first args)
                        (logic.translate-or-term (cdr args))))
               (first args))
           ''nil))
```

Next we address `cond`. Recall that `(cond)` abbreviates `nil`, while

$$\texttt{(cond (}\mathit{cond}_1\ \ \mathit{result}_1\texttt{)}\ \ \ldots\ \ \texttt{(}\mathit{cond}_n\ \ \mathit{result}_n\texttt{))}$$

abbreviates

$$\texttt{(if }\mathit{cond}_1\ \ \mathit{result}_1\ \texttt{(cond (}\mathit{cond}_2\ \ \mathit{result}_2\texttt{)}\ \ \ldots\ \ \texttt{(}\mathit{cond}_n\ \ \mathit{result}_n\texttt{)))}.$$

Given $\mathit{tests} = (\mathit{test}_1\ \ \ldots\ \ \mathit{test}_n)$ and $\mathit{thens} = (\mathit{then}_1\ \ \ldots\ \ \mathit{then}_n)$ as arguments, where each test_i and then_i is a term, `logic.translate-cond-term` translates

$$\texttt{(cond (}\mathit{test}_1\ \ \mathit{then}_1\texttt{)}\ \ \ldots\ \ \texttt{(}\mathit{test}_n\ \ \mathit{then}_n\texttt{))}.$$

Translator Definition 12: `logic.translate-cond-term`

```
(pequal* (logic.translate-cond-term tests thens)
         (if (consp tests)
             (let ((test1 (car tests))
                   (then1 (car thens)))
               (logic.function
                'if
                (list test1
                      then1
                      (logic.translate-cond-term (cdr tests)
                                                 (cdr thens)))))
           ''nil))
```

Recall that given unique variables, *var*$_1, \ldots,$ *var*$_n$,

$$\texttt{(let ((}var_1\ term_1\texttt{)}\ \ldots\ \texttt{(}var_n\ term_n\texttt{))}\ \beta\texttt{)}$$

is an abbreviation for

$$\texttt{((lambda (}x_1\ \ldots\ x_m\ var_1\ \ldots\ var_n\texttt{)}\ \beta\texttt{)}\ x_1\ \ldots\ x_m\ term_1\ \ldots\ term_n\texttt{)},$$

where $x_1, \ldots, x_m$ are the free variables of β besides *var*$_1, \ldots,$ *var*$_n$ in lexicographic order. Given *vars* = (*var*$_1$ ... *var*$_n$), *terms* = (*term*$_1$ ... *term*$_n$), and *body*, where the *vars* are unique, the *terms* are terms, and *body* is a term, the function `logic.translate-let-term` produces the translation of

$$\texttt{(let ((}var_1\ term_1\texttt{)}\ \ldots\ \texttt{(}var_n\ term_n\texttt{))}\ body\texttt{)}.$$

Translator Definition 13: `logic.translate-let-term`

```
(pequal*
 (logic.translate-let-term vars terms body)
 (let* ((body-vars (remove-duplicates (logic.term-vars body)))
        (id-vars   (sort-symbols (difference body-vars vars)))
        (formals   (app id-vars vars))
        (actuals   (app id-vars terms)))
```

```
    (logic.lambda formals body actuals)))
```

Finally, recall from page 43 that `(let* () β)` abbreviates β, while

`(let* ((`var_1 $term_1$`) ... (`var_n $term_n$`)) β)`

abbreviates

```
(let ((var₁ term₁))
  (let* ((var₂ term₂) ... (varₙ termₙ))
    β)).
```

Given *vars* = (var_1 ... var_n), *terms* = ($term_1$ ... $term_n$), and *body*, where the *vars* are unique, the *terms* are terms, and *body* is a term, the function `logic.translate-let*-term` produces the translation of

`(let* ((`var_1 $term_1$`) ... (`var_n $term_n$`))` *body*`)`.

Translator Definition 14: `logic.translate-let*-term`

```
(pequal* (logic.translate-let*-term vars terms body)
         (if (consp vars)
             (logic.translate-let-term
              (list (car vars))
              (list (car terms))
              (logic.translate-let*-term (cdr vars)
                                         (cdr terms)
                                         body))
           body))
```

We now combine these utilities to create our translator, `logic.flag-translate`. This is a flag function which can be used to translate an object from $\mathbb{U}$ into (1) a term, or (2) a list of terms, depending upon its mode of operation.

Translation may fail—for instance, the object we are translating may contain malformed abbreviations like `(cond 3)`, which does not have the proper structure, or `(let ((x 1) (x 2)) (+ x x))`, which attempts to bind the same variable twice—and we handle failure differently depending upon which mode we are in. In term mode, a successful translation produces a term (i.e., an object accepted by `logic.termp`) and NIL is returned to indicate failure; this is unambiguous since NIL is not a valid term. In list mode, `logic.flag-translate` always returns a cons of the form (*successp* . *value*), where *successp* is T when the translation has been successful or NIL otherwise, and where *value* is a list of abbreviation-free terms on success or NIL on failure.

Since `logic.flag-translate` is rather long, we split up its definition into short segments which we can comment upon. Recall from page 41 that a numeric token, n, may be used as an abbreviation for 'n, `t` abbreviates `'t`, and `nil` abbreviations `'nil`. Our first few cases implement these abbreviations.

Translator Definition 15: `logic.flag-translate`

```
(pequal* (logic.flag-translate flag x)
         (if (equal flag 'term)
             (cond ((natp x)
                    (list 'quote x))
                   ((symbolp x)
                    (if (or (equal x nil)
                            (equal x t))
                        (list 'quote x)
                      x))
```

Otherwise, we have a cons. Most of the interesting cases occur when the car is a symbol, which might be an abbreviation, a function application, or QUOTE. We begin by handling QUOTE. If x has the form (QUOTE v), then it is already a term and we return it unchanged; otherwise this is an error.

```
((symbolp (car x))
 (let ((fn (car x)))
   (cond ((equal fn 'quote)
          (if (tuplep 2 x)
              x
            nil))
```

Next, we handle the abbreviations `first`, `second`, `third`, `fourth`, and `fifth`. Here, x must again be a 2-tuple for the abbreviation to be well-formed. We begin by recursively translating the argument, and if that is successful, we wrap the resulting term in the necessary `car` and `cdr` applications.

```
((memberp fn '(first second third fourth fifth))
 (and (tuplep 2 x)
      (let ((arg (logic.flag-translate 'term (second x))))
        (and arg
             (let* ((1cdr (logic.function 'cdr (list arg)))
                    (2cdr (logic.function 'cdr (list 1cdr)))
                    (3cdr (logic.function 'cdr (list 2cdr)))
                    (4cdr (logic.function 'cdr (list 3cdr))))
               (logic.function
                'car
                (list (cond ((equal fn 'first)  arg)
                            ((equal fn 'second) 1cdr)
                            ((equal fn 'third)  2cdr)
                            ((equal fn 'fourth) 3cdr)
                            (t                  4cdr)))))))))
```

For the abbreviations `and`, `or`, and `list`, we recursively translate the arguments. If these translations are successful, we use our utility functions to build the appropriate term.

```
((memberp fn '(and or list))
 (and (true-listp (cdr x))
      (let ((arguments+ (logic.flag-translate 'list (cdr x))))
        (and (car arguments+)
             (cond
              ((equal fn 'and)
               (logic.translate-and-term (cdr arguments+)))
              ((equal fn 'or)
               (logic.translate-or-term (cdr arguments+)))
              (t
               (logic.translate-list-term (cdr arguments+))))))))
```

For the abbreviation `cond`, we first ensure that we are given a true-list of 2-tuples as arguments. We then extract the *tests* and *thens* and attempt to translate them recursively. If this is all successful, we combine everything into a term using our utility function, `logic.translate-cond-term`.

```
((equal fn 'cond)
 (and (true-listp (cdr x))
      (tuple-listp 2 (cdr x))
      (let* ((tests  (strip-firsts (cdr x)))
             (thens  (strip-seconds (cdr x)))
             (tests+ (logic.flag-translate 'list tests))
             (thens+ (logic.flag-translate 'list thens)))
        (and (car tests+)
             (car thens+)
             (logic.translate-cond-term (cdr tests+)
                                        (cdr thens+))))))
```

For `let` and `let*`, we first ensure that we are given a 3-tuple whose second component is a true list of 2-tuples. We extract the variables and ensure they are variables and, in the case of `let`, ensure they are unique. We extract the terms being bound to these variables and recursively translate them, and recursively translate the body. If all of this is successful, we use our utility functions to build the appropriate lambda term.

```
((memberp fn '(let let*))
 (and (tuplep 3 x)
      (let ((pairs (second x))
            (body  (logic.flag-translate 'term (third x))))
        (and body
             (true-listp pairs)
             (tuple-listp 2 pairs)
             (let* ((vars   (strip-firsts pairs))
                    (terms  (strip-seconds pairs))
                    (terms+ (logic.flag-translate 'list terms)))
               (and (car terms+)
                    (logic.variable-listp vars)
                    (cond
                     ((equal fn 'let)
                      (and (uniquep vars)
                           (logic.translate-let-term vars
                                                     (cdr terms+)
                                                     body)))
                     (t
                      (logic.translate-let*-term vars
                                                 (cdr terms+)
                                                 body)))))))))
```

At this point we have handled all of the abbreviations. For function applications, we first ensure the arguments are a true list, then try to recursively translate them and apply the function to the resulting terms.

```
((logic.function-namep fn)
 (and (true-listp (cdr x))
      (let ((arguments+ (logic.flag-translate 'list (cdr x))))
            (and (car arguments+)
                 (logic.function fn (cdr arguments+))))))
```

Otherwise, we are still in the case where the car is a symbol, but it is not QUOTE, an abbreviation, or a function name, so the translation fails.

```
(t
  nil))))
```

To translate a lambda abbreviation, we first ensure it has the appropriate structure. The car must be a 3-tuple whose first component is the symbol LAMBDA, and the actuals must be a true list. We attempt to recursively translate the body and the actuals, and ensure that a valid lambda can be produced: the formals must be a true list of unique variables; there must be the same number of formals and actuals; and the formals must mention all of the body's free variables. If all these conditions are met, we build a new lambda abbreviation, using the translated body and actuals.

```
((and (tuplep 3 (car x))
      (true-listp (cdr x)))
 (let* ((lambda-symbol (first (car x)))
        (vars          (second (car x)))
        (body          (third (car x)))
        (new-body      (logic.flag-translate 'term body))
        (actuals+      (logic.flag-translate 'list (cdr x))))
   (and (equal lambda-symbol 'lambda)
        (true-listp vars)
        (logic.variable-listp vars)
        (uniquep vars)
        new-body
        (subsetp (logic.term-vars new-body) vars)
        (car actuals+)
        (equal (len vars) (len (cdr actuals+)))
        (logic.lambda vars new-body (cdr actuals+)))))
```

There are no other valid objects which can be translated into terms, so if none of the above cases have matched, we end the term mode with failure.

```
(t
 nil))
```

Finally, implementing the list mode is entirely straightforward. We simply try to translate each element in the list, and propagate any failures.

```
(if (consp x)
    (let ((first (logic.flag-translate 'term (car x)))
          (rest  (logic.flag-translate 'list (cdr x))))
      (if (and first (car rest))
          (cons t (cons first (cdr rest)))
        (cons nil nil)))
  (cons t nil))))
```

As usual, we also implement a flag-free wrapper function, `logic.translate`, which attempts to translate its argument into a term. It returns a term on success, or NIL on failure.

Translator Definition 16: `logic.translate`

```
(pequal* (logic.translate x)
         (logic.flag-translate 'term x))
```

4.3 The History

Our system allows its user to extend a history with admissible events. We introduce the global variables `*arity-table*`, `*axioms*`, and `*theorems*` to store the arity table, the list of axioms, and the list of theorems associated with the current history, respectively.

Lisp Code

```
(CL-USER::in-package "CL-USER")
(defvar *arity-table* nil)
(defvar *axioms* nil)
(defvar *theorems* nil)
```

When our program is started, its user begins in the empty history. Recall from page 58 that the arity table for the empty history consists of the entries in `(logic.initial-arity-table)`, as well as entries for `not`, `rank`, `ordp`, and `ord<`, so we initialize `*arity-table*` with these entries.

Lisp Code

```
(in-package "MILAWA")
```

```
(CL-USER::setf CL-USER::*arity-table*
  (app '((rank . 1)
         (ordp . 1)
         (ord< . 2))
       (logic.initial-arity-table)))
```

The axioms of the empty history are the fifty-six numbered axioms mentioned in Chapter 2, so we initialize `*axioms*` with these formulas. Except for the definitions of `not`, `rank`, `ord<` and `ordp`, these formulas are abbreviation-free, so we can put them in directly. Then, we use `logic.translate` to add the definitions which use abbreviations.

Lisp Code

```
(CL-USER::setf CL-USER::*axioms*
  (app '(;; reflexivity
         (pequal* x x)

         ... and so on ...

         ;; closed-universe
         (por* (pequal* (natp x) 't)
               (por* (pequal* (symbolp x) 't)
                     (pequal* (consp x) 't)))
         )

       (list
        ;; definition-of-not
        (logic.pequal '(not x) (logic.translate '(if x nil t)))

        ... and so on ...

        ;; definition-of-ordp
        (logic.pequal '(ordp x) (logic.translate ...)))))
```

Finally, the empty history has no theorems, so we leave `*theorems*` with its initial value, `nil`.

How can we say our program begins in the empty history when we have already defined several functions—the proof checker, translator, and termination obligation routines—in the `MILAWA` package that are not in the empty history? Since we have not added the names of these functions to `*arity-table*` and have not added their definitions to `*axioms*`, they are effectively "invisible," and initially cannot be reasoned about. Later, as the user works, he may try to define these functions. Our use of `defun-safe` ensures that he can only use precisely the definition we have given above, and even then he will still have to prove the definition is admissible. As part of our bootstrapping process (Chapter 12), we admit all of these functions.

4.4 Termination Obligations

Recall from page 59 that for a recursive function definition to be admissible, its termination obligations must be provable. It is straightforward to construct the ordinal obligation, but to determine what the progress obligations are, we will need to construct a call map and then process it. As with the proof checker and term translator, we implement our functions in the logic, then use `defun-safe` to produce a `MILAWA`-package Common Lisp function.

We will represent call maps as association lists where each key contains the actuals of a recursive call, and where each value is a list of the rulers of this recursive call. In other words, each key is a list of terms, and each value is also a list of terms.

The algorithm for computing the call map for a function is given on page 60, and we will review it in a moment. But first, we develop a couple of utility routines. To handle `if` expressions, we need a way to extend a call map by adding a new ruler to each of its entries. We implement this operation as the function `cons-onto-ranges`, which builds a new association list from x by consing a onto each of the values of x.

Termination Definition 1: `cons-onto-ranges`

```
(pequal* (cons-onto-ranges a x)
         (if (consp x)
             (cons (cons (car (car x))
                         (cons a (cdr (car x))))
                   (cons-onto-ranges a (cdr x)))
           nil))
```

Similarly, to build the call map for a lambda abbreviation, we must be able to apply a substitution to all calls and rulers in a call map. We implement this using the function `logic.substitute-callmap`.

Termination Definition 2: `logic.substitute-callmap`

```
(pequal* (logic.substitute-callmap x sigma)
         (if (consp x)
             (let ((actuals (car (car x)))
                   (rulers  (cdr (car x))))
               (cons (cons (logic.substitute-list actuals sigma)
                           (logic.substitute-list rulers sigma))
                     (logic.substitute-callmap (cdr x) sigma)))
           nil))
```

We now review the CALLMAP algorithm and provide an implementation. Given a function name, f, the function `logic.flag-callmap` constructs either (1) its call map for a term, or (2) the union of its call maps for each term in a list, based upon the mode of operation specified by its flag parameter. When x is a constant or a variable, there are no recursive calls of f within x, so CALLMAP(f, x) is empty.

Termination Definition 3: `logic.flag-callmap`

```
(pequal*
 (logic.flag-callmap flag f x)
 (if (equal flag 'term)
     (cond ((logic.constantp x) nil)
           ((logic.variablep x) nil)
```

When x is `(if` a b c`)`, CALLMAP(f, x) includes the calls from a, verbatim; the calls from b, but modified so that a is also a ruler of each call; and the calls of c, modified so `(not` a`)` is also a ruler of each call.

```
((logic.functionp x)
 (let ((name (logic.function-name x))
       (args (logic.function-args x)))
   (cond ((and (equal name 'if)
               (equal (len args) 3))
          (let ((test-calls
                  (logic.flag-callmap 'term f (first args)))
                 (true-calls
                  (cons-onto-ranges
                   (first args)
                   (logic.flag-callmap 'term f (second args))))
                 (else-calls
                  (cons-onto-ranges
                   (logic.function 'not (list (first args)))
                   (logic.flag-callmap 'term f (third args)))))
            (app test-calls (app true-calls else-calls))))
```

Still in the function case, when x is `(`f t_1 … t_n`)`, CALLMAP(f, x) associates `(`f t_1 … t_n`)` with no rulers, and also includes the calls from CALLMAP(f, t_i).

```
((equal name f)
 (let ((this-call   (cons args nil))
       (child-calls (logic.flag-callmap 'list f args)))
   (cons this-call child-calls)))
```

Still in the function case, when x is any other function call, `(`g t_1 … t_m`)`, CALLMAP(f, x) is the union of CALLMAP(f, t_i).

```
(t
 (logic.flag-callmap 'list f args)))))
```

Otherwise, and finishing the term mode, when x is a lambda abbreviation, `((lambda (`x_1 ... x_n`)` β`)` t_1 ... t_n`)`, its call map includes all calls in the actuals, i.e., CALLMAP(f, t_i), and also includes the modified call map of β, formed by substituting $\sigma = [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ into each call and all rulers.

```
((logic.lambdap x)
 (let ((formals (logic.lambda-formals x))
       (body    (logic.lambda-body x))
       (actuals (logic.lambda-actuals x)))
   (let ((actuals-calls (logic.flag-callmap 'list f actuals))
         (body-calls    (logic.flag-callmap 'term f body))
         (sigma         (pair-lists formals actuals)))
     (app actuals-calls
          (logic.substitute-callmap body-calls sigma))))))
```

Finally, in the list mode, we simply combine the call maps of each term in the list.

```
(if (consp x)
    (app (logic.flag-callmap 'term f (car x))
         (logic.flag-callmap 'list f (cdr x)))
  nil)))
```

As usual, we introduce a flag-free wrapper, `logic.callmap`, to compute the call map for a term.

Termination Definition 4: `logic.callmap`

```
(pequal* (logic.callmap f x)
         (logic.flag-callmap 'term f x))
```

Once we have the call map, it is straightforward to produce the progress obligations. Recall that for each recursive call, `(`f a_1 ... a_n`)`, associated with the rulers $r_1, \ldots, r_m$, we have the obligation

```
(por* (pequal* (ord< m/σ m) t)
      (por* (pequal* r_1 nil)
            ⋱
               (por* (pequal* r_{m-1} nil)
                     (pequal* r_m nil))) ... )),
```

where $\sigma = [x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n]$. To construct these obligations, a couple of utility functions are useful. The function `repeat` creates a list containing n copies of the element a.

Termination Definition 5: `repeat`

```
(pequal* (repeat a n)
         (if (zp n)
             nil
           (cons a (repeat a (- n 1)))))
```

Given two equal-length lists of terms, $(x_1 \ \ldots \ x_n)$ and $(y_1 \ \ldots \ y_n)$, the function `logic.pequal-list` creates a the list of formulas,

$$((\texttt{pequal*}\ x_1\ y_1)\ \ \ldots\ \ (\texttt{pequal*}\ x_n\ y_n)).$$

Termination Definition 6: `logic.pequal-list`

```
(pequal* (logic.pequal-list x y)
         (if (and (consp x)
                  (consp y))
             (cons (logic.pequal (car x) (car y))
                   (logic.pequal-list (cdr x) (cdr y)))
           nil))
```

The function `logic.progress-obligation` constructs the formula for a single progress obligation. As inputs, it takes the measure and formals from the proposed definition, along with the actuals and rulers from the entry in the call map.

Termination Definition 7: `logic.progress-obligation`

```
(pequal* (logic.progress-obligation measure formals actuals rulers)
          (let* ((sigma    (pair-lists formals actuals))
                 (m/sigma  (logic.substitute measure sigma))
                 (ord-term (logic.function 'ord<
                                           (list m/sigma measure))))
            (logic.disjoin-formulas
             (cons (logic.pequal ord-term ''t)
                   (logic.pequal-list
                    rulers
                    (repeat ''nil (len rulers)))))))
```

We extend this with `logic.progress-obligations`, which constructs the progress obligations for an entire call map.

Termination Definition 8: `logic.progress-obligations`

```
(pequal* (logic.progress-obligations measure formals callmap)
         (if (consp callmap)
             (let* ((entry   (car callmap))
                    (actuals (car entry))
                    (rulers  (cdr entry)))
               (cons (logic.progress-obligation measure formals
                                                actuals rulers)
                     (logic.progress-obligations measure formals
                                                 (cdr callmap))))
           nil))
```

Finally, `logic.termination-obligations` produces the list of all admission obligations—the ordinal obligation and the progress obligations—for a proposed function definition given the name, formals, body, and measure.

Termination Definition 9: `logic.termination-obligations`

```
(pequal* (logic.termination-obligations name formals body measure)
         (let ((callmap (logic.callmap name body)))
           (if callmap
               (cons (logic.pequal
                      (logic.function 'ordp (list measure))
```

```
                    ''t)
                   (logic.progress-obligations measure formals
                                               callmap))
               nil)))
```

4.5 Establishing Provability

To admit a theorem event, we must first ensure its formula is provable with respect to the current axioms and theorems of our history. Likewise, to admit a recursive function definition, we must first ensure its termination obligations are provable.

In the logic, we express the provability of a formula with `logic.provablep`. But this function cannot be used by our system to determine whether an arbitrary formula is provable; it is defined in terms of the witnessing function `logic.provable-witness`, so calling it would only cause an error.

When we need to know a formula is provable, our approach is to require the user to provide a proof. Initially, the user will be required to provide a `logic.proofp`-checkable proof of the desired formula. He may use any tools at all, including untrusted or extralogical ones, to create these proofs. The real problem is not that it is inconvenient to write tools to construct proofs, but that unless we can somehow increase the level of abstraction, interesting proofs become so large that it is impractical to construct, store, and check them. To increase our level of abstraction, our system allows for the development of more powerful proof-checking functions. Once a new proof checker has been verified, we can begin using it to check proofs.

All proof-checkers accepted by our system take the same arguments as `logic.proofp`—x, an appeal to be checked; *axioms*, the formulas considered to be axioms; *thms*, the formulas considered to be theorems; and *atbl*, the arity-table being used. At any point in time, exactly one proof-checker is considered to be active, and its name is held in the global variable `*proof-checker*`, which is initially set to `logic.proofp`. The function `check-proof` calls upon the current `*proof-checker*` to check a particular proof, while `check-proof-list` uses it to check a list of proofs.

Lisp Code

```
(CL-USER::in-package "CL-USER")

(defvar *proof-checker* 'MILAWA::logic.proofp)

(defun-comp check-proof (x axioms thms atbl)
  (funcall *proof-checker* x axioms thms atbl))

(defun-comp check-proof-list (x axioms thms atbl)
  (if (consp x)
      (and (check-proof (car x) axioms thms atbl)
           (check-proof-list (cdr x) axioms thms atbl))
    t))
```

The connection between `logic.proofp` and the proofs of our logic has been discussed in depth in Chapter 3, so we consider it to be a valid proof checker. But what about other functions? How can we trust that they accept only provable formulas?

We say that the *fidelity claim* for a function name, f, is the formula

```
(por* (pequal* (logic.appealp x) nil)
      (por* (pequal* (f x axioms thms atbl) nil)
            (pnot* (pequal* (logic.provablep (logic.conclusion x)
                                             axioms thms atbl)
                            nil)))).
```

Given a particular function name, we may construct the fidelity claim with `logic.fidelity-claim`.

Definition: `logic.fidelity-claim`

```
(pequal* (logic.fidelity-claim name)
         (logic.por
          '(pequal* (logic.appealp x) 'nil)
          (logic.por
           (logic.pequal (logic.function name '(x axioms thms atbl))
                         ''nil)
           '(pnot* (pequal* (logic.provablep (logic.conclusion x)
                                             axioms thms atbl)
                            'nil)))))
```

Suppose we have used `logic.proofp` to prove the fidelity claim for some new proof-checker, f. Then, we know that any time f accepts some appeal, x, the conclusion of x is provable in the sense of `logic.proofp`. In other words, f only accepts formulas which are provable. Hence, we can trust f, and we will allow it to be used as a proof checker.

The function `switch-proof-checker` takes the name of a function as an argument. It ensures the fidelity claim has been established for this function, and then switches `*proof-checker*` to the new function. If the fidelity claim has not been established, an error is caused.

Lisp Code
```
(defun-comp switch-proof-checker (name)
  (unless (MILAWA::logic.function-namep name)
    (error "The name is invalid"))
  (unless (MILAWA::memberp (MILAWA::logic.fidelity-claim name)
                           *theorems*)
    (error "The fidelity claim has not been proven"))
  (setf *proof-checker* name))
```

4.6 Reading Objects

Because the proofs accepted by `logic.proofp` are sometimes quite large, it is useful to add a file-reading capability to our system so that proofs may be stored in separate files. Reading these files can sometimes take a long time, so we introduce a simple time-reporting macro. This macro evaluates *form* and returns its result, but as a side-effect also prints a message that says how long it took to evaluate *form*.

Lisp Code
```
(CL-USER::in-package "CL-USER")

(defmacro report-time (message form)
  `(let* ((start-time (get-internal-real-time))
          (value      ,form)
          (stop-time  (get-internal-real-time))
          (elapsed    (/ (coerce (- stop-time start-time) 'float)
                         internal-time-units-per-second)))
     (format t ";; ~A took ~$ seconds~%" ,message elapsed)
     value))
```

Common Lisp provides a flexible reader which can be used to parse text into Lisp objects, and we will use this reader to obtain objects from the user. Lisp's reader allows for the introduction of numbered abbreviations so that, for instance, one may write `#1=(a . b)` to define `#1#` as an abbreviation for `(a . b)`.

Abbreviations are very useful, but many Lisps implement them quite inefficiently, e.g., using association lists. We therefore provide our own implementation of the sharp-equal (e.g., `#1=`) and sharp-sharp (e.g., `#1#`) reader macros which store the abbreviations in a hash table.

Lisp Code
```
(defvar *milawa-abbreviations-hash-table*)
(declaim (type hash-table *milawa-abbreviations-hash-table*))


(defun-comp milawa-sharp-equal-reader (stream subchar arg)
  (declare (ignore subchar))
  (multiple-value-bind
   (value presentp)
   (gethash arg *milawa-abbreviations-hash-table*)
   (declare (ignore value))
```

```
    (when presentp
      (error "#~A= is already defined." arg))
    (let ((object (read stream)))
      (setf (gethash arg *milawa-abbreviations-hash-table*)
            object))))

(defun-comp milawa-sharp-sharp-reader (stream subchar arg)
  (declare (ignore stream subchar))
  (or (gethash arg *milawa-abbreviations-hash-table*)
      (error "#~A# used but not defined." arg)))
```

To instruct the Common Lisp reader to use our implementation of these macros, we set up a new *readtable* and configure it appropriately.

Lisp Code

```
(defvar *milawa-readtable* (copy-readtable *readtable*))
(declaim (readtable *milawa-readtable*))

(let ((*readtable* *milawa-readtable*))
  (set-dispatch-macro-character #\# #\#
                                #'milawa-sharp-sharp-reader)
  (set-dispatch-macro-character #\# #\=
                                #'milawa-sharp-equal-reader))
```

To read objects from a file on disk, we use `milawa-read-file`. The let-bindings ensure that a fresh hash-table is used for abbreviations, our custom sharp-equal and sharp-sharp macros are used, and that symbols in the file are from the `MILAWA` package by default. We read the entire file at once, and ensure that its contents are acceptable.

Lisp Code

```
(defconstant unique-cons-for-eof (cons 'unique-cons 'for-eof))

(defun-comp milawa-read-file-aux (stream)
  (let ((obj (read stream nil unique-cons-for-eof)))
    (cond ((eq obj unique-cons-for-eof)
           nil)
          (t
           (cons obj (milawa-read-file-aux stream))))))

(defun-comp milawa-read-file (filename)
  (format t ";; Reading from ~A~%" filename)
  (report-time "Reading the file"
    (let* ((*milawa-abbreviations-hash-table* (make-hash-table
                                               :size 10000
                                               :rehash-size 100000
                                               :test 'eql))
           (*readtable* *milawa-readtable*)
           (*package* milawa-package)
           (stream (open filename
                         :direction :input
                         :if-does-not-exist :error))
           (contents (milawa-read-file-aux stream)))
      (close stream)
      (if (acceptable-objectp contents)
          contents
        (error "unacceptable object encountered")))))
```

4.7 Events

We now explain how our system admits theorem events, recursive function definition events, and witnessing function definition events.

Recall that a theorem event extends the history by adding some formula to the list of theorems. To be admissible, the formula must be well-formed with respect to the current arity table, and must be provable from the current axioms and theorems. The function admit-theorem takes two arguments, the formula to prove and the name of a file which allegedly contains a proof of this formula. It checks that the formula is well formed and that the proof is valid, and extends the history by adding the formula as a theorem. If the formula is already a theorem, we do not add it again.

Lisp Code

```
(defun-comp admit-theorem (formula filename)
  (unless (MILAWA::logic.formulap formula)
    (error "The conclusion, ~A, is not a formula" formula))
  (unless (MILAWA::logic.formula-atblp formula *arity-table*)
    (error "The conclusion, ~A, is not well-formed" formula))
  (let ((proof (car (milawa-read-file filename))))
    (unless (MILAWA::logic.appealp proof)
      (error "The proof is not a valid appeal"))
    (unless (equal (MILAWA::logic.conclusion proof) formula)
      (error "The proof does not have the right conclusion"))
    (unless (check-proof proof *axioms* *theorems* *arity-table*)
      (error "The proof was rejected")))
  (unless (MILAWA::memberp formula *theorems*)
    (push formula *theorems*))
  t)
```

Recall from page 59 that a recursive function definition event extends the history with a definitional axiom and a new binding in the arity table. To be admissible, the name must be a new name which is not already in the arity table of h, the body and measure must be well-formed with respect to the new arity table and may only mention the formals, and the termination obligations must be provable.

The function admit-defun checks that these conditions are met, and if so extends the history appropriately. As arguments, it takes the name, formals, body, and measure of the function to be defined, and a flag that indicates whether the function should be inlined or not; it also takes the name of a file which should contain proofs of the termination obligations.

We begin by translating away any abbreviations in the body and the measure, and by checking the admission criteria other than the termination obligations.

Lisp Code

```
(defun-comp admit-defun (name formals raw-body raw-measure inlinep
                         filename)
  (let* ((body     (MILAWA::logic.translate raw-body))
         (measure  (MILAWA::logic.translate raw-measure))
         (arity    (MILAWA::len formals))
         (new-atbl (cons (cons name arity) *arity-table*)))
    (unless (MILAWA::logic.function-namep name)
      (error "The name is invalid"))
    (unless (MILAWA::logic.variable-listp formals)
      (error "The formals are not variables"))
    (unless (MILAWA::uniquep formals)
      (error "The formals are not unique"))
    (unless (MILAWA::logic.termp body)
      (error "The body did not translate to a term"))
    (unless (MILAWA::logic.termp measure)
      (error "The measure did not translate to a term"))
    (unless (MILAWA::subsetp (MILAWA::logic.term-vars body) formals)
      (error "The body mentions variables besides the formals"))
    (unless (MILAWA::subsetp (MILAWA::logic.term-vars measure)
                             formals)
      (error "The measure mentions variables besides the formals"))
    (unless (MILAWA::logic.term-atblp body new-atbl)
      (error "The body is not well-formed"))
    (unless (MILAWA::logic.term-atblp measure new-atbl)
      (error "The measure is not well-formed"))
```

Next, given that all of the above criteria have been met, we compute the termination obligations. To ensure these formulas are provable, we check that the supplied proofs are a list of appeals which establish each of these formulas, and that these proofs are accepted by the currently trusted proof checker.

```
    (let ((obligations (MILAWA::logic.termination-obligations
                        name formals body measure))
          (proofs (car (milawa-read-file filename))))
      (unless (MILAWA::logic.appeal-listp proofs)
        (error "The proofs are not a list of appeals"))
      (unless (equal (MILAWA::logic.strip-conclusions proofs)
                     obligations)
        (error "The proofs have the wrong conclusions"))
      (unless (report-time "Checking the proofs"
                           (check-proof-list proofs *axioms*
                                             *theorems* new-atbl))
        (error "A proof was rejected")))
```

Finally, we create a new Lisp function in the `MILAWA` package, extend the arity table, and add the definitional axiom to the list of axioms. If the function already has a Lisp definition, then `defun-safe-fn` will cause an error unless this new definition is identical. As with theorem events, we do not redundantly extend the arity table or list of axioms if the function has already been defined.

```
      (defun-safe-fn name formals raw-body inlinep)
      (unless (MILAWA::lookup name *arity-table*)
        (push (cons name arity) *arity-table*))
      (let ((new-axiom (MILAWA::logic.pequal
                         (MILAWA::logic.function name formals)
                         body)))
        (unless (MILAWA::memberp new-axiom *axioms*)
          (push new-axiom *axioms*))))
    t)
```

Recall that a witnessing function definition includes a function name, f, a variable, v, called the bound variable, a list of distinct variables, $x_1, \ldots, x_n$ called the free variables, and a term, β, called its body. Such an event extends the arity table by associating f with n, and adds the axiom

```
(por* (pequal* β nil)
      (pnot* (pequal* ((lambda (v x₁ ... xₙ) β)
                        (f x₁ ... xₙ) x₁ ... xₙ)
                      nil))).
```

To be admissible, f must be a new name which is not already in the arity table, v must not be any of the free variables, β must be well-formed with respect to the arity table, and $\textsc{freevars}(\beta)$ must be a subset of $\{v, x_1, \ldots, x_n\}$.

The function admit-witness checks that a witnessing definition is admissible and, if so, extends the history appropriately. As arguments, it takes the name, bound variable, free variables, and body of the witnessing definition. We begin by translating away any abbreviations in the body, and checking that the admissibility criteria have been met.

Lisp Code

```
(defun-comp admit-witness (name bound-var free-vars raw-body)
  (let* ((body     (MILAWA::logic.translate raw-body))
         (all-vars (cons bound-var free-vars)))
    (unless (MILAWA::logic.function-namep name)
      (error "Invalid function name"))
    (unless (MILAWA::logic.variablep bound-var)
      (error "The bound-var is not a variable"))
    (unless (MILAWA::logic.variable-listp free-vars)
      (error "The free-vars are not variables"))
    (unless (MILAWA::uniquep (cons bound-var free-vars))
      (error "The variables are not unique"))
    (unless (MILAWA::logic.termp body)
      (error "The body did not translate to a term"))
    (unless (MILAWA::subsetp (MILAWA::logic.term-vars body)
                             all-vars)
      (error "The body's variables are not legal"))
    (unless (MILAWA::logic.term-atblp body *arity-table*)
      (error "The body is not well-formed"))
```

Given that all of the criteria have been met, we create a Lisp function in the MILAWA package for this definition which simply causes an error. We also extend the arity table and add the witnessing axiom. As with definitions, we do not extend the arity table or axioms redundantly if this definition has already been given.

```
(defun-safe-fn name free-vars
    `(CL-USER::error "Called witnessing function ~A.~%"
                     '(,name ,bound-var ,free-vars ,raw-body))
    nil)
(unless (MILAWA::lookup name *arity-table*)
  (push (cons name (MILAWA::len free-vars)) *arity-table*))
(let ((new-axiom
      (MILAWA::logic.por
       (MILAWA::logic.pequal body ''nil)
       (MILAWA::logic.pnot
        (MILAWA::logic.pequal
         (MILAWA::logic.lambda
          all-vars body
          (cons (MILAWA::logic.function name free-vars)
                free-vars))
         ''nil)))))
(unless (MILAWA::memberp new-axiom *axioms*)
  (push new-axiom *axioms*))))
t)
```

4.8 Checkpointing

Checking a large collection of proofs can take a long time, so it is useful to be able to save our progress from time to time during the process. Although there is no standard checkpointing mechanism for Common Lisp, many Lisp implementations provide a way to save an "image" of a running Lisp session which can be restarted later. In many Lisps, creating such an image also terminates the currently running program.

Using "features," it is generally possible to detect which Lisp implementation we are running on. Then, on a per-implementation basis, we can implement the function `save-and-exit`, which saves an image using the given filename as a prefix. That is, the code following `#+allegro` is only used on Allegro Common Lisp, the code following #+clozure is only used on Clozure Common Lisp, and so on. To facilitate checking proofs with multiple computers that share a file system, it is convenient to be able to give different names to the image files created by each system. Our save-and-exit function requires that the constant image-extension has been defined by the user.

Lisp Code

```
(defun-comp save-and-exit (filename)
  #+allegro
  (progn
    (setq EXCL::*restart-init-function* 'main)
    (EXCL::dumplisp
     :name (concatenate 'string filename "." image-extension))
    (exit))
  #+clozure
  (CCL::save-application
   (concatenate 'string filename "." image-extension)
   :toplevel-function #'main
   :purify t)
  #+clisp
  (progn
    (EXT:saveinitmem
     (concatenate 'string filename "." image-extension)
     :init-function #'main)
    (quit))
  #+cmu
  (EXTENSIONS::save-lisp
   (concatenate 'string filename "." image-extension)
   :init-function #'main
   :purify t)
  #+sbcl
  (SB-EXT:save-lisp-and-die
   (concatenate 'string filename "." image-extension)
   :toplevel #'main
   :purify t)
  #+scl
  (EXT:save-lisp
   (concatenate 'string filename "." image-extension)
   :init-function #'main
   :gc :full
   :purify t)
  ;; Handler for other lisps
  (error "implement save-and-exit on this lisp"))
```

4.9 The Command Loop

We now combine the above definitions into a coherent program. Our program reads commands from standard input, processing each command in turn. Five kinds of commands are supported.

(`VERIFY` *name formula filename*)
(`DEFINE` *name formals body measure inlinep filename*)
(`SKOLEM` *name bound-var free-var body*)
(`SWITCH` *name*)
(`FINISH` *filename*)

The `VERIFY` command is used to process a theorem event; the *name* is ignored by the system and is only an annotation for the user, and the *filename* indicates a file where a proof of the *formula* should be found. The `DEFINE` command is used to process a recursive function definition event; its arguments are the same as those for `admit-defun`, except that *filename* indicates a file where a list of proofs for the termination obligations should be found. The `SKOLEM` command is used to process a witnessing function definition, and its arguments are the same as those for `admit-witness`. The `SWITCH` command is used to begin using *name* as the proof checker. Finally, the `FINISH` command can be used to save the current session as a new Lisp image, and stop processing commands.

To process any one of these commands, we introduce the function try-to-accept-command. Most commands are not acceptable objects since they include file names, which are strings. Since we have said we will not call MILAWA-package functions such as MILAWA::tuplep on unacceptable objects, we implement safe-tuplep as an alternative.

Lisp Code
```
(defun-comp safe-tuplep (n x)
  (if (= n 0)
      (not x)
    (and (consp x)
         (safe-tuplep (- n 1) (cdr x)))))

(defun-comp try-to-accept-command (cmd)
  (cond
   ((not (consp cmd))
    (error "Invalid command ~A.~%" cmd))

   ((eq (car cmd) 'MILAWA::verify)
    (unless
     (and (safe-tuplep 4 cmd)
          (let ((name     (second cmd))
                (formula  (third cmd))
                (filename (fourth cmd)))
            (format t "> VERIFY ~A~%" name)
            (report-time "VERIFY"
             (and (acceptable-objectp name)
                  (acceptable-objectp formula)
                  (stringp filename)
                  (admit-theorem formula filename)))))
     (error "Invalid VERIFY: ~A" cmd)))

   ((eq (car cmd) 'MILAWA::DEFINE)
    (unless
     (and (safe-tuplep 7 cmd)
          (let ((name     (second cmd))
                (formals  (third cmd))
```

```
              (body     (fourth cmd))
              (measure  (fifth cmd))
              (inlinep  (sixth cmd))
              (filename (seventh cmd)))
          (format t "> DEFINE ~A~%" name)
          (report-time "DEFINE"
           (and (acceptable-objectp name)
                (acceptable-objectp formals)
                (acceptable-objectp body)
                (acceptable-objectp measure)
                (stringp filename)
                (admit-defun name formals body measure inlinep
                             filename)))))
  (error "Invalid DEFINE: ~A" cmd)))

((eq (car cmd) 'MILAWA::SKOLEM)
 (unless
  (and (safe-tuplep 5 cmd)
       (let ((name      (second cmd))
             (bound-var (third cmd))
             (free-vars (fourth cmd))
             (body      (fifth cmd)))
         (format t "> SKOLEM ~A~%" name)
         (report-time "SKOLEM"
          (and (acceptable-objectp name)
               (acceptable-objectp bound-var)
               (acceptable-objectp free-vars)
               (acceptable-objectp body)
               (admit-witness name bound-var free-vars body)))))
  (error "Invalid SKOLEM: ~A" cmd)))

((eq (car cmd) 'MILAWA::SWITCH)
 (unless
  (and (safe-tuplep 2 cmd)
       (let ((name (second cmd)))
         (format t "> SWITCH ~A~%" name)
         (report-time "SWITCH"
          (switch-proof-checker name))))
  (error "Invalid SWITCH: ~A" cmd)))
```

```
   ((eq (car cmd) 'MILAWA::FINISH)
    (unless
     (and (safe-tuplep 2 cmd)
          (let ((filename (second cmd)))
            (format t "> FINISH ~A~%" filename)
            (and (stringp filename)
                 (save-and-exit filename))))
     (error "Invalid FINISH: ~A" cmd)))

   (t
    (error "Invalid command: ~A" cmd))))
```

We then repeatedly call `try-to-accept-command` on the commands we read from the user. We read each command with `milawa-read-command`, which simply reads from standard input. If the end of file is reached before any `FINISH` command is encountered, we simply print a message and quit.

Lisp Code

```
(defun-comp try-to-accept-all-commands ()
  (let* ((*package* milawa-package)
         (cmd       (milawa-read-command)))
    (when (eq cmd unique-cons-for-eof)
      (format t "All commands have been accepted.~%")
      (quit))
    (try-to-accept-command cmd)
    (try-to-accept-all-commands)))
```

Our `main` function, which is the starting point for the Lisp images we create with `save-and-exit`, just prints a welcome message and calls `try-to-accept-all-commands`.

Lisp Code

```
(defun-comp main ()
  (format t "Milawa Proof Checker.  %")
  (try-to-accept-all-commands))
```

And with that, we use `save-and-exit` to create the executable for our base system.

Lisp Code

```
(save-and-exit "base")
```

Part II

Building Proofs

Chapter 5

Propositional Calculus

If we regard equality formulas as atomic propositions, then together the associativity, contraction, cut, expansion, and propositional schema rules form a propositional calculus which can be used to prove any tautology. But it would be difficult to carry out much propositional reasoning using the system presented in Chapter 4 since so many proof steps would be required.

In a mathematical logic, a *derived rule of inference* is an explanation of how, given certain inputs, a particular sequence of steps may be followed to obtain a desired proof. For instance, in our logic, if we are given a proof of `(por*` A B`)`, where A and B are any formulas, then may obtain a proof of `(por*` B A`)` using a derived rule we call *commutativity of or*—first, use the propositional schema to conclude `(por* (pnot*` A`)` A`)`; next use the cut rule to combine this with the given proof of `(por*` A B`)` to conclude `(por*` B A`)`.

In this chapter, we explain how we implement and reason about derived rules of inference, and develop several derived rules that allow propositional reasoning to be carried out more easily.

5.1 Implementing Derived Rules

Derived rules of inference can be implemented as functions which, given the necessary input proofs and formulas, construct the desired proof. This is a fully expansive style of proof building. Calling these functions is like writing proofs at a "high level," i.e., in terms of derived rules. But the resulting proofs are composed entirely of the primitive proof steps accepted by `logic.proofp`. A limitation of this approach is that, in practice, these proofs may be too large to construct and check, so we must be conscious of how many primitive steps our derivations will require.

Before implementing derived rules, we begin by writing proof-building functions for each primitive rule of inference. We generally use the prefix "build." when naming functions that construct proofs. Given any formula A, `build.propositional-schema` constructs an appeal which concludes `(por* (pnot*` A`)` A`)`. If A is a well-formed formula, then this appeal will be accepted by `logic.proofp` since the propositional schema is one of the primitive rules it allows.

Definition: `build.propositional-schema`

```
(pequal* (build.propositional-schema a)
         (list 'propositional-schema
               (logic.por (logic.pnot a) a)))
```

As another example, `build.expansion` implements the expansion rule. Given a formula, A, and a proof, x, of some formula B, it produces a proof of `(por*` A B`)`. Notice how the input proof, x, becomes a subproof of the newly constructed proof.

Definition: `build.expansion`

```
(pequal* (build.expansion a x)
         (list 'expansion
               (logic.por a (logic.conclusion x))
               (list x)))
```

Similarly, we can define `build.cut`, which, given proofs of `(por*` A B`)` and `(por* (pnot*` A`)` C`)`, builds a proof of `(por*` B C`)`.

Definition: `build.cut`

```
(pequal* (build.cut x y)
         (list 'cut
               (logic.por (logic.vrhs (logic.conclusion x))
                          (logic.vrhs (logic.conclusion y)))
               (list x y)))
```

After introducing similar constructors for the other primitive rules, we can begin implementing derived rules of inference. For instance, here is how we might implement commutativity of or.

Definition: build.commute-or

```
(pequal* (build.commute-or x)
         (build.cut x (build.propositional-schema
                         (logic.vlhs (logic.conclusion x)))))
```

As another example, the derived rule called *right expansion* allows us to derive (por* A B) given a proof of A—first, from the given proof, use expansion to conclude (por* B A); next, apply the commutativity of or rule to conclude (por* A B). We define build.right-expansion to perform these steps; as arguments, x should be the proof of A, and b should be the formula B.

Definition: build.right-expansion

```
(pequal* (build.right-expansion x b)
         (build.commute-or (build.expansion b x)))
```

To the caller, there is little difference between a derived rule and a primitive rule. For instance, in build.right-expansion above, we called upon build.commute-or just as we have called upon build.cut, build.expansion, and so on. This is useful since it allows us to begin describing proofs in more concise terms, i.e., we can now refer to commute or, right expansion, and so on, rather than using only primitive steps.

5.2 Reasoning about Derived Rules

These functions also play a key role in our fidelity proof. To justify the use of every proof-building function, f, we establish that when f is given valid inputs, then its result is (1) "well-typed"—it is a valid appeal, (2) "relevant"—it has the desired conclusion, and (3) "faithful"—it is accepted by `logic.proofp`. Since our derived rules never look inside input proofs to see which steps were used, but instead only consider the conclusion of each input proof, these lemmas are sufficient for reasoning about the composition of proof-building functions.

We now go into some detail about how this is done in our ACL2 proof plan. Our usual sequence for introducing a proof-building function, f, is as follows. First, we define f and prove these three properties. Then we "disable" f, which instructs ACL2 not to use its definition in later proofs, but instead to reason about f using only these lemmas.

To begin with, we establish these properties for the primitive proof-building functions. This work is somewhat different than proving the properties for derived rules since these functions are not written in terms of other proof-building functions, but rather are explicitly making appeals by consing together a method, a conclusion, and so on. Hence, to show these functions are well-typed, relevant, and faithful, we instruct ACL2 to use the definitions of functions which we would normally leave disabled, including our accessors for appeals (`logic.method`, `logic.conclusion`, . . .), our recognizers for appeals and proofs (`logic.appealp`, `logic.proofp`), and our step-checking functions (`logic.appeal-step-okp`, `logic.propositional-schema-okp`, `logic.cut-okp`, . . .).

As an example, here are our ACL2 theorems for the propositional schema.

ACL2 Code
```
(defthm logic.appealp-of-build.propositional-schema
  (implies (logic.formulap a)
           (logic.appealp (build.propositional-schema a))))

(defthm logic.conclusion-of-build.propositional-schema
  (equal (logic.conclusion (build.propositional-schema a))
         (logic.por (logic.pnot a) a)))

(defthm logic.proofp-of-build.propositional-schema
  (implies (logic.formula-atblp a atbl)
           (logic.proofp (build.propositional-schema a)
                         axioms thms atbl)))
```

The theorems for expansion are quite similar.

ACL2 Code
```
(defthm logic.appealp-of-build.expansion
  (implies (and (logic.formulap a)
                (logic.appealp x))
           (logic.appealp (build.expansion a x))))

(defthm logic.conclusion-of-build.expansion
  (equal (logic.conclusion (build.expansion a x))
         (logic.por a (logic.conclusion x))))

(defthm logic.proofp-of-build.expansion
  (implies (and (logic.formula-atblp a atbl)
                (logic.proofp x axioms thms atbl))
           (logic.proofp (build.expansion a x) axioms thms atbl)))
```

The theorems for cut are slightly more involved since, to be valid, the two input proofs must be related to each other in just the right way.

ACL2 Code

```
(defthm logic.appealp-of-build.cut
  (implies
   (and (logic.appealp x)
        (logic.appealp y)
        (equal (logic.fmtype (logic.conclusion x)) 'por*)
        (equal (logic.fmtype (logic.conclusion y)) 'por*)
        (equal (logic.fmtype (logic.vlhs (logic.conclusion y)))
               'pnot*)
        (equal (logic.vlhs (logic.conclusion x))
               (logic.~arg (logic.vlhs (logic.conclusion y)))))
   (logic.appealp (build.cut x y))))

(defthm logic.conclusion-of-cut
  (equal (logic.conclusion (build.cut x y))
         (logic.por (logic.vrhs (logic.conclusion x))
                    (logic.vrhs (logic.conclusion y)))))

(defthm logic.proofp-of-build.cut
  (implies
   (and (equal (logic.fmtype (logic.conclusion x)) 'por*)
        (equal (logic.fmtype (logic.conclusion y)) 'por*)
        (equal (logic.fmtype (logic.vlhs (logic.conclusion y)))
               'pnot*)
        (equal (logic.vlhs (logic.conclusion x))
               (logic.~arg (logic.vlhs (logic.conclusion y))))
        (logic.proofp x axioms thms atbl)
        (logic.proofp y axioms thms atbl))
   (logic.proofp (build.cut x y) axioms thms atbl)))
```

After we have proven analogous rules for each of the other primitive proof-building functions, we disable their definitions and also the definitions of `logic.appealp`, `logic.proofp`, and so on. When we introduce our functions for derived rules, such as `build.commute-or`, we always construct proofs using these proof-building functions, rather than by directly writing (list 'propositional-schema ...) and so on, so that we do not need to consider the definition of logic.proofp again.

How can we prove the three theorems for build.commute-or? The easiest example is the relevance theorem. Recall that the commutativity of or rule is intended to allow us to prove (por* B A) when given a proof of (por* A B), so we write its relevance theorem as the following goal for ACL2 to prove:

```
(equal (logic.conclusion (build.commute-or x))
       (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vlhs (logic.conclusion x))))
```

The ACL2 proof proceeds roughly as follows. First, by the definition of build.commute-or, the goal is equivalent to:

```
(equal (logic.conclusion
        (build.cut x (build.propositional-schema
                      (logic.vlhs (logic.conclusion x)))))
       (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vlhs (logic.conclusion x))))
```

Next, by the relevance theorem for cut, logic.conclusion-of-build.cut, we may reduce the goal to:

```
(equal (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vrhs (logic.conclusion
                               (build.propositional-schema
                                (logic.vlhs
                                 (logic.conclusion x))))))
       (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vlhs (logic.conclusion x))))
```

Now, we use the relevance theorem for the propositional schema, logic.conclusion-of-build.propositional-schema, to reduce the goal to:

```
(equal (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vrhs
                   (logic.por
                    (logic.pnot (logic.vlhs (logic.conclusion x)))
                    (logic.vlhs (logic.conclusion x)))))
       (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vlhs (logic.conclusion x))))
```

Finally, a trivial theorem, `(equal (logic.vrhs (logic.por x y)) y)`, can be used to reduce the goal to:

```
(equal (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vlhs (logic.conclusion x)))
       (logic.por (logic.vrhs (logic.conclusion x))
                  (logic.vlhs (logic.conclusion x))))
```

Which is trivially true by the reflexivity of `equal`.

It is slightly more work to prove the other theorems, because the well-typedness and faithfulness theorems `build.propositional-schema` and `build.cut` can only be used when certain hypotheses are shown to hold. But the basic approach is the same: we expand the definition of `build.commute-or` to recast the problem into simpler proof-building functions, and we then call upon our lemmas about these functions to prove the resulting goal.

In the end, we use ACL2 to prove:

ACL2 Code
```
(defthm logic.appealp-of-build.commute-or
  (implies (and (logic.appealp x)
                (equal (logic.fmtype (logic.conclusion x)) 'por*))
           (logic.appealp (build.commute-or x))))

(defthm logic.conclusion-of-build.commute-or
  (equal (logic.conclusion (build.commute-or x))
         (logic.por (logic.vrhs (logic.conclusion x))
                    (logic.vlhs (logic.conclusion x)))))

(defthm logic.proofp-of-build.commute-or
  (implies (and (logic.proofp x axioms thms atbl)
                (equal (logic.fmtype (logic.conclusion x)) 'por*))
           (logic.proofp (build.commute-or x) axioms thms atbl)))
```

At this point, reasoning about `build.commute-or` is no more difficult than reasoning about the primitives.

5.3 Simple Derivations

Since we will be writing many formulas in this chapter, we adopt a more concise, infix notation. We generally use upper-case italic letters, $A, B, \ldots$, to stand for formulas, and lower-case italic letters such as $a, b, \ldots$ for terms. We write $t_1 = t_2$ for `(pequal*` t_1 t_2`)`, $t_1 \neq t_2$ for `(pnot* (pequal*` t_1 t_2`))`, $\neg A$ for `(pnot*` A`)`, and $A \vee B$ for `(por*` A B`)`. We say $\vee$ associates to the right so $A \vee B \vee C$ means $A \vee (B \vee C)$, and $\neg$ binds more tightly than $\vee$ so $\neg A \vee B$ means $(\neg A) \vee B$.

We will also write derivations in a concise format. When, given a proof of the formulas $\mathit{premise}_1, \ldots, \mathit{premise}_n$, the rule allows us to obtain a proof of *conclusion*, we begin by writing

$$\begin{array}{l} \mathit{premise}_1 \\ \vdots \\ \mathit{premise}_n \\ \hline \mathit{conclusion}. \end{array}$$

We then explain how the conclusion is derived from the premises by writing a list of formulas and their justifications. Often, a particular formula follows from the previous formula (or, for rules such as Cut, from the previous two formulas), and so we will only mention which rule is being used. In other cases, we may label formulas so that we may refer to them later.

For instance, the derivation below describes `build.commute-or`. For every such derivation that we write, we introduce a proof-building function that carries out the described steps. We also prove this function is well-typed, relevant, and faithful, as described in the last section. Note that all of the derivations and formal theorems presented in this dissertation are transcribed from their implementation as functions. Since their correctness has been mechanically checked, any errors in our presentation are transcription errors.

Derived Rule 1. Commute or

$$\frac{A \vee B}{B \vee A}$$

Derivation. (2)

$A \vee B$	Given	
$\neg A \vee A$	Prop. schema	
$B \vee A$	Cut	□

We use these proof-building functions in our bootstrapping process to construct the fully expansive proofs of Milawa's fidelity for `logic.proofp` to check. Because of this, the number of proof steps introduced by each builder is a practical concern. The (2) above denotes this cost, and indicates that each use of this rule extends the input proof with two primitive steps.

As a special note to the reader, this dissertation presents so many derivations that it would be hard to remember them all. In the electronic version, our justifications are hyperlinks which can be followed to see the rule being used. In the paper version, the index topics *Derived Rules* and *Formal Theorems* may be useful.

We now present a number of simple derived rules. Finding these derivations is often an exercise for students in introductory logic classes, so this is all fairly routine. Since our rules of propositional logic are the same as Shoenfield [83] introduced, and have since then notably been used by Shankar [82], NQTHM [12], ACL2 [50], we have taken advantage of these resources when finding the derivations below.

Derived Rule 2. Right expansion

$$\frac{A}{A \lor B}$$

Derivation. (3)

$A \lor B$	Given
$B \lor A$	Expansion
$A \lor B$	Commute or

□

Derived Rule 3. Modus ponens

$$\frac{\begin{array}{l} A \\ \neg A \lor B \end{array}}{B}$$

Derivation. (5)

A	Given
$A \lor B$	Right expansion
$\neg A \lor B$	Given
$B \lor B$	Cut
B	Contraction

□

Derived Rule 4. Modus ponens 2

$$\frac{\begin{array}{l} \neg A \\ A \lor B \end{array}}{B}$$

Derivation. (5)

$\neg A$	Given
$\neg A \vee B$	Right expansion
$A \vee B$	Given
$B \vee B$	Cut
B	Contraction □

Derived Rule 5. Right associativity

$$\frac{(A \vee B) \vee C}{A \vee B \vee C}$$

Derivation. (8)

$(A \vee B) \vee C$	Given
$C \vee A \vee B$	Commute or
$(C \vee A) \vee B$	Associativity
$B \vee C \vee A$	Commute or
$(B \vee C) \vee A$	Associativity
$A \vee B \vee C$	Commute or □

Derived Rule 6. Cancel ¬¬

$$\frac{\neg\neg A}{A}$$

Derivation. (6)

$\neg\neg A$	Given
$\neg A \vee A$	Prop. schema
A	Modus ponens 2 □

Derived Rule 7. Insert ¬¬

$$\frac{A}{\neg\neg A}$$

Derivation. (8)

$\neg\neg A \vee \neg A$	Prop. schema

$\neg A \vee \neg\neg A$	Commute or
A	Given
$\neg\neg A$	Modus ponens $\square$

Derived Rule 8. Lhs insert $\neg\neg$

$$\frac{A \vee B}{\neg\neg A \vee B}$$

Derivation. (6)

$\neg\neg A \vee \neg A$	Prop. schema
$\neg A \vee \neg\neg A$	Commute or
$A \vee B$	Given
$B \vee \neg\neg A$	Cut
$\neg\neg A \vee B$	Commute or $\square$

Derived Rule 9. Lhs cancel $\neg\neg$

$$\frac{\neg\neg A \vee B}{A \vee B}$$

Derivation. (2)

$\neg A \vee A$	Prop. schema
$\neg\neg A \vee B$	Given
$A \vee B$	Cut $\square$

We call many derivations *disjoined*, by which we mean they perform some operation in the presence of an "extra" disjunct, usually named P. In logic classes, the proof of the deduction law generally involves proving the disjoined version of each primitive rule. It is straightforward to prove the disjoined rules for expansion and contraction.

Derived Rule 10. Disjoined left expansion

$$\frac{P \vee A}{P \vee B \vee A}$$

Derivation. (6)

$P \vee A$	Given
$A \vee P$	Commute or
$B \vee A \vee P$	Expansion
$(B \vee A) \vee P$	Associativity
$P \vee B \vee A$	Commute or □

Derived Rule 11. Disjoined contraction

$$\frac{P \vee A \vee A}{P \vee A}$$

Derivation. (6)

$P \vee A \vee A$	Given
$(P \vee A) \vee A$	Associativity
$A \vee P \vee A$	Commute or
$P \vee A \vee P \vee A$	Expansion
$(P \vee A) \vee P \vee A$	Associativity
$P \vee A$	Contraction □

Our derivation of the disjoined associativity and cut rules is considerably more involved, and we make use of some auxiliary derivations.

Derived Rule 12. Merge negatives

$$\frac{\begin{array}{l}\neg A\\ \neg B\end{array}}{\neg(A \vee B)}$$

Derivation. (16)

$\neg(A \vee B) \vee A \vee B$	Prop. schema
$(\neg(A \vee B) \vee A) \vee B$	Associativity

$B \vee \neg(A \vee B) \vee A$	Commute or
$\neg B$	Given
$\neg(A \vee B) \vee A$	Modus ponens 2
$A \vee \neg(A \vee B)$	Commute or
$\neg A$	Given
$\neg(A \vee B)$	Modus ponens 2 $\square$

Derived Rule 13. Merge implications lemma 1

$$\frac{\neg B \vee C}{A \vee C \vee \neg(A \vee B)}$$

Derivation. (10)

$\neg(A \vee B) \vee A \vee B$	Prop. schema
$(\neg(A \vee B) \vee A) \vee B$	Associativity
$B \vee \neg(A \vee B) \vee A$	Commute or
$\neg B \vee C$	Given
$(\neg(A \vee B) \vee A) \vee C$	Cut
$C \vee \neg(A \vee B) \vee A$	Commute or
$(C \vee \neg(A \vee B)) \vee A$	Associativity
$A \vee C \vee \neg(A \vee B)$	Commute or $\square$

Derived Rule 14. Merge implications lemma 2

$$\frac{\begin{array}{l} A \vee C \vee D \\ \neg A \vee C \end{array}}{D \vee C}$$

Derivation. (12)

$A \vee C \vee D$	Given
$\neg A \vee C$	Given
$(C \vee D) \vee C$	Cut
$C \vee C \vee D$	Commute or
$(C \vee C) \vee D$	Associativity
$D \vee C \vee C$	Commute or
$D \vee C$	Dj. contraction $\square$

Derived Rule 15. Merge implications

$$\frac{\begin{array}{l}\neg A \lor C \\ \neg B \lor C\end{array}}{\neg(A \lor B) \lor C}$$

Derivation. (22)

$\neg B \lor C$	Given
$A \lor C \lor \neg(A \lor B)$	Merge imp. lm. 1
$\neg A \lor C$	Given
$\neg(A \lor B) \lor C$	Merge imp. lm. 2 □

Derived Rule 16. Disjoined commute or lemma 1

$$\frac{P \lor A \lor B}{A \lor (B \lor A) \lor P}$$

Derivation. (9)

$P \lor A \lor B$	Given
$(P \lor A) \lor B$	Associativity
$A \lor (P \lor A) \lor B$	Expansion
$(A \lor P \lor A) \lor B$	Associativity
$B \lor A \lor P \lor A$	Commute or
$(B \lor A) \lor P \lor A$	Associativity
$((B \lor A) \lor P) \lor A$	Associativity
$A \lor (B \lor A) \lor P$	Commute or □

Derived Rule 17. Disjoined commute or

$$\frac{P \lor A \lor B}{P \lor B \lor A}$$

Derivation. (20)

$P \lor A \lor B$	Given
$A \lor (B \lor A) \lor P$	Dj. comm. or lm. 1
$B \lor A \lor (B \lor A) \lor P$	Expansion
$(B \lor A) \lor (B \lor A) \lor P$	Associativity

$((B \lor A) \lor B \lor A) \lor P$	Associativity
$P \lor (B \lor A) \lor B \lor A$	Commute or
$P \lor B \lor A$	Dj. contraction $\square$

Derived Rule 18. Disjoined assoc lemma 1a

$$\frac{P \lor A \lor D}{A \lor B \lor (C \lor D) \lor P}$$

Derivation. (10)

$P \lor A \lor D$	Given
$(P \lor A) \lor D$	Associativity
$D \lor P \lor A$	Commute or
$C \lor D \lor P \lor A$	Expansion
$(C \lor D) \lor P \lor A$	Associativity
$((C \lor D) \lor P) \lor A$	Associativity
$B \lor ((C \lor D) \lor P) \lor A$	Expansion
$(B \lor (C \lor D) \lor P) \lor A$	Associativity
$A \lor B \lor (C \lor D) \lor P$	Commute or $\square$

Derived Rule 19. Disjoined assoc lemma 1

$$\frac{}{\neg(A \lor D) \lor (A \lor B) \lor C \lor D}$$

Derivation. (15)

$\neg(A \lor D) \lor A \lor D$	Prop. schema
$A \lor B \lor (C \lor D) \lor \neg(A \lor D)$	Dj. assoc lm. 1a
$(A \lor B) \lor (C \lor D) \lor \neg(A \lor D)$	Associativity
$((A \lor B) \lor C \lor D) \lor \neg(A \lor D)$	Associativity
$\neg(A \lor D) \lor (A \lor B) \lor C \lor D$	Commute or $\square$

Derived Rule 20. Disjoined assoc lemma 2a

$$\frac{P \lor B \lor C}{A \lor B \lor (C \lor D) \lor P}$$

Derivation. (10)

$P \lor B \lor C$	Given
$(P \lor B) \lor C$	Associativity
$D \lor (P \lor B) \lor C$	Expansion
$(D \lor P \lor B) \lor C$	Associativity
$C \lor D \lor P \lor B$	Commute or
$(C \lor D) \lor P \lor B$	Associativity
$((C \lor D) \lor P) \lor B$	Associativity
$B \lor (C \lor D) \lor P$	Commute or
$A \lor B \lor (C \lor D) \lor P$	Expansion $\square$

Derived Rule 21. Disjoined assoc lemma 2

$$\frac{}{\neg(B \lor C) \lor (A \lor B) \lor C \lor D}$$

Derivation. (15)

$\neg(B \lor C) \lor B \lor C$	Prop. schema
$A \lor B \lor (C \lor D) \lor \neg(B \lor C)$	Dj. assoc lm. 2a
$(A \lor B) \lor (C \lor D) \lor \neg(B \lor C)$	Associativity
$((A \lor B) \lor C \lor D) \lor \neg(B \lor C)$	Associativity
$\neg(B \lor C) \lor (A \lor B) \lor C \lor D$	Commute or $\square$

Derived Rule 22. Disjoined assoc lemma 3a

$$\frac{}{\neg((A \lor D) \lor B \lor C) \lor (A \lor B) \lor C \lor D}$$

Derivation. (52)

$\neg(A \lor D) \lor (A \lor B) \lor C \lor D$	Dj. assoc lm. 1
$\neg(B \lor C) \lor (A \lor B) \lor C \lor D$	Dj. assoc lm. 2
$\neg((A \lor D) \lor B \lor C) \lor (A \lor B) \lor C \lor D$	Merge imp. $\square$

Derived Rule 23. Disjoined assoc lemma 3

$$\frac{(A \vee D) \vee B \vee C}{(A \vee B) \vee C \vee D}$$

Derivation. (57)

$\neg((A \vee D) \vee B \vee C) \vee (A \vee B) \vee C \vee D$	Dj. assoc lm. 3a
$(A \vee D) \vee B \vee C$	Given
$(A \vee B) \vee C \vee D$	Modus ponens $\square$

Derived Rule 24. Disjoined right associativity

$$\frac{P \vee (A \vee B) \vee C}{P \vee A \vee B \vee C}$$

Derivation. (86)

$P \vee (A \vee B) \vee C$	Given
$P \vee C \vee A \vee B$	Dj. commute or
$(P \vee C) \vee A \vee B$	Associativity
$(P \vee A) \vee B \vee C$	Dj. assoc lm. 3
$P \vee A \vee B \vee C$	Right assoc. $\square$

Derived Rule 25. Disjoined assoc lemma 4

$$\frac{(P \vee A) \vee B \vee C}{(P \vee C) \vee A \vee B}$$

Derivation. (97)

$(P \vee A) \vee B \vee C$	Given
$(P \vee A) \vee C \vee B$	Dj. commute or
$(P \vee C) \vee B \vee A$	Dj. assoc lm. 3
$(P \vee C) \vee A \vee B$	Dj. commute or $\square$

Derived Rule 26. Disjoined associativity

$$\frac{P \vee A \vee B \vee C}{P \vee (A \vee B) \vee C}$$

Derivation. (126)

$P \vee A \vee B \vee C$	Given	
$(P \vee A) \vee B \vee C$	Associativity	
$(P \vee C) \vee A \vee B$	Dj. assoc lm. 4	
$P \vee C \vee A \vee B$	Right assoc.	
$P \vee (A \vee B) \vee C$	Dj. commute or	$\square$

Derived Rule 27. Disjoined cut lemma 1

$$\frac{\begin{array}{l} P \vee A \vee B \\ P \vee \neg A \vee C \end{array}}{(B \vee P) \vee C \vee P}$$

Derivation. (21)

$P \vee A \vee B$	Given	
$(A \vee B) \vee P$	Commute or	
$A \vee B \vee P$	Right assoc.	(*1)
$P \vee \neg A \vee C$	Given	
$(\neg A \vee C) \vee P$	Commute or	
$\neg A \vee C \vee P$	Right assoc.	
$(B \vee P) \vee C \vee P$	Cut *1	$\square$

Derived Rule 28. Disjoined cut lemma 2

$$\frac{\begin{array}{l} P \vee A \vee B \\ P \vee \neg A \vee C \end{array}}{(B \vee C) \vee P \vee P}$$

Derivation. (78)

$P \vee A \vee B$	Given	
$P \vee \neg A \vee C$	Given	
$(B \vee P) \vee C \vee P$	Dj. cut lemma 1	
$(B \vee C) \vee P \vee P$	Dj. assoc lm. 3	$\square$

Derived Rule 29. Disjoined cut

$$\frac{\begin{array}{l} P \vee A \vee B \\ P \vee \neg A \vee C \end{array}}{P \vee B \vee C}$$

Derivation. (86)

$P \vee A \vee B$	Given
$P \vee \neg A \vee C$	Given
$(B \vee C) \vee P \vee P$	Dj. cut lemma 2
$(B \vee C) \vee P$	Dj. contraction
$P \vee B \vee C$	Commute or ☐

Next, for pragmatic reasons, it is useful to develop efficient functions to perform disjoined right expansion and modus ponens steps.

Derived Rule 30. Disjoined right expansion

$$\frac{P \vee A}{P \vee A \vee B}$$

Derivation. (7)

$P \vee A$	Given
$B \vee P \vee A$	Expansion
$(B \vee P) \vee A$	Associativity
$A \vee B \vee P$	Commute or
$(A \vee B) \vee P$	Associativity
$P \vee A \vee B$	Commute or ☐

Derived Rule 31. Disjoined modus ponens

$$\frac{\begin{array}{l} P \vee A \\ P \vee \neg A \vee B \end{array}}{P \vee B}$$

Derivation. (14)

$P \vee A$	Given

$B \lor P \lor A$	Expansion	
$(B \lor P) \lor A$	Associativity	
$A \lor B \lor P$	Commute or	(*1)
$P \lor \neg A \lor B$	Given	
$(P \lor \neg A) \lor B$	Associativity	
$B \lor P \lor \neg A$	Commute or	
$(B \lor P) \lor \neg A$	Associativity	
$\neg A \lor B \lor P$	Commute or	
$(B \lor P) \lor B \lor P$	Cut *1	
$B \lor P$	Contraction	
$P \lor B$	Commute or	$\square$

Derived Rule 32. Disjoined modus ponens 2

$$\begin{array}{l} P \lor \neg A \\ P \lor A \lor B \\ \hline P \lor B \end{array}$$

Derivation. (14)

$P \lor A \lor B$	Given	
$(P \lor A) \lor B$	Associativity	
$B \lor P \lor A$	Commute or	
$(B \lor P) \lor A$	Associativity	
$A \lor B \lor P$	Commute or	(*1)
$P \lor \neg A$	Given	
$B \lor P \lor \neg A$	Expansion	
$(B \lor P) \lor \neg A$	Associativity	
$\neg A \lor B \lor P$	Commute or	
$(B \lor P) \lor B \lor P$	Cut *1	
$B \lor P$	Contraction	
$P \lor B$	Commute or	$\square$

5.4 Recursive Derivations

The rules we have developed so far are very limited. More interesting derived rules are possible when proofs are constructed recursively. As a first example of a recursive derived rule, we develop a rule which allows us to repeatedly apply modus ponens. We write this rule as follows:

$$\frac{\begin{array}{l} A_1 \\ \vdots \\ A_n \\ \neg A_1 \vee \cdots \vee \neg A_n \vee B \end{array}}{B}$$

We adopt some conventions for the use of ellipses in disjunctions. When we write $A \vee B_1 \vee \cdots \vee B_n \vee C$, we intend to represent the formula which would be obtained by running

```
(logic.disjoin-formulas (list A B1 ... Bn C)).
```

That is, when $n = 0$, the formula is $A \vee C$; when $n = 1$, it is $A \vee B_1 \vee C$; when $n = 2$, it is $A \vee B_1 \vee B_2 \vee C$; and so on.

Parentheses may be used to group elided disjuncts into a right-associated subformula. For instance, consider $A \vee (B_1 \vee \cdots \vee B_n) \vee C$. In this case, when $n = 0$ the formula is $A \vee C$; when $n = 1$, it is $A \vee B_1 \vee C$; when $n = 2$, it is $A \vee (B_1 \vee B_2) \vee C$; and so on. We may also write $A_{1 \ldots n}$ as shorthand for $(A_1 \vee \cdots \vee A_n)$. For instance, $A \vee B_{1 \ldots n} \vee C$ means $A \vee (B_1 \vee \cdots \vee B_n) \vee C$.

Finally, there is no "empty formula," so when a formula is composed entirely of elided disjuncts, then implicitly at least one must be non-empty. For instance, if we use $A_1 \vee \cdots \vee A_n$ as a formula, then n must be at least one. Similarly, if we write $A_{1 \ldots n} \vee B_{1 \ldots m}$, then n and m may not simultaneously be zero.

We now return our attention to our rule for repeatedly applying modus ponens. We explain how to carry out this derivation by induction on n.

Derived Rule 33. Modus ponens list

$$\begin{array}{l} A_1 \\ \vdots \\ A_n \\ \neg A_1 \vee \cdots \vee \neg A_n \vee B \\ \hline B \end{array}$$

Derivation. $(5n)$

As a basis, if $n = 0$, then we have already been given a proof of B. Otherwise, $n > 0$, and we may recursively derive B given proofs of $A_2, \ldots, A_n$, and $\neg A_2 \vee \cdots \vee \neg A_n \vee B$. Since we have been given proofs of $A_2, \ldots, A_n$, we only need to derive $\neg A_2 \vee \cdots \vee \neg A_n \vee B$, which is easy:

A_1	Given
$\neg A_1 \vee \cdots \vee \neg A_n \vee B$	Given
$\neg A_2 \vee \cdots \vee \neg A_n \vee B$	Modus ponens $\square$

It is straightforward to implement this rule as the recursive function, build.-modus-ponens-list, which takes as arguments *b*, the formula B; *as*, the proofs of $A_1, \ldots, A_n$; and *base*, the proof of $\neg A_1 \vee \cdots \vee \neg A_n \vee B$.

Definition: build.modus-ponens-list

```
(pequal* (build.modus-ponens-list b as base)
         (if (consp as)
             (let ((step (build.modus-ponens (car as) base)))
               (build.modus-ponens-list b (cdr as) step))
           base))
```

Recursive derivations fit easily into our reasoning framework. By recreating the derivation above as an inductive proof in ACL2, we can prove that build.modus-ponens-list is well-typed, relevant, and faithful. The resulting ACL2 theorems are as follows.

ACL2 Code

```
(defthm logic.appealp-of-build.modus-ponens-list
  (implies (and (logic.formulap b)
                (logic.appeal-listp as)
                (logic.appealp base)
                (equal (logic.conclusion base)
                       (logic.disjoin-formulas
                        (app (logic.negate-formulas
                              (logic.strip-conclusions as))
                             (list b))))))
           (logic.appealp (build.modus-ponens-list b as base))))

(defthm logic.conclusion-of-build.modus-ponens-list
  (implies (and (logic.formulap b)
                (logic.appeal-listp as)
                (logic.appealp base)
                (equal (logic.conclusion base)
                       (logic.disjoin-formulas
                        (app (logic.negate-formulas
                              (logic.strip-conclusions as))
                             (list b))))))
       (equal (logic.conclusion (build.modus-ponens-list b as base))
              b)))

(defthm forcing-logic.proofp-of-build.modus-ponens-list
  (implies (and (logic.formulap b)
                (logic.appeal-listp as)
                (logic.appealp base)
                (equal (logic.conclusion base)
                       (logic.disjoin-formulas
                        (app (logic.negate-formulas
                              (logic.strip-conclusions as))
                             (list b))))
                (logic.proof-listp as axioms thms atbl)
                (logic.proofp base axioms thms atbl))
           (logic.proofp (build.modus-ponens-list b as base)
                         axioms thms atbl)))
```

We can similarly derive the rules Modus Ponens 2 List and Disjoined Modus Ponens List, but we omit the details since they are nearly identical to the above. The derivations add $5n$ and $14n$ proof steps, respectively.

Derived Rule 34. Modus ponens 2 list

$$\begin{array}{l} \neg A_1 \\ \vdots \\ \neg A_n \\ A_1 \vee \cdots \vee A_n \vee B \\ \hline B \end{array}$$

Derived Rule 35. Disjoined modus ponens list

$$\begin{array}{l} P \vee A_1 \\ \vdots \\ P \vee A_n \\ P \vee \neg A_1 \vee \cdots \vee \neg A_n \vee B \\ \hline P \vee B \end{array}$$

Another example of a recursively defined rule is Multi-assoc expansion. Note that this rule is quite efficient, expanding into at most $2i + 7$ proof steps.

Derived Rule 36. Multi-assoc expansion

$$\begin{array}{l} A_i \vee P \\ \hline (A_1 \vee \cdots \vee A_n) \vee P \end{array}$$

Derivation. $(\sim 2i + 7)$

As a basis, suppose $n = 1$. Then $i = 1$ and we have already been given the desired proof.

Otherwise, $n > 1$ and we consider two cases. If $i = 1$, then we may derive our goal as follows:

$A_1 \vee P$	Given
$A_1 \vee (A_2 \vee \cdots \vee A_n) \vee P$	Dj. left expansion
$(A_1 \vee \cdots \vee A_n) \vee P$	Associativity

Otherwise, $i > 1$ so we may recursively derive $(A_2 \vee \cdots \vee A_n) \vee P$ from $A_i \vee P$. Then,

$A_i \vee P$	Given
$(A_2 \vee \cdots \vee A_n) \vee P$	Recursive construction
$A_1 \vee (A_2 \vee \cdots \vee A_n) \vee P$	Expansion
$(A_1 \vee \cdots \vee A_n) \vee P$	Associativity $\square$

Definition: `build.multi-assoc-expansion`

```
(pequal* (build.multi-assoc-expansion x as)
         (if (and (consp as)
                  (consp (cdr as)))
             (if (equal (car as) (logic.vlhs (logic.conclusion x)))
                 ;; i = 1
                 (build.associativity
                  (build.disjoined-left-expansion
                   x
                   (logic.disjoin-formulas (cdr as))))
               ;; i > 1
               (build.associativity
                (build.expansion
                 (car as)
                 (build.multi-assoc-expansion x (cdr as)))))
           x))
```

5.5 Subsets

We now turn our attention to developing a more powerful rule of inference, the Generic Subset rule, which, from a proof of $A_1 \vee \cdots \vee A_n$, allows us to derive the formula $B_1 \vee \cdots \vee B_m$, so long as $\{A_1, \ldots, A_n\} \subseteq \{B_1, \ldots, B_m\}$.

This effort will require us to develop a few auxiliary rules. Our approach is adapted from Shankar's [82] development of the same derived rule. In particular, our Multi Expansion rule is what he called M1-proof, our Multi Or Expansion (Step) is his M2-Proof(-Step), our Generic Subset Step rule is a variant of his M3-Proof, and our Generic Subset rule is his M-Proof.

Derived Rule 37. Multi-expansion

$$\frac{A_i}{A_1 \vee \cdots \vee A_n}$$

Derivation. $(\sim i + 3)$

As a basis, if $n = 1$ then we are given our desired proof. Otherwise, suppose $n > 1$. If $i = 1$ then we have been given a proof of A_1, so we may follow these steps:

A_1	Given
$A_1 \vee \cdots \vee A_n$	Right expansion

Otherwise, $i > 1$ so we may recursively construct a proof of $A_2 \vee \cdots \vee A_n$ from A_i. Then,

A_i	Given
$A_2 \vee \cdots \vee A_n$	Recursive construction
$A_1 \vee \cdots \vee A_n$	Expansion □

Derived Rule 38. Multi-or expansion step

$$\frac{P \vee A_i}{P \vee A_1 \vee \cdots \vee A_n}$$

Derivation. $(\sim 6i + 7)$

As a basis, if $n = 1$ then we are given our desired proof.

Otherwise, suppose $n > 1$. If $i = 1$, we have been given $P \vee A_1$. Then,

$P \vee A_1$	Given
$P \vee A_1 \vee \cdots \vee A_n$	Dj. right expansion

Otherwise, from our proof of A_i we can recursively build a proof of $P \vee A_2 \vee \cdots \vee A_n$. Then,

A_i	Given
$P \vee (A_2 \vee \cdots \vee A_n)$	Recursive construction
$P \vee (A_1 \vee \cdots \vee A_n)$	Dj. left expansion $\square$

Derived Rule 39. Multi-or expansion

$$\frac{A_i \vee A_j}{A_1 \vee \cdots \vee A_n}$$

Derivation. ($\mathcal{O}(n)$)

As a basis, if $n = 1$ then $i, j = 1$ and we are given a proof of $A_1 \vee A_1$.

$A_1 \vee A_1$	Given
A_1	Contraction

Otherwise, if $i = 1$, we are given a proof of $A_1 \vee A_j$. Then,

$A_1 \vee A_j$	Given
$A_1 \vee \cdots \vee A_n$	Multi-or expansion step

Otherwise, if $j = 1$, we are given a proof of $A_i \vee A_1$. Then,

$A_i \vee A_1$	Given
$A_1 \vee A_i$	Commute or
$A_1 \vee \cdots \vee A_n$	Multi-or expansion step

Finally, if $i, j \neq 1$, then from $A_i \vee A_j$ we may recursively construct a proof of $A_2 \vee \cdots \vee A_n$. Then,

$A_i \vee A_j$	Given
$A_2 \vee \cdots \vee A_n$	Recursive construction
$A_1 \vee \cdots \vee A_n$	Expansion $\square$

Derived Rule 40. Generic subset step lemma 1

$$\frac{(P \vee A) \vee P}{P \vee A}$$

Derivation. (13)

$(P \vee A) \vee P$	Given
$P \vee P \vee A$	Commute or
$(P \vee P) \vee A$	Associativity
$A \vee P \vee P$	Commute or
$A \vee P$	Dj. contraction
$P \vee A$	Commute or $\square$

Derived Rule 41. Generic subset step

$$\frac{(A_i \vee A_j) \vee A_1 \vee \cdots \vee A_n}{A_1 \vee \cdots \vee A_n}$$

Derivation. ($\mathcal{O}(n^2)$)

$(A_i \vee A_j) \vee A_1 \vee \cdots \vee A_n$	Given
$(A_1 \vee \cdots \vee A_n) \vee A_i \vee A_j$	Commute or
$((A_1 \vee \cdots \vee A_n) \vee A_i) \vee A_j$	Associativity
$((A_1 \vee \cdots \vee A_n) \vee A_i) \vee A_1 \vee \cdots \vee A_n$	Multi-or exp. step
$(A_1 \vee \cdots \vee A_n) \vee A_i$	Generic subset step lm. 1
$(A_1 \vee \cdots \vee A_n) \vee A_1 \vee \cdots \vee A_n$	Multi-or exp. step
$A_1 \vee \cdots \vee A_n$	Contraction $\square$

Derived Rule 42. Generic subset

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

Derivation. ($\mathcal{O}(n^3)$)

Suppose $n = 1$. In this case, we are given a proof of A_1, and we know that $\{A_1\} \subseteq \{B_1, \ldots, B_m\}$. In other words, we are given a proof of B_i for some i. Then,

B_i	Given
$B_1 \vee \cdots \vee B_m$	Multi-expansion

Otherwise, suppose $n = 2$. Now we are given $A_1 \vee A_2$, which is the same as $B_i \vee B_j$ for some i, j. Then,

$B_i \vee B_j$	Given
$B_1 \vee \cdots \vee B_m$	Multi-or expansion

Finally, suppose $n \geq 3$. Let $C = A_1 \vee A_2$. Now $C \vee A_3 \vee \cdots \vee A_n$ is a disjunction of $n-1$ formulas, so we may recursively prove $C \vee B_1 \vee \cdots \vee B_m$ from a proof of $C \vee A_3 \vee \cdots \vee A_n$. Then,

$A_1 \vee \cdots \vee A_n$	Given
$(A_1 \vee A_2) \vee (A_3 \vee \cdots \vee A_n)$	Associativity
$C \vee (A_3 \vee \cdots \vee A_n)$	*Restated to introduce C*
$C \vee (B_1 \vee \cdots \vee B_m)$	Recursive construction
$(A_1 \vee A_2) \vee (B_1 \vee \cdots \vee B_m)$	*Restated to remove C*
$B_1 \vee \cdots \vee B_m$	Generic subset step $\square$

The generic subset rule is powerful, but it is not very efficient. In special cases, more efficient derivations are possible. One such case is reversing a disjunction. If our goal is to prove $A_n \vee \cdots \vee A_1$ from a proof of $A_1 \vee \cdots \vee A_n$, we can use a derivation which takes $\mathcal{O}(n)$ steps rather than $\mathcal{O}(n^3)$.

Derived Rule 43. Revappend disjunction

$$\frac{(T_1 \vee \cdots \vee T_n) \vee D_1 \vee \cdots \vee D_m}{T_n \vee \cdots \vee T_1 \vee D_1 \vee \cdots \vee D_m}$$

Derivation. $\mathcal{O}(n)$

We think of the T_i portion of the disjunction as "to do," and the D_i portion as "done". As a basis, if n is 0 or 1, then we have already been given the desired proof. Otherwise, assume $n \geq 2$, and consider two cases.

If $m = 0$, then we may recursively construct a proof of $T_n \vee \cdots \vee T_2 \vee A$ for any A, given a proof of $T_{2\ldots n} \vee A$ (this is well-founded since the number of T_i has been decreased). Then,

$T_1 \vee T_{2\ldots n}$	Given

$T_{2\dots n} \lor T_1$	Commute or
$T_n \lor \dots \lor T_1$	Recursive construction, $A \leftarrow T_1$

Otherwise, if $m \geq 1$, we may recursively construct a proof of $T_n \lor \dots \lor T_2 \lor A \lor D_{1\dots m}$, for any A, given a proof of $T_{2\dots n} \lor A \lor D_{1\dots m}$ (this is well-founded since the number of T_i has been decreased). Then,

$(T_1 \lor T_{2\dots n}) \lor D_{1\dots m}$	Given
$D_{1\dots m} \lor T_1 \lor T_{2\dots n}$	Commute or
$(D_{1\dots m} \lor T_1) \lor T_{2\dots n}$	Associativity
$T_{2\dots n} \lor (D_{1\dots m} \lor T_1)$	Commute or
$T_{2\dots n} \lor (T_1 \lor D_{1\dots m})$	Disjoined commute or
$T_n \lor \dots \lor T_2 \lor T_1 \lor D_{1\dots m}$	Recursive construction, $A \leftarrow T_1$ $\square$

Derived Rule 44. Rev disjunction

$$\frac{A_1 \lor \dots \lor A_n}{A_n \lor \dots \lor A_1}$$

Derivation. ($\mathcal{O}(n)$)

$A_1 \lor \dots \lor A_n$	Given
$A_n \lor \dots \lor A_1$	Revappend disjunction, $D \leftarrow \emptyset$ $\square$

The rev disjunction typically builds much smaller proofs than generic subset. As a simple empirical test, we created a one-step, axiomatic appeal which claims to prove `a1` $=$ `a1-prime` $\lor \dots \lor$ `a`n $=$ `a`n`-prime`, and then instructed each function to build a proof of the reversed disjunction, `a`n $=$ `a`n`-prime` $\lor \dots \lor$ `a1` $=$ `a1-prime`. We then measured the sizes of the resulting proofs with `rank`:

n	Generic Subset	Rev Disjunction	Savings
1	5	5	0%
2	38	38	0%
3	1,864	771	59%
5	15,194	3,605	76%
10	176,269	18,670	89%
20	2,042,969	83,000	96%
30	8,936,569	192,930	98%

Another special case of the Generic Subset rule is when $A_1, \ldots, A_n$ is an ordered subset of $B_1, \ldots, B_m$. In practice, this frequently arises when absurd and duplicate literals are removed from clauses, as mentioned in Chapter 7. In this case, we can again develop a custom derivation which often produces smaller proofs.

Derived Rule 45. Ordered subset aux

$$\frac{(D_1 \vee \cdots \vee D_k) \vee A_1 \vee \cdots \vee A_n}{B_m \vee \cdots \vee B_1 \vee D_1 \vee \cdots \vee D_k}$$

Where $A_1, \ldots, A_n$ is an ordered subset of $B_1, \ldots, B_m$.

Derivation. We will assume we can recursively perform this derivation when $n + m$ has decreased. As a basis, if $m = 0$ then $n = 0$. Since n and k may not be simultaneously zero, $k > 0$ and we have been given a proof of our goal. So, assume $m > 0$. Furthermore, if $n = 0$, then we know $k > 0$, and since the empty set is an ordered subset of $B_1, \ldots, B_{m-1}$, we may derive our goal as follows:

$D_{1\ldots k}$	Given
$B_{m-1} \vee \cdots \vee B_1 \vee D_{1\ldots k}$	Recursively, $D \leftarrow D_{1\ldots k}; B \leftarrow B_{1\ldots m-1}; A \leftarrow \emptyset$
$B_m \vee \cdots \vee B_1 \vee D_{1\ldots k}$	Expansion

So, for the remainder of the derivation, assume $n, m > 0$. If we further suppose $k = 0$, we have two cases:

A1. $A_1 = B_1$. Now, since $A_2, \ldots, A_n$ is an ordered subset of $B_2, \ldots, B_m$,

$B_1 \vee A_2 \vee \cdots \vee A_n$	Given
$B_m \vee \cdots \vee B_1$	Recursively, $D \leftarrow B_1; B \leftarrow B_{2\ldots m}; A \leftarrow A_{2\ldots n}$

A2. $A_1 \neq B_1$. Now, since $A_1, \ldots, A_n$ is an ordered subset of $B_2, \ldots, B_m$,

$A_1 \vee \cdots \vee A_n$	Given
$B_1 \vee A_1 \vee \cdots \vee A_n$	Expansion
$B_m \vee \cdots \vee B_1$	Recursively, $D \leftarrow B_1; B \leftarrow B_{2\ldots m}; A \leftarrow A_{1\ldots n}$

Otherwise, it must be that $n, m, k > 0$. We now have three cases:

B1. $A_1 = B_1$, $n = 1$. Now, since the empty set is an ordered subset of $B_2, \ldots, B_m$,

$D_{1\ldots k} \vee B_1$	Given
$B_1 \vee D_{1\ldots k}$	Commute or
$B_m \vee \cdots \vee B_1 \vee D_{1\ldots k}$	Recursively, $D \leftarrow B_1, D_{1\ldots k}; B \leftarrow B_{2\ldots m}; A \leftarrow \emptyset$

B2. $A_1 = B_1$, $n > 1$. Now, since $A_2, \ldots, A_n$ is an ordered subset of $B_2, \ldots, B_m$,

$D_{1\ldots k} \vee B_1 \vee A_{2\ldots n}$	Given
$D_{1\ldots k} \vee A_{2\ldots n} \vee B_1$	Disjoined commute or
$(D_{1\ldots k} \vee A_{2\ldots n}) \vee B_1$	Associativity
$B_1 \vee D_{1\ldots k} \vee A_{2\ldots n}$	Commute or
$(B_1 \vee D_{1\ldots k}) \vee A_{2\ldots n}$	Associativity
$B_m \vee \cdots \vee B_1 \vee D_{1\ldots k}$	Recursively, $D \leftarrow B_1, D_{1\ldots k}; B \leftarrow B_{2\ldots m}; A \leftarrow A_{2\ldots n}$

B3. $A_1 \neq B_1$. Now, since $A_1, \ldots, A_n$ is an ordered subset of $B_2, \ldots, B_m$,

$D_{1\ldots k} \vee A_{1\ldots n}$	Given
$B_1 \vee D_{1\ldots k} \vee A_{1\ldots n}$	Expansion
$(B_1 \vee D_{1\ldots k}) \vee A_{1\ldots n}$	Associativity
$B_m \vee \cdots \vee B_1 \vee D_{1\ldots k}$	Recursively, $D \leftarrow B_1, D_{1\ldots k}; B \leftarrow B_{2\ldots m}; A \leftarrow A_{1\ldots n}$ $\square$

Derived Rule 46. Ordered subset

$$\frac{A_1 \vee \cdots \vee A_n}{B_1 \vee \cdots \vee B_m}, \text{ where } A_1, \ldots, A_n \text{ is an ordered subset of } B_1, \ldots, B_m.$$

Derivation.

$A_1 \vee \cdots \vee A_n$	Given
$B_m \vee \cdots \vee B_1$	Ordered subset aux, $D \leftarrow \emptyset; B \leftarrow B_{1\ldots m}; A \leftarrow A_{1\ldots n}$
$B_1 \vee \cdots \vee B_m$	Rev disjunction $\square$

The ordered subset approach does not always outperform generic subset, but it does well when the sets involved are large. As a simple empirical test, we let A_i be the formula `a`i = `a`i`-prime`, B_i be `b`i = `b`i`-prime`, and C_i be `c`i = `c`i`-prime`. Then, beginning with a one-step, axiomatic appeal that concludes

$$A_1 \vee \cdots \vee A_n,$$

we instructed each builder to prove

$$C_1 \vee A_1 \vee B_1 \vee \cdots \vee C_n \vee A_n \vee B_n,$$

and measured the size of the resulting proofs with `rank`.

n	Generic Subset	Ordered Subset	Savings
1	64	818	Lose
2	523	6,626	Lose
3	8,557	17,378	Lose
4	25,380	33,074	Lose
5	53,714	55,555	Lose
6	103,552	79,298	23%
10	579,540	231,074	60%
15	2,306,725	532,034	77%
20	6,263,860	956,594	85%

After implementing these derivations as functions, it is straightforward to develop an "adaptive" function, which we call Disjoined Subset, that tries to construct whichever derivation seems likely to be the shortest.

– First, we determine if the subset, $A_1, \ldots, A_n$, is identical to the superset, in which case we can just reuse the input proof.

– Next, we check if the subset is the reverse of the superset, in which case we use rev disjunction, since it is a particularly efficient rule.

– Next, we see if we have an ordered subset. If this is the case, and heuristically the subset is at least of length 5 and the superset of length 10, we try the ordered subset builder, since it tends to be more efficient when the sets are larger.

– Otherwise, we use the generic subset builder.

5.6 Tautologies

Following the work of Shoenfield [83] and Shankar [82], we now introduce a rule which can be used to derive any propositional tautology. We regard equality formulas as atomic propositions and call them *atoms*. A *truth valuation*, v, assigns to every atom, A, a truth value, A^v. We extend truth valuations to arbitrary formulas by defining

$$(\neg F)^v = \text{not } F^v\text{, and}$$

$$(F \vee G)^v = F^v \text{ or } G^v.$$

A *tautology* is a formula whose every truth valuation is true. This notion of tautology is blind to the meaning of particular atoms, so formulas such as $\mathtt{x} = \mathtt{x}$ will not be thought of as tautologies even though they are true under every interpretation. Instead, these tautologies are formulas such as $A \vee \neg A$.

A *basic formula* is an atom or its negation. Given a basic formula, A, we define the complement of A, written $\overline{A}$, as follows: if A is an atom, then $\overline{A}$ is $\neg A$; otherwise A is the formula $\neg B$ for some atom B, and $\overline{A}$ is B. If a formula and its complement are both among a list of basic formulas, $F_1, \ldots, F_n$, then the disjunction $F_1 \vee \cdots \vee F_n$ is a tautology. On the other hand, if $F_1, \ldots, F_n$ is complement-free, then the truth valuation v, which assigns A^v to true exactly when $A = \overline{F_i}$ for some i, renders $(F_1 \vee \cdots \vee F_n)^v$ false.

We now describe the tautology-checking algorithm. Given a list of formulas, $T_1, \ldots, T_n$, and a complement-free list of basic formulas, $D_1, \ldots, D_m$, $TC(T_{1\ldots n}, D_{1\ldots m})$ determines whether $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is a tautology. To determine whether a particular formula, F, is a tautology, we can then simply run $TC([F], [])$.

In this algorithm, the formulas T_i are considered "to do" while the formulas D_i are considered "done". At each step, we work toward converting T_1 into an equivalent disjunction of basic formulas, which we then move into the done list. If at any point the complement of some D_i is generated, then the formula $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is a tautology; otherwise, if we run out of T_i without finding any complements, it is not a tautology.

$$\begin{aligned}
TC([], D_{1\ldots m}) &\triangleq \text{false} \\
TC([x_1 = x_2, T_{2\ldots n}], D_{1\ldots m}) &\triangleq x_1 \neq x_2 \in D_{1\ldots m} \text{ or } TC(T_{2\ldots n}, [x_1 = x_2, D_{1\ldots m}]) \\
TC([x_1 \neq x_2, T_{2\ldots n}], D_{1\ldots m}) &\triangleq x_1 = x_2 \in D_{1\ldots m} \text{ or } TC(T_{2\ldots n}, [x_1 \neq x_2, D_{1\ldots m}]) \\
TC([\neg\neg A, T_{2\ldots n}], D_{1\ldots m}) &\triangleq TC([A, T_{2\ldots n}], D_{1\ldots m}) \\
TC([\neg(A \vee B), T_{2\ldots n}], D_{1\ldots m}) &\triangleq TC([\neg A, T_{2\ldots n}], D_{1\ldots m}) \text{ and } TC([\neg B, T_{2\ldots n}], D_{1\ldots m}) \\
TC([A \vee B, T_{2\ldots n}], D_{1\ldots m}) &\triangleq TC([A, B, T_{2\ldots n}], D_{1\ldots m})
\end{aligned}$$

The termination of TC is justified by the measure $\sum_{i=1\ldots n} \text{SIZE}(T_i)$, where the size of a formula F, $\text{SIZE}(F)$, be defined as follows. If F is an atomic formula, then $\text{SIZE}(F) = 1$. Meanwhile, $\text{SIZE}(\neg G) = 1 + \text{SIZE}(G)$ and $\text{SIZE}(G \vee H) = 1 + \text{SIZE}(G) + \text{SIZE}(H)$.

The following theorem establishes that TC is correct with respect to our notion of tautologies.

Theorem 5.1. *If $TC(T_{1\ldots n}, D_{1\ldots m})$, then $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is a tautology.*

Proof. The proof is by induction according to the recursive definition of TC. As a basis, if n is zero, then $TC(T_{1\ldots n}, D_{1\ldots m})$ is false and there is nothing to show.

If T_1 is the equality $x_1 = x_2$, there are two cases. First, if $x_1 \neq x_2 \in D_{1\ldots m}$, then our goal formula, $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$, contains both $x_1 = x_2$ and $x_1 \neq x_2$. So, since any truth valuation v must assign one of these formulas to true, we see that $(T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m)^v$ is true for all valuations, and hence is a tautology. Otherwise, if $x_1 \neq x_2 \notin D_{1\ldots m}$, we may inductively assume that $T_2 \vee \cdots \vee$

$T_n \vee T_1 \vee D_1 \vee \cdots \vee D_m$ is a tautology. But then, trivially, $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is also a tautology.

If T_1 is $x_1 \neq x_2$, the proof is nearly identical to the above, so we omit it.

If T_1 is $\neg\neg A$, we may inductively assume that $A \vee T_2 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is a tautology. But $(\neg\neg A)^v = A^v$, so trivially $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is also a tautology.

If T_1 is $\neg(A \vee B)$, let P be the formula $T_2 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$. We may inductively assume that $\neg A \vee P$ and $\neg B \vee P$ are both tautologies, and our goal is to show $\neg(A \vee B) \vee P$ is a tautology. Let v be an arbitrary truth valuation so we only need to show $(\neg(A \vee B) \vee P)^v$ is true. This is trivial when P^v is true, so assume P^v is false; now we need to show $(\neg(A \vee B))^v$ is true. Since $\neg A \vee P$ and $\neg B \vee P$ are tautologies, $(\neg A \vee P)^v$ and $(\neg B \vee P)^v$ must be true, so $(\neg A)^v$ and $(\neg B)^v$ are true, so A^v and B^v are false, so $(A \vee B)^v$ is false, so $\neg(A \vee B)^v$ is true, which was our goal.

Finally, if T_1 is $A \vee B$, then we may inductively assume $(A \vee B) \vee T_2 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is a tautology. Then, trivially, $T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$ is also a tautology. □

Furthermore, we can explain how to derive a formal proof in our logic of any formula accepted by TC.

Derived Rule 47. Tautology lemma

$$\frac{}{T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m}, \text{ where } TC(T_{1\ldots n}, D_{1\ldots m}) \text{ is true.}$$

Derivation. Since $TC(T_{1\ldots n}, D_{1\ldots m})$, $n > 0$, so consider cases on T_1.

1. T_1 is $x_1 = x_2$. If $\neg T_1 \in D_{1\ldots m}$, let D_i be $\neg T_1$. Now,

$D_i \vee T_1$	Propositional schema
$T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$	Multi-or expansion

Otherwise, $TC(T_{2...n}, [T_1, D_{1...m}])$ is true, so $T_2 \vee \cdots \vee T_n \vee T_1 \vee D_{1...m}$ may be recursively derived. Our goal then follows by the generic subset rule.

2. T_1 is $x_1 \neq x_2$. If $(x_1 = x_2) \in D_{1...m}$, let D_i be $x_1 = x_2$. Now,

$T_i \vee D_i$	Propositional schema
$T_1 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$	Multi-or expansion

   Otherwise, $TC(T_{2...n}, [T_1, D_{1...m}])$ is true, so $T_2 \vee \cdots \vee T_n \vee T_1 \vee D_{1...m}$ may be recursively derived. As before, our goal follows from the generic subset rule.

3. T_1 is $\neg\neg A$. Now $TC([A, T_{2...n}], D_{1...m})$ is true. If $n = 1$ and $m = 0$, then we may recursively derive A. Now, our goal is $\neg\neg A$, which may be derived with the insert $\neg\neg$ rule. Otherwise, we may recursively derive $A \vee T_2 \vee \cdots \vee T_n \vee D_1 \vee \cdots \vee D_m$, and our goal follows from the lhs insert $\neg\neg$ rule.

4. T_1 is $\neg(A \vee B)$. Now, $TC([\neg A, T_{2...n}], D_{1...m})$ and $TC([\neg B, T_{2...n}], D_{1...m})$ are both true. If $n = 1$ and $m = 0$, then we may recursively derive $\neg A$ and $\neg B$, and our goal, $\neg(A \vee B)$, follows from the merge negatives rule. Otherwise, we may recursively derive $\neg A \vee T_2 \vee \cdots \vee T_n \vee D_{1...m}$ and $\neg B \vee T_2 \vee \cdots \vee T_n \vee D_{1...m}$, and our goal follows from the merge implications rule.

5. T_1 is $A \vee B$. Now, $TC([A, B, T_{2...n}], D_{1...m})$ is true. If $n = 1$ and $m = 0$, then we may recursively derive $A \vee B$, which is our goal. Otherwise, we may recursively derive $A \vee B \vee T_2 \vee \cdots \vee T_n \vee D_{1...m}$, and by the associativity rule we obtain our goal. $\square$

Derived Rule 48. Tautology

$$\frac{}{A}, \text{ where } TC([A], []) \text{ is true}$$

Derivation. This is a trivial consequence of the tautology lemma, setting $T \leftarrow [A]$ and $D \leftarrow \emptyset$. □

5.7 Equivalence Substitution

In this section we develop an equivalence substitution rule. This again follows Shoenfield's [83] original presentation and Shankar's [82] implementation for his system, except that we optimize the derivation to emit smaller proofs.

To begin, the logical connectives for implication, conjunction, and equivalence can be introduced as abbreviations for other formulas.

Abbreviation	Meaning
$A \rightarrow B$	$\neg A \vee B$
$A \wedge B$	$\neg(\neg A \vee \neg B)$
$A \leftrightarrow B$	$(A \rightarrow B) \wedge (B \rightarrow A)$

Given these abbreviations, we can develop some simple rules for working with conjunctions.

Derived Rule 49. First conjunct

$$\frac{\neg(\neg A \vee \neg B)}{A}$$

Derivation. (13)

$\neg A \vee A$	Prop. schema
$A \vee \neg A$	Commute or
$\neg B \vee A \vee \neg A$	Expansion
$(\neg B \vee A) \vee \neg A$	Associativity
$\neg A \vee \neg B \vee A$	Commute or
$(\neg A \vee \neg B) \vee A$	Associativity
$\neg(\neg A \vee \neg B)$	Given
A	Modus ponens 2 □

Derived Rule 50. Second conjunct

$$\frac{\neg(\neg A \vee \neg B)}{B}$$

Derivation. (8)

$\neg B \vee B$	Prop. schema
$\neg A \vee \neg B \vee B$	Expansion
$(\neg A \vee \neg B) \vee B$	Associativity
$\neg(\neg A \vee \neg B)$	Given
B	Modus ponens 2 $\square$

Derived Rule 51. Conjoin

$$\frac{\begin{array}{l} A \\ B \end{array}}{\neg(\neg A \vee \neg B)}$$

Derivation. (16)

$\neg(\neg A \vee \neg B) \vee \neg A \vee \neg B$	Prop. schema
$(\neg(\neg A \vee \neg B) \vee \neg A) \vee \neg B$	Associativity
$\neg B \vee \neg(\neg A \vee \neg B) \vee \neg A$	Commute or
B	Given
$\neg(\neg A \vee \neg B) \vee \neg A$	Modus ponens
$\neg A \vee \neg(\neg A \vee \neg B)$	Commute or
A	Given
$\neg(\neg A \vee \neg B)$	Modus ponens $\square$

Informally, the equivalence substitution rule is the following. Let G be obtained from F by replacing some occurrences of A_i with $A_i{}'$, respectively. Then, given proofs of $A_1 \leftrightarrow A_1{}'$, ..., and $A_n \leftrightarrow A_n{}'$, we may derive $F \leftrightarrow G$. More precisely, given formulas F and G, and a list of equivalence formulas, $\mathit{equivs} = [A_1 \leftrightarrow A_1{}', \ldots, A_n \leftrightarrow A_n{}']$, we define $\textsc{es}(F, G, \mathit{equivs})$ to return true when the rule may be applied, and false otherwise. In particular, we begin with

$$\textsc{es}(F, F, \mathit{equivs}) \triangleq \text{true},$$
$$\textsc{es}(F, G, [..., F \leftrightarrow G, ...]) \triangleq \text{true},$$

otherwise, as special cases, we have

$$\textsc{es}(\neg A, \neg B, \mathit{equivs}) \triangleq \textsc{es}(A, B, \mathit{equivs}),$$
$$\textsc{es}(A \vee B, C \vee D, \mathit{equivs}) \triangleq \textsc{es}(A, C, \mathit{equivs}) \text{ and } \textsc{es}(B, D, \mathit{equivs}),$$

and otherwise, $\textsc{es}(F, G, \mathit{equivs})$ is false.

Derived Rule 52. Equivalence substitution

$$\frac{\begin{array}{l} A_1 \leftrightarrow A_1{}' \\ \vdots \\ A_n \leftrightarrow A_n{}' \end{array}}{F \leftrightarrow G} \quad \text{where } \textsc{es}(F, G, [A_1 \leftrightarrow A_1{}', \ldots, A_n \leftrightarrow A_n{}']) \text{ is true}$$

Derivation. Let *equivs* be $[A_1 \leftrightarrow A_1{}', \ldots, A_n \leftrightarrow A_n{}']$, and consider the cases where $\textsc{es}(F, G, \mathit{equivs})$ is true.

1. Suppose $F = G$. Then our goal is $F \leftrightarrow F$. But $F \rightarrow F$ is just $\neg F \vee F$, so we have:

   $F \rightarrow F$ Propositional schema
   $F \leftrightarrow F$ Conjoin (the above with itself)

2. Suppose $F \leftrightarrow G$ occurs in *equivs*. Now, one of our premises is $F \leftrightarrow G$, so we have been given a proof of our goal.

3. Suppose $F = \neg A$, $G = \neg B$, and $\textsc{es}(A, B, \mathit{equivs})$ is true. We may recursively derive $A \leftrightarrow B$. Now, to derive our goal,

$A \rightarrow B \wedge B \rightarrow A$	Recursive construction	(*1)
$\neg B \vee A$	Second conjunct	
$A \vee \neg B$	Commute or	
$\neg A \rightarrow \neg B$	Lhs insert $\neg\neg$	(*2)
$\neg A \vee B$	First conjunct *1	
$B \vee \neg A$	Commute or	
$\neg B \rightarrow \neg A$	Lhs insert $\neg\neg$	
$\neg A \leftrightarrow \neg B$	Conjoin *2	

4. Suppose $F = A \vee B$, $G = C \vee D$, $\textsc{es}(A, C, \mathit{equivs})$ is true, and $\textsc{es}(B, D, \mathit{equivs})$ is true. We may recursively derive $A \leftrightarrow B$ and $C \leftrightarrow D$. Now, to derive our goal, $A \vee B \leftrightarrow C \vee D$,

$A \rightarrow C \wedge C \rightarrow A$	Recursive construction	(*1)
$B \rightarrow D \wedge D \rightarrow B$	Recursive construction	(*2)
$A \rightarrow C$	First conjunct *1	
$A \rightarrow (C \vee D)$	Disjoined right expansion	
$B \rightarrow D$	First conjunct *2	
$B \rightarrow (C \vee D)$	Disjoined left expansion	
$(A \vee B) \rightarrow (C \vee D)$	Merge implications	(*3)
$C \rightarrow A$	Second conjunct *1	
$C \rightarrow (A \vee B)$	Disjoined right expansion	
$D \rightarrow B$	Second conjunct *2	
$D \rightarrow (A \vee B)$	Disjoined left expansion	
$(C \vee D) \rightarrow (A \vee B)$	Merge implications	
$(A \vee B) \leftrightarrow (C \vee D)$	Conjoin *3	$\square$

Chapter 6

Equality

The tools developed in the last chapter are a good start toward making our proof-checking system usable, but propositional reasoning alone is insufficient to prove even simple formulas such as `t` $=$ `t`. In this chapter, we develop additional derived rules which allow us to carry out equality reasoning.

6.1 Simple Derivations

We begin with some simple rules of inference which allow us to manipulate equality formulas. First, we have two completely trivial rules for instantiating the reflexivity and equality axioms with our choice of terms.

Derived Rule 53. Reflexivity

$$\frac{}{a = a}$$

Derivation. (2)

`x` $=$ `x`	Axiom reflexivity
$a = a$	Instantiation

□

Derived Rule 54. Equality

$$\frac{}{a_1 \neq b_1 \vee a_2 \neq b_2 \vee a_1 \neq a_2 \vee b_1 = b_2}$$

Derivation. (2)

$\mathtt{x1} \neq \mathtt{y1} \vee \mathtt{x2} \neq \mathtt{y2} \vee \mathtt{x1} \neq \mathtt{x2} \vee \mathtt{y1} = \mathtt{y2}$	Axiom equality	
$a_1 \neq b_1 \vee a_2 \neq b_2 \vee a_1 \neq a_2 \vee b_1 = b_2$	Instantiation	□

Unlike the derivations in Chapter 5, these rules make use of axioms. In our ACL2 proof plan, we account for this by constraining the axioms in the faithfulness theorem. For instance, to show the faithfulness of `build.reflexivity`—the function that implements our reflexivity rule—we begin by introducing a zero-ary function, `axiom-reflexivity`, which simply returns `(pequal* x x)`. We use a zero-ary function, rather than writing the formula explicitly, so that we can disable the function in later proofs. Then, in the faithfulness theorem for `build.reflexivity`, we require that this formula is among the axioms given to `logic.proofp`, as follows.

ACL2 Code
```
(defthm logic.proofp-of-build.reflexivity
  (implies (and (logic.termp a)
                (logic.term-atblp a atbl)
                (memberp (axiom-reflexivity) axioms))
           (logic.proofp (build.reflexivity a) axioms thms atbl)))
```

To have our functions emit shorter derivations, a standard trick is to prove a theorem ahead of time which, combined with instantiation and modus ponens, will provide the desired manipulation. This way, even if the proof of this theorem is quite long, derivations which make use of the theorem can be short. Our first example of this is in commuting equalities. We first prove the theorem $\mathtt{x} = \mathtt{y} \rightarrow \mathtt{y} = \mathtt{x}$. Our proof of this theorem takes thirty-one primitive steps, but this is a one-time cost and, afterward, instantiating the theorem to derive $b = a$ from $a = b$ takes only seven steps.

Formal Theorem 1. Commutativity of =

$$\mathtt{x} \neq \mathtt{y} \vee \mathtt{y} = \mathtt{x}$$

Proof.

$x = x$	Reflexivity	
$x \neq y \vee x = x$	Expansion	(*1)
$x \neq y \vee x \neq x \vee x \neq x \vee y = x$	Equality	
$x \neq y \vee x \neq x \vee y = x$	Dj. mp *1	
$x \neq y \vee y = x$	Dj. mp *1	□

Derived Rule 55. Commute =

$$\frac{a = b}{b = a}$$

Derivation. (7)

$x \neq y \vee y = x$	Th. comm. =	
$a \neq b \vee b = a$	Instantiation	
$a = b$	Given	
$b = a$	Modus ponens	□

Of course, the proofs emitted by commute = will only be valid in histories where the theorem has been previously admitted. In our ACL2 proofs of faithfulness, we handle this much like we handled axioms. First, we write a zero-ary function, `theorem-commutativity-of-pequal`, that returns the formula $x \neq y \vee y = x$. Then, in the faithfulness theorem for `build.commute-pequal`—our function that implements the commute = rule—we require that this theorem be among the theorems given to `logic.proofp`, as follows.

ACL2 Code

```
(defthm logic.proofp-of-build.commute-pequal
  (implies (and (logic.appealp x)
                (logic.proofp x axioms thms atbl)
                (equal (logic.fmtype (logic.conclusion x)) 'pequal*)
                (memberp (theorem-commutativity-of-pequal) thms))
           (logic.proofp (build.commute-pequal x)
                                   axioms thms atbl)))
```

Theorems can also be useful in disjoined rules. The pattern here is to instantiate the theorem, expand it with the extra disjunct, and then use disjoined modus ponens to obtain the goal.

Derived Rule 56. Disjoined commute =

$$\frac{P \vee a = b}{P \vee b = a}$$

Derivation. (17)

$x \neq y \vee y = x$	Th. comm. =
$a \neq b \vee b = a$	Instantiation
$P \vee a \neq b \vee b = a$	Expansion
$P \vee a = b$	Given
$P \vee b = a$	Dj. modus ponens □

Now, without much further commentary, we present a number of rules that make use of these techniques.

Derived Rule 57. Commute ≠

$$\frac{a \neq b}{b \neq a}$$

Derivation. (9)

$x \neq y \vee y = x$	Th. comm. =
$y = x \vee x \neq y$	Commute or
$a = b \vee b \neq a$	Instantiation
$a \neq b$	Given
$b \neq a$	Modus ponens 2 □

Derived Rule 58. Disjoined commute $\neq$

$$\frac{P \vee a \neq b}{P \vee b \neq a}$$

Derivation. (19)

$x \neq y \vee y = x$	Th. comm. =
$y = x \vee x \neq y$	Commute or
$a = b \vee b \neq a$	Instantiation
$P \vee a = b \vee b \neq a$	Expansion
$P \vee a \neq b$	Given
$P \vee b \neq a$	Dj. mp2 □

Formal Theorem 2. Substitute into $\neq$

$$x = y \vee z \neq x \vee z \neq y$$

Proof.

$y = y$	Reflexivity
$z \neq x \vee y = y$	Expansion
$z \neq x \vee y \neq y \vee z \neq y \vee x = y$	Equality
$z \neq x \vee z \neq y \vee x = y$	Dj. modus ponens
$(z \neq x \vee z \neq y) \vee x = y$	Associativity
$x = y \vee z \neq x \vee z \neq y$	Commute or □

Derived Rule 59. Substitute into $\neq$

$$\frac{\begin{array}{l} a \neq b \\ c = a \end{array}}{c \neq b}$$

Derivation. (12)

$x = y \vee z \neq x \vee z \neq y$	Th. sub. into $\neq$
$a = b \vee c \neq a \vee c \neq b$	Instantiation
$a \neq b$	Given
$c \neq a \vee c \neq b$	Modus ponens 2
$c = a$	Given

$c \neq b$	Modus ponens	$\square$

Derived Rule 60. Disjoined substitute into $\neq$ lemma 1

$$\frac{P \lor a \neq b}{P \lor c \neq a \lor c \neq b}$$

Derivation. (17)

$\mathrm{x} = \mathrm{y} \lor \mathrm{z} \neq \mathrm{x} \lor \mathrm{z} \neq \mathrm{y}$	Th. sub. into $\neq$	
$a = b \lor c \neq a \lor c \neq b$	Instantiation	
$P \lor a = b \lor c \neq a \lor c \neq b$	Expansion	
$P \lor a \neq b$	Given	
$P \lor c \neq a \lor c \neq b$	Dj. mp2	$\square$

Derived Rule 61. Disjoined substitute into $\neq$

$$\frac{\begin{array}{l} P \lor a \neq b \\ P \lor c = a \end{array}}{P \lor c \neq b}$$

Derivation. (31)

$P \lor c \neq a \lor c \neq b$	Dj. sub. into $\neq$ lm. 1	
$P \lor c = a$	Given	
$P \lor c \neq b$	Dj. modus ponens	$\square$

Formal Theorem 3. Transitivity of $=$

$$\mathrm{x} \neq \mathrm{y} \lor \mathrm{y} \neq \mathrm{z} \lor \mathrm{x} = \mathrm{z}$$

Proof.

$\mathrm{x} \neq \mathrm{y} \lor \mathrm{y} = \mathrm{x}$	Th. comm. $=$	
$\mathrm{x} \neq \mathrm{y} \lor \mathrm{y} \neq \mathrm{z} \lor \mathrm{y} = \mathrm{x}$	Dj. left expansion	
$(\mathrm{x} \neq \mathrm{y} \lor \mathrm{y} \neq \mathrm{z}) \lor \mathrm{y} = \mathrm{x}$	Associativity	(*1)
$\mathrm{y} \neq \mathrm{z} \lor \mathrm{y} = \mathrm{z}$	Prop. schema	
$\mathrm{x} \neq \mathrm{y} \lor \mathrm{y} \neq \mathrm{z} \lor \mathrm{y} = \mathrm{z}$	Expansion	

$(\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z}) \lor \mathtt{y} = \mathtt{z}$	Associativity	(*2)
$\mathtt{y} = \mathtt{y}$	Reflexivity	
$(\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z}) \lor \mathtt{y} = \mathtt{y}$	Expansion	(*3)
$\mathtt{y} \neq \mathtt{x} \lor \mathtt{y} \neq \mathtt{z} \lor \mathtt{y} \neq \mathtt{y} \lor \mathtt{x} = \mathtt{z}$	Equality	
$(\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z}) \lor \mathtt{y} \neq \mathtt{x} \lor \mathtt{y} \neq \mathtt{z} \lor \mathtt{y} \neq \mathtt{y} \lor \mathtt{x} = \mathtt{z}$	Expansion	
$(\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z}) \lor \mathtt{y} \neq \mathtt{z} \lor \mathtt{y} \neq \mathtt{y} \lor \mathtt{x} = \mathtt{z}$	Dj. mp *1	
$(\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z}) \lor \mathtt{y} \neq \mathtt{y} \lor \mathtt{x} = \mathtt{z}$	Dj. mp *2	
$(\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z}) \lor \mathtt{x} = \mathtt{z}$	Dj. mp *3	
$\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z} \lor \mathtt{x} = \mathtt{z}$	Right assoc.	□

Derived Rule 62. Transitivity of =

$$\begin{array}{l} a = b \\ b = c \\ \hline a = c \end{array}$$

Derivation. (12)

$\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z} \lor \mathtt{x} = \mathtt{z}$	Th. trans. =	
$a \neq b \lor b \neq c \lor a = c$	Instantiation	
$a = b$	Given	
$b \neq c \lor a = c$	Modus ponens	
$b = c$	Given	
$a = c$	Modus ponens	□

Derived Rule 63. Disjoined transitivity of =

$$\begin{array}{l} P \lor a = b \\ P \lor b = c \\ \hline P \lor a = c \end{array}$$

Derivation. (31)

$\mathtt{x} \neq \mathtt{y} \lor \mathtt{y} \neq \mathtt{z} \lor \mathtt{x} = \mathtt{z}$	Th. trans. =	
$a \neq b \lor b \neq c \lor a = c$	Instantiation	
$P \lor a \neq b \lor b \neq c \lor a = c$	Expansion	
$P \lor a = b$	Given	
$P \lor b \neq c \lor a = c$	Dj. modus ponens	
$P \lor b = c$	Given	
$P \lor a = c$	Dj. modus ponens	□

Formal Theorem 4. Not t or not nil

$$\texttt{x} \neq \texttt{t} \lor \texttt{x} \neq \texttt{nil}$$

Proof.

$\texttt{t} \neq \texttt{nil}$	Axiom t not nil
$\texttt{x} \neq \texttt{t} \lor \texttt{t} \neq \texttt{nil}$	Expansion
$\texttt{x} \neq \texttt{t} \lor \texttt{x} = \texttt{t}$	Prop. schema
$\texttt{x} \neq \texttt{t} \lor \texttt{x} \neq \texttt{nil}$	Dj. sub. into $\neq$ □

Derived Rule 64. Not nil from t

$$\frac{a = \texttt{t}}{a \neq \texttt{nil}}$$

Derivation. (7)

$\texttt{x} \neq \texttt{t} \lor \texttt{x} \neq \texttt{nil}$	Th. not t or nnil
$a \neq \texttt{t} \lor a \neq \texttt{nil}$	Instantiation
$a = \texttt{t}$	Given
$a \neq \texttt{nil}$	Modus ponens □

Derived Rule 65. Disjoined not nil from t

$$\frac{P \lor a = \texttt{t}}{P \lor a \neq \texttt{nil}}$$

Derivation. (17)

$\texttt{x} \neq \texttt{t} \lor \texttt{x} \neq \texttt{nil}$	Th. not t or nnil
$a \neq \texttt{t} \lor a \neq \texttt{nil}$	Instantiation
$P \lor a \neq \texttt{t} \lor a \neq \texttt{nil}$	Expansion
$P \lor a = \texttt{t}$	Given
$P \lor a \neq \texttt{nil}$	Dj. modus ponens □

Derived Rule 66. Not t from nil

$$\frac{a = \texttt{nil}}{a \neq \texttt{t}}$$

Derivation. (9)

$\texttt{x} \neq \texttt{t} \vee \texttt{x} \neq \texttt{nil}$	Th. not t or nnil
$\texttt{x} \neq \texttt{nil} \vee \texttt{x} \neq \texttt{t}$	Commute or
$a \neq \texttt{nil} \vee a \neq \texttt{t}$	Instantiation
$a = \texttt{nil}$	Given
$a \neq \texttt{t}$	Modus ponens □

Derived Rule 67. Disjoined not t from nil

$$\frac{P \vee a = \texttt{nil}}{P \vee a \neq \texttt{t}}$$

Derivation. (19)

$\texttt{x} \neq \texttt{t} \vee \texttt{x} \neq \texttt{nil}$	Th. not t or nnil
$\texttt{x} \neq \texttt{nil} \vee \texttt{x} \neq \texttt{t}$	Commute or
$a \neq \texttt{nil} \vee a \neq \texttt{t}$	Instantiation
$P \vee a \neq \texttt{nil} \vee a \neq \texttt{t}$	Expansion
$P \vee a = \texttt{nil}$	Given
$P \vee a \neq \texttt{t}$	Dj. modus ponens □

6.2 Term-Level Equality

The primitive function `equal` provides a term-level version of the formula-level `pequal*`. We now derive many simple rules for working with the `equal` function. We begin with reflexivity.

Formal Theorem 5. Reflexivity of equal

$$\texttt{(equal x x)} = \texttt{t}$$

Proof.

$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{t}$	Ax. eq., same
$\texttt{x} \neq \texttt{x} \vee \texttt{(equal x x)} = \texttt{t}$	Instantiation
$\texttt{x} = \texttt{x}$	Reflexivity
$\texttt{(equal x x)} = \texttt{t}$	Modus ponens

□

Derived Rule 68. Equal reflexivity

$$\frac{}{\texttt{(equal } a\ \ a\texttt{)} = \texttt{t}}$$

Derivation. (2)

$\texttt{(equal x x)} = \texttt{t}$	Th. refl. equal
$\texttt{(equal } a\ \ a\texttt{)} = \texttt{t}$	Instantiation

□

This is the first derived rule we have introduced which mentions a particular function. When we use the Milawa proof checker, we know the arity of `equal` will always be two since it is part of the initial history and no event can change the arity of an existing function. But this knowledge is not part of the definition of `logic.proofp`, which requires that the formulas involved in proofs are well-formed with respect to an arity table. In our ACL2 proof of faithfulness, we account for this by constraining the arity of `equal`, as follows.

ACL2 Code

```
(defthm logic.proofp-of-build.equal-reflexivity
  (implies (and (logic.termp a)
                (logic.term-atblp a atbl)
                (equal (cdr (lookup 'equal atbl)) 2)
                (memberp (theorem-reflexivity-of-equal) thms))
           (logic.proofp (build.equal-reflexivity a)
                         axioms thms atbl)))
```

Since `equal` always returns `t` or `nil`, it is useful to have some rules that capture its Boolean nature.

Formal Theorem 6. Equal nil or t

$$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff
$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{t}$	Ax. eq., same
$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$	Cut □

Derived Rule 69. Equal t from not nil

$$\frac{\texttt{(equal } a \; b\texttt{)} \neq \texttt{nil}}{\texttt{(equal } a \; b\texttt{)} = \texttt{t}}$$

Derivation. (7)

$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$	Th. equal nil or t
$\texttt{(equal } a \; b\texttt{)} = \texttt{nil} \vee \texttt{(equal } a \; b\texttt{)} = \texttt{t}$	Instantiation
$\texttt{(equal } a \; b\texttt{)} \neq \texttt{nil}$	Given
$\texttt{(equal } a \; b\texttt{)} = \texttt{t}$	Modus ponens 2 □

Derived Rule 70. Disjoined equal t from not nil

$$\frac{P \vee \texttt{(equal } a \; b\texttt{)} \neq \texttt{nil}}{P \vee \texttt{(equal } a \; b\texttt{)} = \texttt{t}}$$

Derivation. (17)

$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$	Th. equal nil or t
$\texttt{(equal } a \; b\texttt{)} = \texttt{nil} \vee \texttt{(equal } a \; b\texttt{)} = \texttt{t}$	Instantiation
$P \vee \texttt{(equal } a \; b\texttt{)} = \texttt{nil} \vee \texttt{(equal } a \; b\texttt{)} = \texttt{t}$	Expansion
$P \vee \texttt{(equal } a \; b\texttt{)} \neq \texttt{nil}$	Given
$P \vee \texttt{(equal } a \; b\texttt{)} = \texttt{t}$	Dj. mp2 □

Derived Rule 71. Equal nil from not t

$$\frac{\texttt{(equal}\ a\ b\texttt{)} \neq \texttt{t}}{\texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}}$$

Derivation. (9)

$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$	Th. equal nil or t
$\texttt{(equal x y)} = \texttt{t} \vee \texttt{(equal x y)} = \texttt{nil}$	Commute or
$\texttt{(equal}\ a\ b\texttt{)} = \texttt{t} \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}$	Instantiation
$\texttt{(equal}\ a\ b\texttt{)} \neq \texttt{t}$	Given
$\texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}$	Modus ponens 2 $\square$

Derived Rule 72. Disjoined equal nil from not t

$$\frac{P \vee \texttt{(equal}\ a\ b\texttt{)} \neq \texttt{t}}{P \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}}$$

Derivation. (19)

$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$	Th. equal nil or t
$\texttt{(equal x y)} = \texttt{t} \vee \texttt{(equal x y)} = \texttt{nil}$	Commute or
$\texttt{(equal}\ a\ b\texttt{)} = \texttt{t} \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}$	Instantiation
$P \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{t} \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}$	Expansion
$P \vee \texttt{(equal}\ a\ b\texttt{)} \neq \texttt{t}$	Given
$P \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{nil}$	Dj. mp2 $\square$

It is also convenient to have rules which allow us to move from `equal` to `pequal*`, and vice-versa.

Derived Rule 73. Equal from =

$$\frac{a = b}{\texttt{(equal}\ a\ b\texttt{)} = \texttt{t}}$$

Derivation. (7)

$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{t}$	Ax. eq., same
$a \neq b \vee \texttt{(equal}\ a\ b\texttt{)} = \texttt{t}$	Instantiation
$a = b$	Given

$\texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Modus ponens	$\square$

Derived Rule 74. Disjoined equal from =

$$\frac{P \vee a = b}{P \vee \texttt{(equal } a\ b\texttt{)} = \texttt{t}}$$

Derivation. (17)

$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{t}$	Ax. eq., same	
$a \neq b \vee \texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Instantiation	
$P \vee a \neq b \vee \texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Expansion	
$P \vee a = b$	Given	
$P \vee \texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Dj. modus ponens	$\square$

Derived Rule 75. = from equal

$$\frac{\texttt{(equal } a\ b\texttt{)} = \texttt{t}}{a = b}$$

Derivation. (16)

$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff	
$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{x} = \texttt{y}$	Commute or	
$\texttt{(equal } a\ b\texttt{)} = \texttt{nil} \vee a = b$	Instantiation	(*1)
$\texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Given	
$\texttt{(equal } a\ b\texttt{)} \neq \texttt{nil}$	Not nil from t	
$a = b$	Mp2 *1	$\square$

Derived Rule 76. Disjoined = from equal

$$\frac{P \vee \texttt{(equal } a\ b\texttt{)} = \texttt{t}}{P \vee a = b}$$

Derivation. (36)

$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff
$\texttt{(equal x y)} = \texttt{nil} \vee \texttt{x} = \texttt{y}$	Commute or

$\texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil} \vee a = b$	Instantiation	
$P \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil} \vee a = b$	Expansion	(*1)
$P \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{t}$	Given	
$P \vee \texttt{(equal }\ a\ \ b\texttt{)} \neq \texttt{nil}$	Dj. not nil from t	
$P \vee a = b$	Dj. mp2 *1	$\square$

Derived Rule 77. Not equal from $\neq$

$$\frac{a \neq b}{\texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}}$$

Derivation. (7)

$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff	
$a = b \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}$	Instantiation	
$a \neq b$	Given	
$\texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}$	Modus ponens 2	$\square$

Derived Rule 78. Disjoined not equal from $\neq$

$$\frac{P \vee a \neq b}{P \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}}$$

Derivation. (17)

$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff	
$a = b \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}$	Instantiation	
$P \vee a = b \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}$	Expansion	
$P \vee a \neq b$	Given	
$P \vee \texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}$	Dj. mp2	$\square$

Derived Rule 79. $\neq$ from not equal

$$\frac{\texttt{(equal }\ a\ \ b\texttt{)} = \texttt{nil}}{a \neq b}$$

Derivation. (18)

$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{t}$	Ax. eq., same

$(\texttt{equal x y}) = \texttt{t} \lor \texttt{x} \neq \texttt{y}$	Commute or	
$(\texttt{equal}\ a\ b) = \texttt{t} \lor a \neq b$	Instantiation	(*1)
$(\texttt{equal}\ a\ b) = \texttt{nil}$	Given	
$(\texttt{equal}\ a\ b) \neq \texttt{t}$	Not t from nil	
$a \neq b$	Mp2 *1	$\square$

Derived Rule 80. Disjoined $\neq$ from not equal

$$\frac{P \lor (\texttt{equal}\ a\ b) = \texttt{nil}}{P \lor a \neq b}$$

Derivation. (38)

$\texttt{x} \neq \texttt{y} \lor (\texttt{equal x y}) = \texttt{t}$	Ax. eq., same	
$(\texttt{equal x y}) = \texttt{t} \lor \texttt{x} \neq \texttt{y}$	Commute or	
$(\texttt{equal}\ a\ b) = \texttt{t} \lor a \neq b$	Instantiation	
$P \lor (\texttt{equal}\ a\ b) = \texttt{t} \lor a \neq b$	Expansion	(*1)
$P \lor (\texttt{equal}\ a\ b) = \texttt{nil}$	Given	
$P \lor (\texttt{equal}\ a\ b) \neq \texttt{t}$	Dj. not t from nil	
$P \lor a \neq b$	Dj. mp2 *1	$\square$

We have already addressed the reflexivity of `equal`. Now we develop some rules about its commutativity and transitivity.

Formal Theorem 7. Symmetry of equal

$$(\texttt{equal x y}) = (\texttt{equal y x})$$

Proof.

$\texttt{x} = \texttt{y} \lor (\texttt{equal x y}) = \texttt{nil}$	Ax. eq. when diff	(*1)
$\texttt{y} = \texttt{x} \lor (\texttt{equal y x}) = \texttt{nil}$	Instantiation	
$\texttt{y} = \texttt{x} \lor \texttt{nil} = (\texttt{equal y x})$	Dj. commute $=$	
$\texttt{nil} = (\texttt{equal y x}) \lor \texttt{y} = \texttt{x}$	Commute or	
$\texttt{nil} = (\texttt{equal y x}) \lor \texttt{x} = \texttt{y}$	Dj. commute $=$	
$\texttt{x} = \texttt{y} \lor \texttt{nil} = (\texttt{equal y x})$	Commute or	
$\texttt{x} = \texttt{y} \lor (\texttt{equal x y}) = (\texttt{equal y x})$	Dj. trans. $=$ *1	(*2)
$\texttt{x} \neq \texttt{y} \lor (\texttt{equal x y}) = \texttt{t}$	Ax. eq., same	(*3)
$\texttt{y} \neq \texttt{x} \lor (\texttt{equal y x}) = \texttt{t}$	Instantiation	

$\texttt{y} \neq \texttt{x} \vee \texttt{t} = \texttt{(equal y x)}$	Dj. commute $=$	
$\texttt{t} = \texttt{(equal y x)} \vee \texttt{y} \neq \texttt{x}$	Commute or	
$\texttt{t} = \texttt{(equal y x)} \vee \texttt{x} \neq \texttt{y}$	Dj. commute $\neq$	
$\texttt{x} \neq \texttt{y} \vee \texttt{t} = \texttt{(equal y x)}$	Commute or	
$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{(equal y x)}$	Dj. trans. $=$ *3	(*4)
$\texttt{(equal x y)} = \texttt{(equal y x)}$ $\vee \texttt{(equal x y)} = \texttt{(equal y x)}$	Cut *2, *4	
$\texttt{(equal x y)} = \texttt{(equal y x)}$	Contraction	$\square$

Derived Rule 81. Commute equal

$$\frac{\texttt{(equal } a \ b\texttt{)} = \texttt{t}}{\texttt{(equal } b \ a\texttt{)} = \texttt{t}}$$

Derivation. (14)

$\texttt{(equal x y)} = \texttt{(equal y x)}$	Th. symmetry of eq.	
$\texttt{(equal } b \ a\texttt{)} = \texttt{(equal } a \ b\texttt{)}$	Instantiation	
$\texttt{(equal } a \ b\texttt{)} = \texttt{t}$	Given	
$\texttt{(equal } b \ a\texttt{)} = \texttt{t}$	Trans. $=$	$\square$

Derived Rule 82. Disjoined commute equal

$$\frac{P \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}}{P \vee \texttt{(equal } b \ a\texttt{)} = \texttt{t}}$$

Derivation. (34)

$\texttt{(equal x y)} = \texttt{(equal y x)}$	Th. symmetry of eq.	
$\texttt{(equal } b \ a\texttt{)} = \texttt{(equal } a \ b\texttt{)}$	Instantiation	
$P \vee \texttt{(equal } b \ a\texttt{)} = \texttt{(equal } a \ b\texttt{)}$	Expansion	
$P \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}$	Given	
$P \vee \texttt{(equal } b \ a\texttt{)} = \texttt{t}$	Dj. trans. $=$	$\square$

Formal Theorem 8. Transitivity of equal

$$\texttt{(equal x y)} \neq \texttt{t} \vee \texttt{(equal y z)} \neq \texttt{t} \vee \texttt{(equal x z)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{y} \lor \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff	
$\texttt{x} = \texttt{y} \lor \texttt{(equal x y)} \neq \texttt{t}$	Dj. not t from nil	(*1)
$\texttt{(equal y z)} \neq \texttt{t} \lor \texttt{x} = \texttt{y} \lor \texttt{(equal x y)} \neq \texttt{t}$	Expansion	
$(\texttt{(equal y z)} \neq \texttt{t} \lor \texttt{x} = \texttt{y}) \lor \texttt{(equal x y)} \neq \texttt{t}$	Associativity	
$\texttt{(equal x y)} \neq \texttt{t} \lor \texttt{(equal y z)} \neq \texttt{t} \lor \texttt{x} = \texttt{y}$	Commute or	
$(\texttt{(equal x y)} \neq \texttt{t} \lor \texttt{(equal y z)} \neq \texttt{t}) \lor \texttt{x} = \texttt{y}$	Associativity	(*2)
$\texttt{y} = \texttt{z} \lor \texttt{(equal y z)} \neq \texttt{t}$	Instantiation *1	
$\texttt{(equal y z)} \neq \texttt{t} \lor \texttt{y} = \texttt{z}$	Commute or	
$\texttt{(equal x y)} \neq \texttt{t} \lor \texttt{(equal y z)} \neq \texttt{t} \lor \texttt{y} = \texttt{z}$	Expansion	
$(\texttt{(equal x y)} \neq \texttt{t} \lor \texttt{(equal y z)} \neq \texttt{t}) \lor \texttt{y} = \texttt{z}$	Associativity	
$(\texttt{(equal x y)} \neq \texttt{t} \lor \texttt{(equal y z)} \neq \texttt{t}) \lor \texttt{x} = \texttt{z}$	Dj. trans. = *2	(*3)
$(\texttt{(equal x y)} \neq \texttt{t} \lor \texttt{(equal y z)} \neq \texttt{t})$ $\quad \lor \texttt{(equal x z)} = \texttt{t}$	Dj. eq. from =	
$\texttt{(equal x y)} \neq \texttt{t}$ $\quad \lor \texttt{(equal y z)} \neq \texttt{t} \lor \texttt{(equal x z)} = \texttt{t}$	Right assoc.	□

Derived Rule 83. Transitivity of equal

$$\begin{array}{l}\texttt{(equal } a\ b\texttt{)} = \texttt{t} \\ \texttt{(equal } b\ c\texttt{)} = \texttt{t} \\ \hline \texttt{(equal } a\ c\texttt{)} = \texttt{t}\end{array}$$

Derivation. (12)

$\texttt{(equal x y)} \neq \texttt{t}$ $\quad \lor \texttt{(equal y z)} \neq \texttt{t} \lor \texttt{(equal x z)} = \texttt{t}$	Th. trans. equal	
$\texttt{(equal } a\ b\texttt{)} \neq \texttt{t}$ $\quad \lor \texttt{(equal } b\ c\texttt{)} \neq \texttt{t} \lor \texttt{(equal } a\ c\texttt{)} = \texttt{t}$	Instantiation	
$\texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Given	
$\texttt{(equal } b\ c\texttt{)} \neq \texttt{t} \lor \texttt{(equal } a\ c\texttt{)} = \texttt{t}$	Modus ponens	
$\texttt{(equal } b\ c\texttt{)} = \texttt{t}$	Given	
$\texttt{(equal } a\ c\texttt{)} = \texttt{t}$	Modus ponens	□

Derived Rule 84. Disjoined transitivity of equal

$$\begin{array}{l}P \lor \texttt{(equal } a\ b\texttt{)} = \texttt{t} \\ P \lor \texttt{(equal } b\ c\texttt{)} = \texttt{t} \\ \hline P \lor \texttt{(equal } a\ c\texttt{)} = \texttt{t}\end{array}$$

Derivation. (31)

$\texttt{(equal x y)} \neq \texttt{t}$	Th. trans. equal
$\quad \vee \texttt{(equal y z)} \neq \texttt{t} \vee \texttt{(equal x z)} = \texttt{t}$	
$\texttt{(equal } a \ b\texttt{)} \neq \texttt{t}$	Instantiation
$\quad \vee \texttt{(equal } b \ c\texttt{)} \neq \texttt{t} \vee \texttt{(equal } a \ c\texttt{)} = \texttt{t}$	
$P \vee \texttt{(equal } a \ b\texttt{)} \neq \texttt{t}$	Expansion
$\quad \vee \texttt{(equal } b \ c\texttt{)} \neq \texttt{t} \vee \texttt{(equal } a \ c\texttt{)} = \texttt{t}$	
$P \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}$	Given
$P \vee \texttt{(equal } b \ c\texttt{)} \neq \texttt{t} \vee \texttt{(equal } a \ c\texttt{)} = \texttt{t}$	Dj. modus ponens
$P \vee \texttt{(equal } b \ c\texttt{)} = \texttt{t}$	Given
$P \vee \texttt{(equal } a \ c\texttt{)} = \texttt{t}$	Dj. modus ponens □

It is easy enough to prove that a constant is equal to itself: if we want to prove $\texttt{(pequal* } c \ c\texttt{)}$, we can simply use the reflexivity rule, and if we want to prove $\texttt{(equal } c \ c\texttt{)} = \texttt{t}$, we can use equal reflexivity. But if c_1 and c_2 are different constants, how can we prove they are not equal? Since equal is one of our primitive functions, we can do this with the base evaluation rule.

Derived Rule 85. $\neq$ constants

$$\frac{}{c_1 \neq c_2}, \text{ when } c_1 \text{ and } c_2 \text{ are distinct constants}$$

Derivation. (19)

$\texttt{(equal } c1 \ c2\texttt{)} = \texttt{nil}$	Base eval
$c1 \neq c2$	$\neq$ from not equal □

6.3 Equality Substitution

Rules such as substitute into $\neq$ and transitivity of $=$ give us a mechanism for performing equality substitution on "whole terms." We now develop some more general rules which allow us to substitute equalities into subterms.

To begin with, we develop some particularly useful rules which allow us to substitute equal terms into the arguments of function applications and lambda abbreviations. For function applications, this is entirely straightforward.

Derived Rule 86. = by arguments

$$\frac{\begin{array}{l} t_1 = s_1 \\ \vdots \\ t_n = s_n \end{array}}{(f\ \ t_1\ \ \ldots\ \ t_n) = (f\ \ s_1\ \ \ldots\ \ s_n)}$$

Derivation. By the functional equality rule, we may derive

$$t_1 = s_1 \to \cdots \to t_n = s_n \to (f\ \ t_1\ \ \ldots\ \ t_n) = (f\ \ s_1\ \ \ldots\ \ s_n).$$

Then, since we are given proofs of $t_1 = s_1, \ldots, t_n = s_n$, by modus ponens list, we may obtain our goal, $(f\ \ t_1\ \ \ldots\ \ t_n) = (f\ \ s_1\ \ \ldots\ \ s_n)$. $\square$

Argument substitution in lambda abbreviations is more difficult. Our basic approach is to β-reduce the lambda under each set of arguments, then show the resulting terms are equal. This second part is done with the dual substitution rule, but before we present this rule we need to mention a property of substitution.

Theorem 6.1. *If* $\textsc{freevars}(t) \subseteq \{v_1, \ldots, v_n\}$, *then*

$$(t/[v_1 \leftarrow s_1, \ldots, v_1 \leftarrow s_n])/\sigma = t/[v_1 \leftarrow s_1/\sigma, \ldots, v_n \leftarrow s_n/\sigma].$$

Proof by structural induction on t.

If t is a constant, then $(t/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma$ and $t/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma]$ are both t.

If t is a variable, then since $\textsc{freevars}(t) \subseteq \{v_1, \ldots, v_n\}$, $t = v_i$ for some i. So $(t/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma$ is s_i/σ, and $t/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma]$ is also s_i/σ.

If t is $(f\ \ a_1\ \ \ldots\ \ a_m)$, then we may inductively assume

$$(a_i/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma = a_i/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma],$$

and hence,

$$\begin{aligned}
(t/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma &= ((f \;\; a_1 \;\; \ldots \;\; a_m)/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma \\
&= ((f \;\; a_1/[v_{1\ldots n} \leftarrow s_{1\ldots n}] \;\; \ldots \;\; a_m/[v_{1\ldots n} \leftarrow s_{1\ldots n}]))/\sigma \\
&= (f \;\; (a_1/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma \;\; \ldots \;\; (a_m/[v_{1\ldots n} \leftarrow s_{1\ldots n}])/\sigma) \\
&= (f \;\; a_1/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma] \;\; \ldots \;\; a_m/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma])) \\
&= (f \;\; a_1 \;\; \ldots \;\; a_m))/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma] \\
&= t/[v_{1\ldots n} \leftarrow s_{1\ldots n}/\sigma].
\end{aligned}$$

If t is `((lambda` $(x_1 \;\; \ldots \;\; x_m) \;\; \beta) \;\; a_1 \;\; \ldots \;\; a_m)$, then the situation is analogous to the case for functions, so we omit the details. □

Derived Rule 87. Dual substitution lemma 1

$$\frac{\begin{array}{l} a = b \\ b = d \\ c = d \end{array}}{a = c}$$

Derivation. (31)

$a = b$	Given	
$b = d$	Given	
$a = d$	Trans. $=$	(*1)
$c = d$	Given	
$d = c$	Commute $=$	
$a = c$	Trans. $=$ *1	□

Derived Rule 88. Dual substitution

$$\frac{\begin{array}{l} t_1 = s_1 \\ \vdots \\ t_n = s_n \end{array}}{x/[v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n] = x/[v_1 \leftarrow s_1, \ldots, v_n \leftarrow s_n]}$$

Derivation. Let $\sigma_t = [v_{1\ldots n} \leftarrow t_{1\ldots n}]$ and $\sigma_s = [v_{1\ldots n} \leftarrow s_{1\ldots n}]$, so that our goal is to derive $x/\sigma_t = x/\sigma_s$. The derivation proceeds over the recursive structure of the term x.

If x is a constant, then $x/\sigma_t = x$ and $x/\sigma_s = x$, so our goal is to show $x = x$, which can be done using the reflexivity rule.

If x is a variable, then there are two cases. If $x = v_i$ for some i, then $x/\sigma_t = t_i$ and $x/\sigma_s = s_i$, so our goal is to prove $t_i = s_i$. But this is one of our premises, so we may simply use that proof. Otherwise, if $x \neq v_i$ for any i, then $x/\sigma_t = x$ and $x/\sigma_s = x$, so our goal is to show $x = x$, which we can do by reflexivity.

If x is a function application, $\texttt{(}f\ \ a_1\ \ \ldots\ \ a_m\texttt{)}$, then we may recursively construct a proof of $a_i/\sigma_t = a_i/\sigma_s$ for each i. In this case,

$$x/\sigma_t = \texttt{(}f\ \ a_1/\sigma_t\ \ \ldots\ \ a_m/\sigma_t\texttt{)}, \text{ and}$$
$$x/\sigma_s = \texttt{(}f\ \ a_2/\sigma_s\ \ \ldots\ \ a_m/\sigma_s\texttt{)},$$

so our goal is to derive

$$\texttt{(}f\ \ a_1/\sigma_t\ \ \ldots\ \ a_m/\sigma_t\texttt{)} = \texttt{(}f\ \ a_2/\sigma_s\ \ \ldots\ \ a_m/\sigma_s\texttt{)},$$

which we can do via = by arguments and our recursively constructed proofs.

Finally, x may be a lambda abbreviation, $\texttt{((lambda (}w_{1\ldots m}\texttt{)}\ \beta\texttt{)}\ a_{1\ldots m}\texttt{)}$. If we let $c_i = a_i/\sigma_t$ and $d_i = a_i/\sigma_s$, then our goal, $x/\sigma_t = x/\sigma_s$, is the same as

$$\texttt{((lambda (}w_{1\ldots m}\texttt{)}\ \beta\texttt{)}\ c_{1\ldots m}\texttt{)} = \texttt{((lambda (}w_{1\ldots m}\texttt{)}\ \beta\texttt{)}\ d_{1\ldots m}\texttt{)}.$$

Now, as in the function application case, we may recursively construct $a_i/\sigma_t = a_i/\sigma_s$, i.e., proofs of $c_i = d_i$.

Next, let σ_c and σ_d be the following substitution lists,

$$\sigma_c = [w_1 \leftarrow c_1, \ldots, w_m \leftarrow c_m], \text{ and}$$
$$\sigma_d = [w_1 \leftarrow d_1, \ldots, w_m \leftarrow d_m].$$

Since we can prove that each $c_i = d_i$, σ_c and σ_d satisfy the criteria of the dual substitution rule, and since β is smaller than x, it is well-founded to construct a proof of $\beta/\sigma_c = \beta/\sigma_d$ recursively. Rephrasing this conclusion using the definitions of σ_c and σ_d, we find that we have derived

$$\beta/[w_{1\ldots m} \leftarrow a_{1\ldots m}/\sigma_t] = \beta/[w_{i\ldots m} \leftarrow a_{1\ldots m}/\sigma_s],$$

which, by Theorem 6.1, is the same as

$$(\beta[w_{1\ldots m} \leftarrow a_{1\ldots m}])/\sigma_t = (\beta/[w_{1\ldots m} \leftarrow a_{1\ldots m}])/\sigma_s. \quad (*1)$$

Using this result, we finish out the derivation as follows:

$x = \beta/[w_{1\ldots m} \leftarrow a_{1\ldots m}]$	β-reduction	(*2)
$x/\sigma_t = (\beta/[w_{1\ldots m} \leftarrow a_{1\ldots m}])/\sigma_t$	Instantiation *2	
$(\beta[w_{1\ldots m} \leftarrow a_{1\ldots m}])/\sigma_t = (\beta/[w_{1\ldots m} \leftarrow a_{1\ldots m}])/\sigma_s$	*1	
$x/\sigma_s = (\beta/[w_{1\ldots m} \leftarrow a_{1\ldots m}])/\sigma_s$	Instantiation *2	
$x/\sigma_t = x/\sigma_s$	Dual substitution lemma 1	□

With the dual substitution rule in place, argument substitution into lambdas can be done by beta-reducing the lambda with each list of actuals, and then using the dual substitution rule to equate the results.

Derived Rule 89. Lambda = by arguments

$$\frac{\begin{array}{l} t_1 = s_1 \\ \vdots \\ t_n = s_n \end{array}}{\texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ t_{1\ldots n}\texttt{)} = \texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ s_{1\ldots n}\texttt{)}}$$

Derivation.

$\texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ t_{1\ldots n}\texttt{)} = \beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}]$	β-reduction	
$\beta[x_{1\ldots n} \leftarrow t_{1\ldots n}] = \beta[x_{1\ldots n} \leftarrow s_{1\ldots n}]$	Dual subst.	
$\texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ t_{1\ldots n}\texttt{)} = \beta[x_{1\ldots n} \leftarrow s_{1\ldots n}]$	Trans. =	(*1)
$\texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ s_{1\ldots n}\texttt{)} = \beta/[x_{1\ldots n} \leftarrow s_{1\ldots n}]$	β-reduction	
$\beta/[x_{1\ldots n} \leftarrow s_{1\ldots n}] = \texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ s_{1\ldots n}\texttt{)}$	Commute =	
$\texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ t_{1\ldots n}\texttt{)} = \texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ s_{1\ldots n}\texttt{)}$	Trans. = *1	□

It is straightforward, but tedious, to adapt the above arguments to derive the disjoined version of these rules. Accordingly, we only mention what these rules are, without explaining the details of their derivation.

Derived Rule 90. Disjoined = by arguments

$$\begin{array}{l} P \vee t_1 = s_1 \\ \vdots \\ P \vee t_n = s_n \\ \hline P \vee \texttt{(}f\ \ t_1\ \ \ldots\ \ t_n\texttt{)} = \texttt{(}f\ \ s_1\ \ \ldots\ \ s_n\texttt{)} \end{array}$$

Derived Rule 91. Disjoined dual substitution

$$\begin{array}{l} P \vee t_1 = s_1 \\ \vdots \\ P \vee t_n = s_n \\ \hline P \vee x/[v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n] = x/[v_1 \leftarrow s_1, \ldots, v_n \leftarrow s_n] \end{array}$$

Derived Rule 92. Disjoined lambda = by arguments

$$\begin{array}{l} P \vee t_1 = s_1 \\ \vdots \\ P \vee t_n = s_n \\ \hline P \vee \texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ t_{1\ldots n}\texttt{)} = \texttt{((lambda (}x_{1\ldots n}\texttt{)}\ \beta\texttt{)}\ s_{1\ldots n}\texttt{)} \end{array}$$

With the argument-replacement rules in place, we can develop a general-purpose subterm replacement rule. Given terms *old* and *new*, REPL(x, *old*, *new*) produces a new term where all occurrences of *old*, not counting lambda bodies, have been replaced by *new*. In particular,

– If x is *old*, then REPL(x, *old*, *new*) = *new*.

– Otherwise, if x is a constant or a variable, then there are no occurrences of *old* inside of x, so REPL(x, *old*, *new*) = x.

– Otherwise, if x is $\texttt{(}f\ \ t_1\ \ \ldots\ \ t_n\texttt{)}$, then the arguments may contain occurrences of *old*, so REPL(x, *old*, *new*) = $\texttt{(}f\ \ t_1{}'\ \ \ldots\ \ t_n{}'\texttt{)}$, where each $t_i{}'$ is recursively REPL(t_i, *old*, *new*).

– Otherwise, x is `((lambda (`$x_1 \; \ldots \; x_n$`) `β`)` $t_1 \; \ldots \; t_n$`)`. Now,

$$\textsc{repl}(x, \mathit{old}, \mathit{new}) = \texttt{((lambda (}x_1 \; \ldots \; x_n\texttt{) }\beta\texttt{)}\; {t_1}' \; \ldots \; {t_n}'\texttt{)},$$

where each ${t_i}'$ is recursively $\textsc{repl}(t_i, \mathit{old}, \mathit{new})$.

Derived Rule 93. Replace subterm

$$\frac{\mathit{old} = \mathit{new}}{x = \textsc{repl}(x, \mathit{old}, \mathit{new})}$$

Derivation. The derivation follows the recursive structure of REPL.

As a basis, if x is *old* then our goal is $\mathit{old} = \mathit{new}$, and we have been given a proof of this. Otherwise, if x is a constant or a variable, our goal is $x = x$, which is trivial by reflexivity.

Otherwise, if x is `(`$f \; t_1 \; \ldots \; t_n$`)`, let ${t_i}' = \textsc{repl}(t_i, \mathit{old}, \mathit{new})$ for each i. We may recursively derive $t_i = {t_i}'$ for each i. From these proofs, the $=$ by arguments rule allows us to obtain our goal, `(`$f \; t_1 \; \ldots \; t_n$`)` $=$ `(`$f \; {t_1}' \; \ldots \; {t_n}'$`)`.

Finally, if x is `((lambda (`$x_1 \; \ldots \; x_n$`) `β`)` $t_1 \; \ldots \; t_n$`)` the case is similar. For each i, let ${t_i}' = \textsc{repl}(t_i, \mathit{old}, \mathit{new})$ and recursively derive $t_i = {t_i}'$. From these proofs, the lambda $=$ by arguments rule allows us to obtain our goal,

$$\texttt{((lambda (}x_1 \; \ldots \; x_n\texttt{) }\beta\texttt{)}\; t_1 \; \ldots \; t_n\texttt{)} = \texttt{((lambda (}x_1 \; \ldots \; x_n\texttt{) }\beta\texttt{)}\; {t_1}' \; \ldots \; {t_n}'\texttt{)}.$$

□

It is straightforward to adapt this argument to derive the analogous disjoined rule, so we omit the details and only mention the rule, itself.

Derived Rule 94. Disjoined replace subterm

$$\frac{P \vee \mathit{old} = \mathit{new}}{P \vee x = \textsc{repl}(x, \mathit{old}, \mathit{new})}$$

6.4 Evaluation

In the ACL2 theorem prover, and in our system, Skolem functions are usually avoided, and most concepts are introduced as terminating, recursive functions. A useful consequence of this approach is that most *ground terms* (terms with no free variables) may be canonicalized to constants by simple evaluation. We now introduce an evaluator which is similar to McCarthy's [63] evaluator for Lisp.

We say a *definition* is a formula of the form $(f\ \ x_1\ \ \ldots\ \ x_n) = \beta$ where the x_i are distinct variables and $\textsc{freevars}(\beta) \subseteq \{x_1, \ldots, x_n\}$. This is a purely syntactic criterion which is far more relaxed than the admission obligations for a recursive function definition. For instance, we do not require that β is well-formed with respect to an arity table, that f terminates, etc.

Our evaluator function, EV, takes three inputs: x, a ground term to evaluate; *defs*, a list of definitions in the simple, syntactic sense above; and *depth*, a counter which acts like a stack depth to ensure EV terminates. There are many reasons EV might fail. For example, perhaps x is `(f 1 2)` but `f` is 3-ary, or perhaps *depth* is simply too small to finish this computation. To signal failure, EV may return the unquoted symbol NIL, whereas successful evaluations result in quoted constants.

We implement EV as a flag function which has two modes of operation; one for evaluating a term, and another for evaluating a list of terms. The basic operation of EV is as follows. Except where noted, the *depth* argument is not changed in recursive calls.

– To ensure termination, if *depth* is zero, EV fails.

– If x is a constant, then EV simply returns x.

– If x is a variable, then it is not a ground term and EV fails.

– As a special case, `(if` a b c`)` is handled lazily, which allows EV to evaluate recursive functions. First, EV attempts to evaluate a; any failure is propagated, otherwise the constant a' is produced. Then, if a' is not the constant `nil`, EV returns the result of recursively evaluating b, and otherwise it returns the result of recursively evaluating c.

– For other function applications, `(`f t_1 $\ldots$ t_n`)`, EV first eagerly evaluates each t_i; any failure is propagated, otherwise a list of constants, $t_1', \ldots, t_n'$, is produced. Now, there are two cases.

  Suppose f is one of the primitive functions. If n has improper arity, EV fails. Otherwise, `(`f t_1' $\ldots$ t_n'`)` is a base-evaluable term, and EV uses `logic.base-evaluator` to evaluate it to a constant.

  Otherwise, EV consults the list of definitions to determine if f is a defined function of the proper arity. If not, it fails. Otherwise, let the definition be `(`f x_1 $\ldots$ x_n`)` $= \beta$. Now, since FREEVARS$(\beta) \subseteq \{x_1, \ldots, x_n\}$, and each t_i' is a constant, $\beta/[x_1 \leftarrow t_1', \ldots, x_n \leftarrow t_n']$ is a ground term, and EV returns the result of attempting to evaluate it after decreasing the *depth* by one.

– For lambda abbreviations, `((lambda (`x_1 $\ldots$ x_n`)` β`)` t_1 $\ldots$ t_n`)`, EV first eagerly attempts to evaluate each t_i, propagating any failure. Otherwise, each t_i has been successfully evaluated to the constant t_i', and EV returns the result of recursively evaluating the new ground term, $\beta/[x_1 \leftarrow t_i', \ldots, x_n \leftarrow t_n']$, again decreasing the *depth* by one to ensure termination.

The termination of EV can be established using a two-part measure where we first consider the *depth* and then consider the rank of x.

We now turn our attention to the justification of EV.

Derived Rule 95. If when not nil

$$\frac{a \neq \texttt{nil}}{\texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b}$$

Derivation. (7)

$\texttt{x} = \texttt{nil} \lor \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil
$a = \texttt{nil} \lor \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b$	Instantiation
$a \neq \texttt{nil}$	Given
$\texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b$	Modus ponens 2 $\square$

Derived Rule 96. If when nil

$$\frac{a = \texttt{nil}}{\texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = c}$$

Derivation. (7)

$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil
$a \neq \texttt{nil} \lor \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = c$	Instantiation
$a = \texttt{nil}$	Given
$\texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = c$	Modus ponens $\square$

Derived Rule 97. Evaluation

$$\frac{}{x = x'}, \text{ where } \textsc{ev}(x, \mathit{defs}, \mathit{depth}) = x', \text{ and all } \mathit{defs} \text{ are axioms.}$$

Derivation.

The derivation follows the recursive structure of EV. As a basis, if *depth* is zero, then EV has failed so there is nothing to show; if x is a constant, then EV returns x so our goal is to prove $x = x$, which we may do with the reflexivity rule; finally, if x is a variable, then EV fails so there is nothing to show.

Supposing x is $\texttt{(if}\ \ a\ \ b\ \ c\texttt{)}$, we may recursively derive $a = a'$. The first case is that a' is non-$\texttt{nil}$, then we may also recursively derive $b = b'$. Our goal is $(\texttt{if}\ \ a\ \ b\ \ c) = b'$.

$a' \neq \texttt{nil}$	$\neq$ constants	
$a = a'$	Recursive construction	
$a \neq \texttt{nil}$	Substitute into $\neq$	(*1)
$(\texttt{if}\ \ a\ \ b\ \ c) = b$	If when not nil	
$b = b'$	Recursive construction	
$(\texttt{if}\ \ a\ \ b\ \ c) = b'$	Transitivity of $=$ *1	

The second case is that a' is $\texttt{nil}$. Here, we may still recursively derive $a = a'$ (i.e., $a = \texttt{nil}$), and we may also recursively derive $c = c'$. Our goal is $(\texttt{if}\ \ a\ \ b\ \ c) = c'$.

$a = \texttt{nil}$	Recursive construction
$(\texttt{if}\ \ a\ \ b\ \ c) = c$	If when nil
$c = c'$	Recursive construction
$(\texttt{if}\ \ a\ \ b\ \ c) = c'$	Transitivity of $=$

Suppose x is some other function application, $(f\ \ t_1\ \ \ldots\ \ t_n)$. Now we may recursively derive $t_i = t_i{}'$ for each i. The first case is that f is a primitive function. Let c be the result of running $\texttt{logic.base-evaluator}$ on $(f\ \ t_1{}'\ \ \ldots\ \ t_n{}')$. Now, our goal is to derive $(f\ \ t_1\ \ \ldots\ \ t_n) = c$.

$(f\ \ t_1\ \ \ldots\ \ t_n) = (f\ \ t_1{}'\ \ \ldots\ \ t_n{}')$	$=$ by arguments
$((f\ \ t_1{}'\ \ \ldots\ \ t_n{}') = c$	Base evaluation
$(f\ \ t_1\ \ \ldots\ \ t_n) = c$	Transitivity of $=$

The second case is that f is a defined function. Let $(f\ \ x_1\ \ \ldots\ \ x_n) = \beta$ be the definition of f, and note that we have assumed this formula is an axiom. Suppose that c is the result of recursively evaluating $\beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}']$. We may recursively derive $\beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}'] = c$, and our goal is to show $(f\ \ t_1\ \ \ldots\ \ t_n) = c$.

$(f\ \ x_1\ \ \ldots\ \ x_n) = \beta$	Axiom
$(f\ \ t_1{}'\ \ \ldots\ \ t_n{}') = \beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}']$	Instantiation
$\beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}'] = c$	Recursive construction
$(f\ \ t_1{}'\ \ \ldots\ \ t_n{}') = c$	Transitivity of $=$
$(f\ \ t_1\ \ \ldots\ \ t_n) = (f\ \ t_1{}'\ \ \ldots\ \ t_n{}')$	$=$ by arguments
$(f\ \ t_1\ \ \ldots\ \ t_n) = c$	Transitivity of $=$

Finally, suppose x is `((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}$`)`. We may recursively derive $t_i = t_i{}'$ for each i. Let c be the result of recursively evaluating $\beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}']$, so we may also recursively derive $\beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}'] = c$. Our goal is to show `((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}$`)` $= c$, and this may be done as follows:

`((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}$`)` $=$ `((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}{}'$`)`	Lambda = by args.
`((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}{}'$`)` $= \beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}']$	Beta reduction
`((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}$`)` $= \beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}']$	Transitivity of =
$\beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}{}'] = c$	Recursive constr.
`((lambda` $(x_{1\ldots n})$ β`)` $t_{1\ldots n}$`)` $= c$	Transitivity of = $\square$

Part III

Theorem Proving

Chapter 7

Clauses

To implement an effective proof search, we need to be able to work "backward" from a goal instead of "forward" from our axioms. Our basic strategy for backward proof search is as follows. First, we first convert the goal formula into conjunctive normal form clauses, which are more convenient to work with than formulas because of their regular structure. We then try to simplify these clauses, mainly through lemma-driven rewriting, but also through other techniques. Ideally, each simplification will leave us with reduced goals that are simpler to prove, or which are so simple that we can prove them outright. To justify these simplifications, we need to be able to "reverse" each reduction—that is, given proofs of the reduced clauses, we must be able to derive proofs of the original goal clauses.

A *clause* in conjunctive normal form is a disjunction of one or more *literals*. We represent literals as terms and use the words "term" and "literal" interchangeably. We represent clauses as non-empty lists of terms. Given a literal, t, we say the *term formula* for t is $t \neq \texttt{nil}$, and given a clause, $C = [t_1, \ldots, t_n]$, the *clause formula* for C is $t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$. When we speak of proving a clause, we really mean proving the corresponding clause formula.

In this chapter, we explain how formulas may be converted to clauses and introduce some basic ways to simplify clauses. For instance, we explain how to replace the literals of a clause with equivalent literals. We also provide some routines to clean up clauses and to split clauses into simpler subgoals.

7.1 Conversion to Clauses

Any formula can be converted into an equivalent clause. Our conversion process begins with COMP, an algorithm which, given a formula F as input, produces a term, $\text{COMP}(F)$, that is equivalent in the following sense: given a proof of F we may derive $\text{COMP}(F) \neq \texttt{nil}$, and vice versa. We think of this algorithm as "compiling" a formula into a term.

$$\begin{aligned}\text{COMP}(a = b) &\triangleq \texttt{(equal}\ \ a\ \ b\texttt{)}\\ \text{COMP}(\neg A) &\triangleq \texttt{(if}\ \ \text{COMP}(A)\ \ \texttt{nil}\ \ \texttt{t)}\\ \text{COMP}(A \vee B) &\triangleq \texttt{(if}\ \ \text{COMP}(A)\ \ \texttt{t}\ \ \text{COMP}(B)\texttt{)}.\end{aligned}$$

To establish the equivalence of F and $\text{COMP}(F) \neq \texttt{nil}$, we first introduce some supporting derivations to allow us to more easily work with terms involving `if`.

Derived Rule 98. Disjoined if when not nil

$$\frac{P \vee a \neq \texttt{nil}}{P \vee \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b}$$

Derivation. (17)

$\texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil
$a = \texttt{nil} \vee \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b$	Instantiation
$P \vee a = \texttt{nil} \vee \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b$	Expansion
$P \vee a \neq \texttt{nil}$	Given
$P \vee \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = b$	Dj. mp2 ☐

Derived Rule 99. Disjoined if when nil

$$\frac{P \vee a = \texttt{nil}}{P \vee \texttt{(if}\ \ a\ \ b\ \ c\texttt{)} = c}$$

Derivation. (17)

$\texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil

$a \neq \texttt{nil} \lor (\texttt{if}\ a\ b\ c) = c$	Instantiation	
$P \lor a \neq \texttt{nil} \lor (\texttt{if}\ a\ b\ c) = c$	Expansion	
$P \lor a = \texttt{nil}$	Given	
$P \lor (\texttt{if}\ a\ b\ c) = c$	Dj. modus ponens	$\square$

Formal Theorem 9. If redux same

$$(\texttt{if x y y}) = \texttt{y}$$

Proof.

$\texttt{x} = \texttt{nil} \lor (\texttt{if x y z}) = \texttt{y}$	Ax. if when nnil	
$\texttt{x} = \texttt{nil} \lor (\texttt{if x y y}) = \texttt{y}$	Instantiation	(*1)
$\texttt{x} \neq \texttt{nil} \lor (\texttt{if x y z}) = \texttt{z}$	Axiom if when nil	
$\texttt{x} \neq \texttt{nil} \lor (\texttt{if x y y}) = \texttt{y}$	Instantiation	
$(\texttt{if x y y}) = \texttt{y} \lor (\texttt{if x y y}) = \texttt{y}$	Cut *1	
$(\texttt{if x y y}) = \texttt{y}$	Contraction	$\square$

Formal Theorem 10. If when same

$$\texttt{y} \neq \texttt{z} \lor (\texttt{if x y z}) = \texttt{y}$$

Proof.

$\texttt{x} = \texttt{x}$	Reflexivity	
$\texttt{y} \neq \texttt{z} \lor \texttt{x} = \texttt{x}$	Expansion	(*1a)
$\texttt{y} = \texttt{y}$	Reflexivity	
$\texttt{y} \neq \texttt{z} \lor \texttt{y} = \texttt{y}$	Expansion	(*1b)
$\texttt{y} \neq \texttt{z} \lor \texttt{y} = \texttt{z}$	Prop. schema	
$\texttt{y} \neq \texttt{z} \lor \texttt{z} = \texttt{y}$	Dj. commute $=$	(*1c)
$\texttt{y} \neq \texttt{z} \lor (\texttt{if x y z}) = (\texttt{if x y y})$	Dj. $=$ args *1abc	(*1)
$(\texttt{if x y y}) = \texttt{y}$	Th. if redux same	
$\texttt{y} \neq \texttt{z} \lor (\texttt{if x y y}) = \texttt{y}$	Expansion	
$\texttt{y} \neq \texttt{z} \lor (\texttt{if x y z}) = \texttt{y}$	Dj. trans. $=$ *1	$\square$

Derived Rule 100. If when same

$$\frac{b = c}{(\texttt{if}\ \ a\ \ b\ \ c) = b}$$

Derivation. (7)

$\texttt{y} \neq \texttt{z} \lor (\texttt{if}\ \ \texttt{x}\ \ \texttt{y}\ \ \texttt{z}) = \texttt{y}$	Th. if when same
$b \neq c \lor (\texttt{if}\ \ a\ \ b\ \ c) = b$	Instantiation
$b = c$	Given
$(\texttt{if}\ \ a\ \ b\ \ c) = b$	Modus ponens □

Derived Rule 101. Disjoined if when same

$$\frac{P \lor b = c}{P \lor (\texttt{if}\ \ a\ \ b\ \ c) = b}$$

Derivation. (17)

$\texttt{y} \neq \texttt{z} \lor (\texttt{if}\ \ \texttt{x}\ \ \texttt{y}\ \ \texttt{z}) = \texttt{y}$	Th. if when same
$b \neq c \lor (\texttt{if}\ \ a\ \ b\ \ c) = b$	Instantiation
$P \lor b \neq c \lor (\texttt{if}\ \ a\ \ b\ \ c) = b$	Expansion
$P \lor b = c$	Given
$P \lor (\texttt{if}\ \ a\ \ b\ \ c) = b$	Dj. modus ponens □

Derived Rule 102. Compile formula lemma 1

$$\frac{\begin{array}{l} B \lor p = \texttt{nil} \\ C \lor q = \texttt{nil} \end{array}}{(B \lor C) \lor (\texttt{if}\ \ p\ \ \texttt{t}\ \ q) = \texttt{nil}}$$

Derivation. (17)

$B \lor p = \texttt{nil}$	Given
$B \lor (\texttt{if}\ \ p\ \ \texttt{t}\ \ q) = q$	Dj. if when nil
$(B \lor C) \lor (\texttt{if}\ \ p\ \ \texttt{t}\ \ q) = q$	Multi assoc exp. (*1)
$C \lor q = \texttt{nil}$	Given
$(B \lor C) \lor q = \texttt{nil}$	Multi assoc exp.
$(B \lor C) \lor (\texttt{if}\ \ p\ \ \texttt{t}\ \ q) = \texttt{nil}$	Dj. trans. = *1 □

Derived Rule 103. Compile formula lemma 2

$$\frac{\begin{array}{l}\neg B \vee p = \texttt{t} \\ \neg C \vee q = \texttt{t}\end{array}}{\neg(B \vee C) \vee \texttt{(if } p \texttt{ t } q\texttt{)} = \texttt{t}}$$

Derivation. (90)

$\neg B \vee p = \texttt{t}$	Given	
$\neg B \vee p \neq \texttt{nil}$	Dj. not nil from t	
$\neg B \vee \texttt{(if } p \texttt{ t } q\texttt{)} = \texttt{t}$	Dj. if when nnil	(*1)
$\neg C \vee q = \texttt{t}$	Given	
$\neg C \vee \texttt{t} = q$	Dj. commute =	
$\neg C \vee \texttt{(if } p \texttt{ t } q\texttt{)} = \texttt{t}$	Dj. if when same	
$\neg(B \vee C) \vee \texttt{(if } p \texttt{ t } q\texttt{)} = \texttt{t}$	Merge imp. *1	$\square$

We are now ready for the main result. As one might expect, the derivation follows the recursive structure of COMP, but a slight twist is that we actually derive two formulas at once. To indicate this, we put both formulas "below the line" in the rule's description.

Derived Rule 104. Compile Formula

$$\frac{}{\begin{array}{l}\neg F \vee \textsc{comp}(F) = \texttt{t} \\ F \vee \textsc{comp}(F) = \texttt{nil}\end{array}}$$

Derivation.

As a basis, suppose F is $a = b$. Now $\textsc{comp}(F)$ is $\texttt{(equal } a \ b\texttt{)}$, so our goals and their derivations are as follows.

1. $a \neq b \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}$.

$\texttt{x} \neq \texttt{y} \vee \texttt{(equal x y)} = \texttt{t}$	Axiom equal when same
$a \neq b \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}$	Instantiation

2. $a = b \vee$ `(equal` a b`)` $=$ `nil`.

`x` $=$ `y` $\vee$ `(equal x y)` $=$ `nil`	Axiom equal when diff
$a = b \vee$ `(equal` a b`)` $=$ `nil`	Instantiation

Otherwise, suppose F is $\neg A$. Now $\textsc{comp}(F)$ is `(if` $\textsc{comp}(A)$ `nil t)`, and we may recursively derive $\neg A \vee \textsc{comp}(A) =$ `t` and $A \vee \textsc{comp}(A) =$ `nil`. Then,

1. $\neg\neg A \vee$ `(if` $\textsc{comp}(A)$ `nil t)` $=$ `t`.

$A \vee \textsc{comp}(A) =$ `nil`	Recursive construction
$A \vee$ `(if` $\textsc{comp}(A)$ `nil t)` $=$ `t`	Disjoined if when nil
$\neg\neg A \vee$ `(if` $\textsc{comp}(A)$ `nil t)` $=$ `t`	Lhs insert $\neg\neg$

2. $\neg A \vee$ `(if` $\textsc{comp}(A)$ `nil t)` $=$ `nil`.

$\neg A \vee \textsc{comp}(A) =$ `t`	Recursive construction
$\neg A \vee \textsc{comp}(A) \neq$ `nil`	Disjoined not nil from t
$\neg A \vee$ `(if` $\textsc{comp}(A)$ `nil t)` $=$ `nil`	Disjoined if when not nil

Finally, suppose F is $A \vee B$. Now, $\textsc{comp}(F)$ is `(if` $\textsc{comp}(A)$ `t` $\textsc{comp}(B)$`)`, and we may recursively derive $\neg A \vee \textsc{comp}(A) =$ `t`, $\neg B \vee \textsc{comp}(B) =$ `t`, $A \vee \textsc{comp}(A) =$ `nil`, and $B \vee \textsc{comp}(B) =$ `nil`. Now,

1. $\neg(A \vee B) \vee$ `(if` $\textsc{comp}(A)$ `t` $\textsc{comp}(B)$`)` $=$ `t`.

$\neg A \vee \textsc{comp}(A) =$ `t`	Recursive construction
$\neg B \vee \textsc{comp}(B) =$ `t`	Recursive construction
$\neg(A \vee B) \vee$ `(if` $\textsc{comp}(A)$ `t` $\textsc{comp}(B)$`)` $=$ `t`	Compile formula lemma 2

2. $(A \vee B) \vee$ `(if` $\textsc{comp}(A)$ `t` $\textsc{comp}(B)$`)` $=$ `nil`.

$A \vee \textsc{comp}(A) =$ `nil`	Recursive construction
$B \vee \textsc{comp}(B) =$ `nil`	Recursive construction
$(A \vee B) \vee$ `(if` $\textsc{comp}(A)$ `t` $\textsc{comp}(B)$`)` $=$ `nil`	Compile formula lemma 1 $\square$

Hence, given any formula F, the singleton clause whose only literal is $\textsc{comp}(F)$ is equivalent to F. That is, given a proof of the clause formula, $\textsc{comp}(F) \neq \texttt{nil}$, we may use the compile formula rule to derive F, and vice versa.

7.2 Updating Clauses

When we are trying to prove some goal clause, say $C = [t_1, \ldots, t_n]$, we will often simplify each literal to produce an equivalent clause, $C' = [t_1{}', \ldots, t_n{}']$. If, with some further work, we manage to construct a proof of C', we will still need a way to prove the original goal, C. In this section, we develop a rule which allows us to prove C when given (1) a proof of C', and (2) a proof of $t_i = t_i{}'$ for each i.

Derived Rule 105. Aux update clause lemma1

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \\ a = b \end{array}}{a \neq \texttt{nil} \vee P}$$

Derivation. (34)

$a = b$	Given
$P \vee a = b$	Expansion
$P \vee b \neq \texttt{nil}$	Given
$P \vee a \neq \texttt{nil}$	Dj. sub. into $\neq$
$a \neq \texttt{nil} \vee P$	Commute or $\square$

Derived Rule 106. Aux update clause lemma2

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \vee Q \\ a = b \end{array}}{(a \neq \texttt{nil} \vee P) \vee Q}$$

Derivation. (59)

$P \vee b \neq \texttt{nil} \vee Q$	Given
$(P \vee b \neq \texttt{nil}) \vee Q$	Associativity
$Q \vee P \vee b \neq \texttt{nil}$	Commute or

$(Q \lor P) \lor b \neq \texttt{nil}$	Associativity
$a = b$	Given
$a \neq \texttt{nil} \lor Q \lor P$	Aux update clause lm.1
$a \neq \texttt{nil} \lor P \lor Q$	Dj. commute or
$(a \neq \texttt{nil} \lor P) \lor Q$	Associativity ☐

To keep proof sizes down, we implement our clause-updating rule in a tail-recursive style which is somewhat similar to the revappend disjunction rule. At each step, we think of the D_i as "done", and the $s_i \neq \texttt{nil}$ as "to do". Notice that the inductive cases involve mainly a single application of the above lemmas, so the number of proof steps required grows only linearly in the size of the clause.

Derived Rule 107. Aux update clause

$$\frac{\begin{array}{l} D_{1 \ldots m} \lor s_1 \neq \texttt{nil} \lor \cdots \lor s_n \neq \texttt{nil} \\ t_1 = s_1 \\ \vdots \\ t_n = s_n \end{array}}{t_n \neq \texttt{nil} \lor \cdots \lor t_1 \neq \texttt{nil} \lor D_{1 \ldots m}}$$

Derivation. $\mathcal{O}(n)$

As a basis, if n is 0 then we are already given a proof of our goal.

Otherwise, if n is 1 and m is 0,

$s_1 \neq \texttt{nil}$	Given
$t_1 = s_1$	Given
$t_1 \neq \texttt{nil}$	Substitute into $\neq$,

Otherwise, if n is 1 and $m > 0$,

$D_{1 \ldots m} \lor s_1 \neq \texttt{nil}$	Given
$t_1 = s_1$	Given
$t_1 \neq \texttt{nil} \lor D_{1 \ldots m}$	Aux update clause lemma1

Otherwise, if $n > 1$ and m is 0. Let S_i be $s_i \neq \texttt{nil}$. Now, for any A, if we can establish $A \vee S_{2 \ldots n}$, then we may recursively derive $t_n \neq \texttt{nil} \vee \cdots \vee t_2 \neq \texttt{nil} \vee A$ using the given proofs of $t_2 = s_2, \ldots, t_n = s_n$; this is well-founded since n is decreasing. Then,

$s_1 \neq \texttt{nil} \vee S_{2 \ldots n}$	Given
$S_{2 \ldots n} \vee s_1 \neq \texttt{nil}$	Commute or
$t_1 \neq s_1$	Given
$t_1 \neq \texttt{nil} \vee S_{2 \ldots n}$	Aux update clause lemma1
$t_n \neq \texttt{nil} \vee \cdots \vee t_1 \neq \texttt{nil}$	Recursively, $A \leftarrow t_1 \neq \texttt{nil}$

Finally, if $n > 1$ and $m > 1$, then as before let S_i be $s_i \neq \texttt{nil}$ and again note that for any A, if we can establish $A \vee S_{2 \ldots n}$, then we may recursively derive $t_n \neq \texttt{nil} \vee \cdots \vee t_2 \neq \texttt{nil} \vee A$. Now,

$(D_{1 \ldots m}) \vee s_1 \neq \texttt{nil} \vee S_{2 \ldots n}$	Given
$t_1 = s_1$	Given
$(t_1 \neq \texttt{nil} \vee D_{1 \ldots m}) \vee S_{2 \ldots n}$	Aux update clause lemma2
$t_n \neq \texttt{nil} \vee \cdots \vee t_1 \neq \texttt{nil} \vee D_{1 \ldots m}$	Recursively, $A \leftarrow t_1 \neq \texttt{nil} \vee D_{1 \ldots m}$ $\square$

Derived Rule 108. Update clause

$$\frac{\begin{array}{l} {t_1}' \neq \texttt{nil} \vee \cdots \vee {t_n}' \neq \texttt{nil} \\ t_1 = {t_1}' \\ \vdots \\ t_n = {t_n}' \end{array}}{t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}}$$

Derivation. $\mathcal{O}(n)$

When we implement this derivation as a Lisp function, as a special optimization we first check whether any ${t_i}'$ differs from t_i. If none of the terms has changed, we can simply reuse our first premise. Otherwise, we begin by deriving $t_n \neq \texttt{nil} \vee \cdots \vee t_1 \neq \texttt{nil}$ from these premises using the aux update clause rule. Then, via rev disjunction, we obtain our goal. $\square$

It is straightforward to adapt the above derivations to obtain a disjoined version of the update clause rule. We omit the details, and only summarize the rule, below.

Derived Rule 109. Disjoined update clause

$$\frac{\begin{array}{l} P \vee {t_1}' \neq \texttt{nil} \vee \cdots \vee {t_n}' \neq \texttt{nil} \\ P \vee t_1 = {t_1}' \\ \vdots \\ P \vee t_n = {t_n}' \end{array}}{P \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}}$$

7.3 Equivalent Literals

The update clause rule is somewhat weak in that it requires us to show each replacement literal, ${t_i}'$, is equal to the original literal, t_i. But it would suffice to show that t_i and ${t_i}'$ are equivalent in the sense of generalized Booleans—that is, either they are both `nil` or are both non-`nil`. We can identify this situation with the function `iff`, defined as follows.

Definition: `iff`

```
(pequal* (iff x y)
         (if x
             (if y t nil)
           (if y nil t)))
```

The function `iff` is an equivalence relation which has many nice properties, and it is useful to introduce a number of formal theorems and derived rules about it. All of this is quite routine, so we only provide summaries here and put the details in Appendix A.

Derived Rule 110. If of t

$$\frac{}{\texttt{(if t } b \ c\texttt{)} = b}$$

Derived Rule 111. If of nil

$$\frac{}{\texttt{(if nil } b\ c\texttt{)} = c}$$

Formal Theorem 11. Iff lhs false

$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$

Formal Theorem 12. Iff lhs true

$\texttt{x} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y t nil)}$

Formal Theorem 13. Iff rhs false

$\texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if x nil t)}$

Formal Theorem 14. Iff rhs true

$\texttt{y} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if x t nil)}$

Formal Theorem 15. Iff both true

$\texttt{x} = \texttt{nil} \vee \texttt{y} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$

Formal Theorem 16. Iff both false

$\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$

Formal Theorem 17. Iff true false

$\texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{nil}$

Formal Theorem 18. Iff false true

$\texttt{x} \neq \texttt{nil} \vee \texttt{y} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{nil}$

Formal Theorem 19. Iff t when not nil

$\texttt{x} = \texttt{nil} \vee \texttt{(iff x t)} = \texttt{t}$

Derived Rule 112. Iff t from $\neq$ nil

$$\frac{a \neq \texttt{nil}}{\texttt{(iff } a \texttt{ t)} = \texttt{t}}$$

Derived Rule 113. **Disjoined iff t from $\neq$ nil**

$$\frac{P \vee a \neq \texttt{nil}}{P \vee \texttt{(iff } a \texttt{ t)} = \texttt{t}}$$

Formal Theorem 20. **Iff t when nil**

$$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x t)} = \texttt{nil}$$

Derived Rule 114. **$\neq$ nil from iff t**

$$\frac{\texttt{(iff } a \texttt{ t)} \neq \texttt{nil}}{a \neq \texttt{nil}}$$

Derived Rule 115. **Disjoined $\neq$ nil from iff t**

$$\frac{P \vee \texttt{(iff } a \texttt{ t)} \neq \texttt{nil}}{P \vee a \neq \texttt{nil}}$$

Formal Theorem 21. **Iff nil when nil**

$$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x nil)} = \texttt{t}$$

Formal Theorem 22. **Iff nil when not nil**

$$\texttt{x} = \texttt{nil} \vee \texttt{(iff x nil)} = \texttt{nil}$$

Formal Theorem 23. **Iff nil or t**

$$\texttt{(iff x y)} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$$

Formal Theorem 24. **Reflexivity of iff**

$$\texttt{(iff x x)} = \texttt{t}$$

Formal Theorem 25. **Symmetry of iff**

$$\texttt{(iff x y)} = \texttt{(iff y x)}$$

Derived Rule 116. **Iff t from not nil**

$$\frac{\texttt{(iff } a\ b\texttt{)} \neq \texttt{nil}}{\texttt{(iff } a\ b\texttt{)} = \texttt{t}}$$

Derived Rule 117. **Disjoined iff t from not nil**

$$\frac{P \vee \texttt{(iff}\ \ a\ \ b\texttt{)} \neq \texttt{nil}}{P \vee \texttt{(iff}\ \ a\ \ b\texttt{)} = \texttt{t}}$$

Derived Rule 118. **Iff reflexivity**

$$\frac{}{\texttt{(iff}\ \ a\ \ a\texttt{)} = \texttt{t}}$$

Derived Rule 119. **Commute iff**

$$\frac{\texttt{(iff}\ \ a\ \ b\texttt{)} = \texttt{t}}{\texttt{(iff}\ \ b\ \ a\texttt{)} = \texttt{t}}$$

Derived Rule 120. **Disjoined commute iff**

$$\frac{P \vee \texttt{(iff}\ \ a\ \ b\texttt{)} = \texttt{t}}{P \vee \texttt{(iff}\ \ b\ \ a\texttt{)} = \texttt{t}}$$

Formal Theorem 26. **Iff congruence lemma**

$$\texttt{x} = \texttt{nil} \vee \texttt{y} = \texttt{nil} \vee \texttt{(if x a b)} = \texttt{(if y a b)}$$

Formal Theorem 27. **Iff congruence lemma 2**

$$\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(if x a b)} = \texttt{(if y a b)}$$

Formal Theorem 28. **Iff congruent if 1**

$$\texttt{(iff x y)} = \texttt{nil} \vee \texttt{(if x a b)} = \texttt{(if y a b)}$$

Formal Theorem 29. **Iff congruent iff 2**

$$\texttt{(iff x y)} = \texttt{nil} \vee \texttt{(iff z x)} = \texttt{(iff z y)}$$

Formal Theorem 30. **Iff congruent iff 1**

$$\texttt{(iff x y)} = \texttt{nil} \vee \texttt{(iff x z)} = \texttt{(iff y z)}$$

Formal Theorem 31. **Iff of if x t nil**

$$\texttt{(iff (if x t nil) x)} = \texttt{t}$$

Formal Theorem 32. Transitivity of iff

$$\texttt{(iff x y)} \neq \texttt{t} \vee \texttt{(iff y z)} \neq \texttt{t} \vee \texttt{(iff x z)} = \texttt{t}$$

Derived Rule 121. Transitivity of iff

$$\frac{\texttt{(iff } a \ b\texttt{)} = \texttt{t} \qquad \texttt{(iff } b \ c\texttt{)} = \texttt{t}}{\texttt{(iff } a \ c\texttt{)} = \texttt{t}}$$

Derived Rule 122. Disjoined transitivity of iff

$$\frac{P \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t} \qquad P \vee \texttt{(iff } b \ c\texttt{)} = \texttt{t}}{P \vee \texttt{(iff } a \ c\texttt{)} = \texttt{t}}$$

Formal Theorem 33. Iff from $=$

$$\texttt{x} \neq \texttt{y} \vee \texttt{(iff x y)} = \texttt{t}$$

Derived Rule 123. Iff from $=$

$$\frac{a = b}{\texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derived Rule 124. Disjoined iff from $=$

$$\frac{P \vee a = b}{P \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Formal Theorem 34. Iff from equal

$$\texttt{(equal x y)} \neq \texttt{t} \vee \texttt{(iff x y)} = \texttt{t}$$

Derived Rule 125. Iff from equal

$$\frac{\texttt{(equal } a \ b\texttt{)} = \texttt{t}}{\texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derived Rule 126. Disjoined iff from equal

$$\frac{P \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}}{P \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derived Rule 127. **Negative lit from $\neq$ nil**

$$\frac{a \neq \texttt{nil}}{\texttt{(not }\ a\texttt{)} = \texttt{nil}}$$

Derived Rule 128. **Disjoined negative lit from $=$ nil**

$$\frac{P \vee a = \texttt{nil}}{P \vee \texttt{(not }\ a\texttt{)} \neq \texttt{nil}}$$

Derived Rule 129. **Substitute iff into literal**

$$\frac{\begin{array}{l} b \neq \texttt{nil} \\ \texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t} \end{array}}{a \neq \texttt{nil}}$$

Derived Rule 130. **Disjoined substitute iff into literal**

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \\ P \vee \texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t} \end{array}}{P \vee a \neq \texttt{nil}}$$

We now develop a stronger clause-updating rule which only requires that each t_i is iff-equivalent to $t_i{}'$. The derivation closely follows that of our update clause rule, and we begin with iff-based versions of the lemmas.

Derived Rule 131. **Aux update clause iff lemma1**

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \\ \texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t} \end{array}}{a \neq \texttt{nil} \vee P}$$

Derivation. (87)

$\texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t}$	Given
$P \vee \texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t}$	Expansion
$P \vee b \neq \texttt{nil}$	Given
$P \vee a \neq \texttt{nil}$	Dj. sub. iff into literal
$a \neq \texttt{nil} \vee P$	Commute or □

Derived Rule 132. Aux update clause iff lemma2

$$\frac{\begin{array}{l} P \lor b \neq \texttt{nil} \lor Q \\ (\texttt{iff}\ \ a\ \ b) = \texttt{t} \end{array}}{(a \neq \texttt{nil} \lor P) \lor Q}$$

Derivation. (112)

$P \lor b \neq \texttt{nil} \lor Q$	Given	
$(P \lor b \neq \texttt{nil}) \lor Q$	Associativity	
$Q \lor P \lor b \neq \texttt{nil}$	Commute or	
$(Q \lor P) \lor b \neq \texttt{nil}$	Associativity	(*1)
$a = b$	Given	
$a \neq \texttt{nil} \lor Q \lor P$	Aux update clause iff lm.1	
$a \neq \texttt{nil} \lor P \lor Q$	Dj. commute or	
$(a \neq \texttt{nil} \lor P) \lor Q$	Associativity	□

We can now adapt the aux update clause rule to develop an `iff`-based version. Again the derivation follows a tail-recursive style, with the D_i as "done" and the s_i literals as "to do."

Derived Rule 133. Aux update clause iff

$$\frac{\begin{array}{l} D_{1\ldots m} \lor s_1 \neq \texttt{nil} \lor \cdots \lor s_n \neq \texttt{nil} \\ (\texttt{iff}\ \ t_1\ \ s_1) = \texttt{t} \\ \vdots \\ (\texttt{iff}\ \ t_n\ \ s_n) = \texttt{t} \end{array}}{t_n \neq \texttt{nil} \lor \cdots \lor t_1 \neq \texttt{nil} \lor D_{1\ldots m}}$$

Derivation. $\mathcal{O}(n)$

This is just like the aux update clause rule, except that substitute iff into literal is used instead of substitute into $\neq$, and the lemmas above are used instead of the aux update clause lemmas.

□

Derived Rule 134. Update clause iff

$$\frac{\begin{array}{l} {t_1}' \neq \texttt{nil} \vee \cdots \vee {t_n}' \neq \texttt{nil} \\ \texttt{(iff } t_1 \ {t_1}'\texttt{)} = \texttt{t} \\ \vdots \\ \texttt{(iff } t_n \ {t_n}'\texttt{)} = \texttt{t} \end{array}}{t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}}$$

Derivation. $\mathcal{O}(n)$

As with the update clause rule, when we implement this derivation as a Lisp function, we just reuse the proof of the first premise if no ${t_i}'$ is different than t_i. Otherwise, we derive $t_n \neq \texttt{nil} \vee \cdots \vee t_1 \neq \texttt{nil}$ using the aux update clause iff rule, then reverse this with rev disjunction to obtain our goal. □

7.4 Clause Cleaning

Suppose we are trying to prove a list of clauses, $C_1, \ldots, C_n$. We now develop a cleaning routine which performs some lightweight simplifications on these clauses. This process involves standardizing certain literals into a common format, throwing away redundant and useless literals, and eliminating certain "obvious" clauses. The result of cleaning is a new, simpler list of clauses, say $D_1, \ldots, D_m$, which together are sufficient to prove the original clauses—that is, given proofs of $D_1, \ldots, D_m$, it is possible to construct proofs of $C_1, \ldots, C_n$.

We think of each literal as being either *positive* or *negative*, and the first stage in our cleaning process is to normalize negative terms. We say terms of the form

`(not` *guts*`)`,
`(if` *guts* `nil t)`,
`(equal` *guts* `nil)`,
`(equal nil` *guts*`)`,
`(iff` *guts* `nil)`, or
`(iff nil` *guts*`)`,

are negative, and other literals are positive. For any term, t, we define the *guts* of t, GUTS(t), as follows. If t is negative, GUTS(t) is the match for *guts* in the above patterns; when t is positive, GUTS(t) is t, itself. Informally, the term formula for a positive literal means "the guts are true," and the formula for a negative literal means "the guts are false."

We think of `(not` *guts*`)` as the simplest form of negative literals, so we begin by developing a derived rule which can prove t = `(not` GUTS(t)`)` for any negative term, t. We make use of a few theorems to address the various kinds of negative terms.

Formal Theorem 35. Standardize equal x nil

`(equal x nil)` = `(not x)`

Proof.

`x` = `y` ∨ `(equal x y)` = `nil`	Ax. eq. when diff	
`x` = `nil` ∨ `(equal x nil)` = `nil`	Instantiation	(*1a)
`x` = `nil` ∨ `(if x y z)` = `y`	Ax. if when nnil	
`x` = `nil` ∨ `(if x nil t)` = `nil`	Instantiation	
`x` = `nil` ∨ `nil` = `(if x nil t)`	Dj. commute =	
`x` = `nil` ∨ `(equal x nil)` = `(if x nil t)`	Dj. trans. = *1a	(*1)
`x` ≠ `y` ∨ `(equal x y)` = `t`	Ax. eq., same	
`x` ≠ `nil` ∨ `(equal x nil)` = `t`	Instantiation	(*2a)
`x` ≠ `nil` ∨ `(if x y z)` = `z`	Axiom if when nil	
`x` ≠ `nil` ∨ `(if x nil t)` = `t`	Instantiation	
`x` ≠ `nil` ∨ `t` = `(if x nil t)`	Dj. commute =	
`x` ≠ `nil` ∨ `(equal x nil)` = `(if x nil t)`	Dj. trans. = *2a	(*2)
`(equal x nil)` = `(if x nil t)` ∨ `(equal x nil)` = `(if x nil t)`	Cut *1, *2	
`(equal x nil)` = `(if x nil t)`	Contraction	(*3)
`(not x)` = `(if x nil t)`	Definition of not	
`(if x nil t)` = `(not x)`	Commute =	
`(equal x nil)` = `(not x)`	Trans. = *3	□

Formal Theorem 36. Standardize equal nil x

`(equal nil x)` = `(not x)`

Proof.

`(equal x y)` = `(equal y x)`	Th. symmetry of eq.
`(equal nil x)` = `(equal x nil)`	Instantiation
`(equal x nil)` = `(not x)`	Th. standardize eq. x nil
`(equal nil x)` = `(not x)`	Trans. = □

Formal Theorem 37. Standardize iff x nil

`(iff x nil)` = `(not x)`

Proof.

`x` ≠ `nil` ∨ `(not x)` = `t`	Th. not when nil	
`x` ≠ `nil` ∨ `t` = `(not x)`	Dj. commute =	
`x` ≠ `nil` ∨ `(iff x nil)` = `t`	Th. iff nil, nil	
`x` ≠ `nil` ∨ `(iff x nil)` = `(not x)`	Dj. trans. =	(*1)
`x` = `nil` ∨ `(not x)` = `nil`	Th. not when nnil	
`x` = `nil` ∨ `nil` = `(not x)`	Dj. commute =	
`x` = `nil` ∨ `(iff x nil)` = `nil`	Th. iff nil, nnil	
`x` = `nil` ∨ `(iff x nil)` = `(not x)`	Dj. trans. =	
`(iff x nil)` = `(not x)` ∨ `(iff x nil)` = `(not x)`	Cut *1	
`(iff x nil)` = `(not x)`	Contraction	□

Formal Theorem 38. Standardize iff nil x

`(iff nil x)` = `(not x)`

Proof.

`(iff x y)` = `(iff y x)`	Th. symmetry of iff
`(iff nil x)` = `(iff x nil)`	Instantiation
`(iff x nil)` = `(not x)`	Th. standardize iff x nil
`(iff nil x)` = `(not x)`	Trans. = □

Derived Rule 135. Standardize negative term

$$\frac{}{t = \texttt{(not } \textsc{guts}(t)\texttt{)}}, \text{ where } t \text{ is a negative term}$$

Derivation.

If t is `(not` *guts*`)`, the by the reflexivity rule we are done.

If t is `(if` *guts* `nil t)`,

`(not x)` = `(if x nil t)`	Definition of not
`(if x nil t)` = `(not x)`	Commute =
`(if` *guts* `nil t)` = `(not` *guts*`)`	Instantiation

Otherwise, t is `(equal` *guts* `nil)`, `(equal nil` *guts*`)`, `(iff` *guts* `nil)`, or `(iff nil` *guts*`)`, and our goal follows by instantiating the above theorems. □

We say a literal is a *double negative* if it is a negative term with negative guts. To eliminate double negatives, we begin by simplifying them into the form `(not (not` x`))`.

Derived Rule 136. Standardize double negative term

$$\frac{}{t = \texttt{(not (not } \textsc{guts}(\textsc{guts}(t))\texttt{))}}, \text{ where } t \text{ is a double negative term}$$

Derivation.

$\textsc{guts}(t)$ = `(not` $\textsc{guts}(\textsc{guts}(t))$`)`	Std. negative term
`(not` $\textsc{guts}(t)$`)` = `(not (not` $\textsc{guts}(\textsc{guts}(t))$`))`	= by arguments
t = `(not` $\textsc{guts}(t)$`)`	Std. negative term
t = `(not (not` $\textsc{guts}(\textsc{guts}(t))$`))`	Transitivity of = □

Formal Theorem 39. If redux t

`(if t y z)` = `y`

Proof.

`t` ≠ `nil`	Axiom t not nil

`(if t y z)` $=$ `y`	If when not nil	$\square$

Formal Theorem 40. If redux nil

$$\texttt{(if nil y z)} = \texttt{z}$$

Proof.

`nil` $=$ `nil`	Reflexivity	
`(if nil y z)` $=$ `z`	If when nil	$\square$

Formal Theorem 41. If redux test

$$\texttt{(if (if x y z) p q)} = \texttt{(if x (if y p q) (if z p q))}$$

Proof.

`x` $=$ `nil` $\vee$ `(if x y z)` $=$ `y`	Ax. if when nnil	(*1a)
`p` $=$ `p`	Reflexivity	(*p)
`x` $=$ `nil` $\vee$ `p` $=$ `p`	Expansion	(*1b)
`q` $=$ `q`	Reflexivity	(*q)
`x` $=$ `nil` $\vee$ `q` $=$ `q`	Expansion	(*1c)
`x` $=$ `nil` $\vee$ `(if (if x y z) p q)` $=$ `(if y p q)`	Dj. $=$ args *1abc	(*1)
`x` $=$ `nil` $\vee$ `(if x (if y p q) (if z p q))` $=$ `(if y p q)`	Instantiation *1a	
`x` $=$ `nil` $\vee$ `(if y p q)` $=$ `(if x (if y p q) (if z p q))`	Dj. commute $=$	
`x` $=$ `nil` $\vee$ `(if (if x y z) p q)` $=$ `(if x (if y p q) (if z p q))`	Dj. trans. $=$ *1	(**1)
`x` $\neq$ `nil` $\vee$ `(if x y z)` $=$ `z`	Axiom if when nil	(*2a)
`x` $\neq$ `nil` $\vee$ `p` $=$ `p`	Expansion *p	(*2b)
`x` $\neq$ `nil` $\vee$ `q` $=$ `q`	Expansion *q	(*2c)
`x` $\neq$ `nil` $\vee$ `(if (if x y z) p q)` $=$ `(if z p q)`	Dj. $=$ args *2abc	(*2)
`x` $\neq$ `nil` $\vee$ `(if x (if y p q) (if z p q))` $=$ `(if z p q)`	Instantiation *2a	
`x` $\neq$ `nil` $\vee$ `(if z p q)` $=$ `(if x (if y p q) (if z p q))`	Dj. commute $=$	
`x` $\neq$ `nil` $\vee$ `(if (if x y z) p q)` $=$ `(if x (if y p q) (if z p q))`	Dj. trans. $=$ *2	(**2)

`(if (if x y z) p q)`	Cut **1, **2		
`    = (if x (if y p q) (if z p q))`			
`    ∨ (if (if x y z) p q)`			
`    = (if x (if y p q) (if z p q))`			
`(if (if x y z) p q)`	Contraction		□
`    = (if x (if y p q) (if z p q))`			

Formal Theorem 42. Not of not

`(not (not x)) = (if x t nil)`

Proof.

`(not x) = (if x nil t)`	Definition of not		
`(not (not x)) = (not (if x nil t))`	= by args		
`(not (if x nil t))`	Instantiation		
`    = (if (if x nil t) nil t)`			
`(not (not x)) = (if (if x nil t) nil t)`	Trans. =	(*1)	
`x = x`	Reflexivity	(*2a)	
`(if nil y z) = z`	Th. if redux nil		
`(if nil nil t) = t`	Instantiation	(*2b)	
`(if t y z) = y`	Th. if redux t		
`(if t nil t) = nil`	Instantiation	(*2c)	
`(if x (if nil nil t) (if t nil t))`	= args *2abc	(*2)	
`    = (if x t nil)`			
`(if (if x y z) p q)`	Th. if redux test		
`    = (if x (if y p q) (if z p q))`			
`(if (if x nil t) nil t)`	Instantiation		
`    = (if x (if nil nil t) (if t nil t))`			
`(if (if x nil t) nil t) = (if x t nil)`	Trans. = *2		
`(not (not x)) = (if x t nil)`	Trans. = *1		□

Formal Theorem 43. Not of not under iff

`(iff (not (not x)) x) = t`

Proof.

`(not (not x)) = (if x t nil)`	Th. not of not

`x` $=$ `x`	Reflexivity
`(iff (not (not x)) x)` $=$ `(iff (if x t nil) x)`	$=$ by args
`(iff (if x t nil) x)` $=$ `t`	Th. iff of if x t nil
`(iff (not (not x)) x)` $=$ `t`	Trans. $=$ $\square$

Derived Rule 137. Standardize double negative term under iff

$$\frac{}{\texttt{(iff } t \text{ GUTS}(\text{GUTS}(t))\texttt{)} = \texttt{t}}, \text{ where } t \text{ is a double negative term}$$

Derivation. Let t' be GUTS(GUTS(t)). Now,

$t =$ `(not (not` t'`))`	Std. dbl. neg. term	
`(iff` t `(not (not` t'`)))` $=$ `t`	Iff from $=$	(*1)
`(iff (not (not x)) x)` $=$ `t`	Th. not of not under iff	
`(iff (not (not` t'`))` t'`)` $=$ `t`	Instantiation	
`(iff` t t'`)` $=$ `t`	Transitivity of iff *1	$\square$

Since a term might have any number of negatives, we define an algorithm, NORMALIZE-NOTS, which repeatedly strips away double negatives until we are left with a either positive or singly negative term in our preferred form, i.e., `(not` *guts*`)`.

$$\text{NORMALIZE-NOTS}(a) \triangleq \begin{cases} a & \text{if } a \text{ is positive,} \\ \texttt{(not } \text{GUTS}(a)\texttt{)} & \text{otherwise, if GUTS}(a) \text{ is positive, or} \\ \text{NORMALIZE-NOTS}(\text{GUTS}(\text{GUTS}(a))) & \text{otherwise.} \end{cases}$$

We can see that this process leaves us with an `iff`-equivalent term, using the following rule.

Derived Rule 138. Normalize nots

$$\frac{}{\texttt{(iff } a \text{ NORMALIZE-NOTS}(a)\texttt{)} = \texttt{t}}$$

Derivation. If a is positive, then NORMALIZE-NOTS(a) is a and our goal is to show `(iff` a a`)` $=$ `t`, which follows from the iff reflexivity rule.

If a is singly negative, then NORMALIZE-NOTS(a) is (not GUTS(a)) and our goal is to show (iff a (not GUTS(a))) $=$ t. Now,

$a =$ (not GUTS(a))	Standardize negative term
(iff a (not GUTS(a))) $=$ t	Iff from $=$

Finally, suppose a is a double negative. Let a' be GUTS(GUTS(a)) and also let a'' be NORMALIZE-NOTS(a'). Here, we may recursively derive (iff a' a'') $=$ t, and our goal is to derive (iff a a'') $=$ t.

(iff a a') $=$ t	Std. dbl. neg. term under iff
(iff a' a'') $=$ t	Recursive construction
(iff a a'') $=$ t	Transitivity of iff $\square$

The first step in our clause-cleaning routine is to simplify every literal in each clause with NORMALIZE-NOTS. Since we can prove each simplified literal is iff-equivalent to the original literal, we can prove each original clause from a proof of the corresponding simplified clause with the update clause iff rule.

The next stage in our cleaning algorithm is to eliminate certain obvious clauses. We say that certain literals, viz. (not nil) and constants other than nil, are *obviously true*. Given an obviously true literal, a, it is straightforward to prove the term formula, $a \neq$ nil.

Derived Rule 139. Obvious term

$$\frac{}{a \neq \texttt{nil}}, \text{ where } a \text{ is obviously true}$$

Derivation. If a is a constant other than nil, then we need only use the $\neq$ constants rule to conclude $a \neq$ nil.

Otherwise, if a is (not nil), then we may derive $a \neq$ nil as follows.

(not x) = (if x nil t)	Definition of not	
(not nil) = (if nil nil t)	Instantiation	(*1)

$\texttt{(if nil y z)} = \texttt{z}$	Th. if redux nil
$\texttt{(if nil nil t)} = \texttt{t}$	Instantiation
$\texttt{(not nil)} = \texttt{t}$	Transitivity of $=$ *1
$\texttt{(not nil)} \neq \texttt{nil}$	Not nil from t $\square$

We say a clause is *obvious* whenever it contains an obviously true term. It is straightforward to prove any obvious clause. First, using the obvious term rule, we prove $t_i \neq \texttt{nil}$, where t_i is an obvious term in the clause. Then, by multi expansion, we may obtain a proof of the whole clause.

After eliminating the obvious clauses, we throw out some other easy-to-prove clauses. We say a pair of literals matching a and $\texttt{(not } a\texttt{)}$ are *complementary*, and if a clause contains any complementary literals, we call it a *complementary clause.* It is straightforward to prove any complementary clause. In particular, suppose the clause is $[t_1, \ldots, t_n]$, some t_i is a and some t_j is $\texttt{(not } a\texttt{)}$. Then,

$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} = \texttt{t}$	Th. not when nil
$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} \neq \texttt{nil}$	Disjoined not nil from t
$t_i \neq \texttt{nil} \vee t_j \neq \texttt{nil}$	Instantiation, $\texttt{x} \leftarrow a$
$t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Multi-or expansion

Next, we say certain literals, namely $\texttt{nil}$ and any term of the form $\texttt{(not } guts\texttt{)}$, where *guts* is a non-$\texttt{nil}$ constant, are *absurd.* Intuitively, absurd literals are useless when trying to prove a clause—they are like the "false" in $A \vee \text{false}$—so we would like to remove them from each clause.

Suppose our original clause is $C = [t_1, \ldots, t_n]$ and we throw away some (but not all) of the terms to obtain $D = [t_{i_1}, \ldots, t_{i_k}]$. Then given a proof of D, we may prove C using the generic subset rule. What if every literal in C is absurd? Then we have discovered C is unprovable. In this case, our cleaning routine immediately stops and returns the original goals unchanged (which is clearly justifiable). But we also return a flag that indicates an unprovable clause has been discovered, so that the problem may be reported to the user.

After removing absurd literals, we remove any duplicate literals from each clause, which is again justified by the generic subset rule.

Finally, we eliminate any *subsumed* clauses. That is, suppose our list of goal clauses contains C_1 and C_2, where the literals of C_1 are a subset of the literals of C_2. Then, we say that C_2 is subsumed by C_1: given a proof of C_1, we can prove C_2 via the generic subset rule.

To review, the clause-cleaning process takes a list of goal clauses as input, and produces a new, simpler list of goals by

1. standardizing not-variants like `(equal` x `nil)` into `(not` x`)`,
2. normalizing any multiply negative literals,
3. removing any clauses with obvious literals,
4. removing any clauses with complementary literals,
5. removing any absurd and redundant literals from each clause, and
6. removing any subsumed clauses.

For each of these steps, we may obtain proofs of the input goals when given proofs of the resulting clauses. Hence, the entire cleaning process may be justified: given proofs of the cleaned clauses, we can prove our original goals.

7.5 Clause Splitting

In an ordinary mathematical proof, if we want to show some property, P, follows from a compound condition like $A \vee (B \wedge C)$, we usually consider subcases.

That is, first we show P holds when we assume A, then we show it holds when we assume B and also assume C. In our system, `and`, `or`, and `cond` are just abbreviations for `if`-expressions, so an analogous situation occurs when a goal clause contains an `if`. In particular, instead of proving $[\ldots, \texttt{(if }a\ b\ c\texttt{)}, \ldots]$, it may be easier to prove both $[\ldots, \texttt{(not }a\texttt{)}, b, \ldots]$ and $[\ldots, a, c, \ldots]$, which together imply the original.

We now introduce an algorithm for splitting up a clause into new clauses based upon the `if`-expressions at the top of each literal. The core of our routine, CS-AUX, is a recursive function which takes the input clause in two pieces: $t_1, \ldots, t_n$, the literals which are left "to do", and $d_1, \ldots, d_m$, the literals which are already "done." Initially, the done list will be empty and the entire goal clause is placed in the to do list.

Below, we present a simplified version of CS-AUX. We do not necessarily assume the clauses have been cleaned before clause splitting begins, so our algorithm canonicalizes double negations and looks for negative terms in forms besides `(not` *guts*`)`. We write $\overline{a}$ to indicate any negative term whose guts are a, and $\overline{\overline{a}}$ to indicate any negative term whose guts are $\overline{a}$.

1. $\textsc{cs-aux}([], [d_{1\ldots m}]) \triangleq [[d_{1\ldots m}]]$
2. $\textsc{cs-aux}([\overline{\overline{a}}, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq \textsc{cs-aux}([a, t_{2\ldots n}], [d_{1\ldots m}])$
3. $\textsc{cs-aux}([\overline{\texttt{(if }a\ b\ c\texttt{)}}, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq$
   $$\textsc{app}\left(\begin{array}{l}\textsc{cs-aux}([\texttt{(not }a\texttt{)}, \texttt{(not }b\texttt{)}, t_{2\ldots n}], [d_{1\ldots m}]),\\ \textsc{cs-aux}([a, \texttt{(not }c\texttt{)}, t_{2\ldots n}], [d_{1\ldots m}])\end{array}\right)$$
4. $\textsc{cs-aux}([\texttt{(if }a\ b\ c\texttt{)}, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq$
   $$\textsc{app}\left(\begin{array}{l}\textsc{cs-aux}([\texttt{(not }a\texttt{)}, b, t_{2\ldots n}], [d_{1\ldots m}]),\\ \textsc{cs-aux}([a, c, t_{2\ldots n}], [d_{1\ldots m}])\end{array}\right)$$
5. $\textsc{cs-aux}([\overline{a}, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq \textsc{cs-aux}([t_{2\ldots n}], [\texttt{(not }a\texttt{)}, d_{1\ldots m}])$
6. $\textsc{cs-aux}([a, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq \textsc{cs-aux}([t_{2\ldots n}], [a, d_{1\ldots m}])$

To see that CS-AUX terminates, let $m(t)$ measure a term as follows,

$$m(\overline{a}) \triangleq 1 + m(a)$$
$$m(\texttt{(if}\ a\ b\ c\texttt{)}) \triangleq 1 + m(a) + m(b) + m(c)$$
$$m(\_) \triangleq 1.$$

and observe that $\sum m(t_i)$ decreases in each recursive call.

How can we justify CS-AUX? If we let $T_1, \ldots, T_n$ be the term formulas for $t_i, \ldots, t_n$, and similarly let $D_1, \ldots, D_m$ be the term formulas for $d_1, \ldots, d_m$, then we say the *step goal* for CS-AUX is $(T_1 \vee \cdots \vee T_n) \vee (D_1 \vee \cdots \vee D_m)$. We will now show:

– in the basis case, Line 1, the step goal may be derived given a proof of the resulting clause, i.e., given a proof of $D_1 \vee \cdots \vee D_m$, and

– in the recursive cases, Lines 2-6, the step goal may be derived given proofs of the step goals for each recursive call of CS-AUX.

Together, by induction, these results allow us to derive the step goal for CS-AUX when given proofs of the clauses it produces.

These derivations are somewhat involved, so we address each line in turn. We sometimes find it convenient to develop auxiliary rules to handle the various cases for n and m. These rules are tedious, so we only summarize them here and leave their derivations to Appendix B.

Line 1. CS-AUX$([], [d_{1\ldots m}]) \triangleq [[d_{1\ldots m}]]$

This is our basis case, and it is trivial. Our goal is to prove $D_1 \vee \cdots \vee D_m$, and we are given a proof of $[d_{1\ldots m}]$. That is, we have been given a proof of our goal.

Line 2. CS-AUX$([\overline{\overline{a}}, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq$ CS-AUX$([a, t_{2\ldots n}], [d_{1\ldots m}])$

We make use of two rules from Appendix B.

Derived Rule 140. Aux split double negate lemma1

$$\frac{\begin{array}{l}(b \neq \texttt{nil} \vee P) \vee Q \\ \texttt{(iff } a \ b\texttt{)} = \texttt{t}\end{array}}{(a \neq \texttt{nil} \vee P) \vee Q}$$

Derived Rule 141. Aux split double negate lemma2

$$\frac{\begin{array}{l}b \neq \texttt{nil} \vee P \\ \texttt{(iff } a \ b\texttt{)} = \texttt{t}\end{array}}{a \neq \texttt{nil} \vee P}$$

Our goal for this line is to derive $(\overline{\overline{a}} \neq \texttt{nil} \vee T_{2...n}) \vee D_{1...m}$, and we may assume we are given a proof of $(a \neq \texttt{nil} \vee T_{2...n}) \vee D_{1...m}$. As a lemma, we may derive $\texttt{(iff } \overline{\overline{a}} \ a\texttt{)} = \texttt{t}$ using the standardize double negative term under iff rule. Then, our goal follows from either substitute iff into literal or one of the above lemmas, as appropriate for n and m.

Line 3.

$$\begin{array}{l}\text{CS-AUX}([\overline{\texttt{(if } a \ b \ c\texttt{)}}, t_{2...n}], [d_{1...m}]) \triangleq \\ \quad \text{APP}\left(\begin{array}{l}\text{CS-AUX}([\texttt{(not } a\texttt{)}, \texttt{(not } b\texttt{)}, t_{2...n}], [d_{1...m}]), \\ \text{CS-AUX}([a, \texttt{(not } c\texttt{)}, t_{2...n}], [d_{1...m}])\end{array}\right)\end{array}$$

We make use of three rules from Appendix B.

Derived Rule 142. Aux split negative

$$\frac{\begin{array}{l}\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil} \\ a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}\end{array}}{\texttt{(not (if } a \ b \ c\texttt{))} \neq \texttt{nil}}$$

Derived Rule 143. Aux split negative 1

$$\frac{\begin{array}{l}(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil} \vee P) \vee Q \\ (a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil} \vee P) \vee Q \\ \mathit{t1} = \texttt{(not (if } a \ b \ c\texttt{))}\end{array}}{(\mathit{t1} \neq \texttt{nil} \vee P) \vee Q}$$

Derived Rule 144. Aux split negative 2

$$\frac{\begin{array}{l}(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil}) \vee P \\ (a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}) \vee P \\ \mathit{t1} = \texttt{(not (if } a \ b \ c\texttt{))}\end{array}}{\mathit{t1} \neq \texttt{nil} \vee P}$$

Our goal for this line is to derive $(\overline{\texttt{(if}\ a\ b\ c\texttt{)}} \neq \texttt{nil} \vee T_{2\ldots n}) \vee D_{1\ldots m}$, given proofs of $(\texttt{(not}\ a\texttt{)} \neq \texttt{nil} \vee \texttt{(not}\ b\texttt{)} \neq \texttt{nil} \vee T_{2\ldots n}) \vee D_{1\ldots m}$ and $(a \neq \texttt{nil} \vee \texttt{(not}\ c\texttt{)} \neq \texttt{nil} \vee T_{2\ldots n}) \vee D_{1\ldots m}$.

As a lemma, we may derive $\overline{\texttt{(if}\ a\ b\ c\texttt{)}} = \texttt{(not (if}\ a\ b\ c\texttt{))}$ using the standardize negative term rule. When $n = 0$ and $m = 0$, we have $\texttt{(not (if}\ a\ b\ c\texttt{))} \neq \texttt{nil}$ by the aux split negative rule; this can be combined with our lemma via the substitute into $\neq$ rule to prove our goal. Otherwise, when $n \geq 1$ and $m \geq 1$, our goal may be derived from our premises and our lemma via the aux split negative 1 rule. Finally, if only one of $n = 0$ or $m = 0$, our goal follows from the aux split negative 2 rule, but note that in the case where $m = 0$, the premises must first be coerced into the appropriate form via the associativity rule.

Line 4.
$$\begin{array}{l}\textsc{cs-aux}([\texttt{(if}\ a\ b\ c\texttt{)}, t_{2\ldots n}], [d_{1\ldots m}]) \triangleq \\ \quad \textsc{app}\left(\begin{array}{l}\textsc{cs-aux}([\texttt{(not}\ a\texttt{)}, b, t_{2\ldots n}], [d_{1\ldots m}]), \\ \textsc{cs-aux}([a, c, t_{2\ldots n}], [d_{1\ldots m}])\end{array}\right)\end{array}$$

We make use of three rules from Appendix B.

Derived Rule 145. Aux split positive

$$\frac{\begin{array}{l}\texttt{(not}\ a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \\ a \neq \texttt{nil} \vee c \neq \texttt{nil}\end{array}}{\texttt{(if}\ a\ b\ c\texttt{)} \neq \texttt{nil}}$$

Derived Rule 146. Aux split positive 1

$$\frac{\begin{array}{l}(\texttt{(not}\ a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \vee P) \vee Q \\ (a \neq \texttt{nil} \vee c \neq \texttt{nil} \vee P) \vee Q\end{array}}{(\texttt{(if}\ a\ b\ c\texttt{)} \neq \texttt{nil} \vee P) \vee Q}$$

Derived Rule 147. Aux split positive 2

$$\frac{\begin{array}{l}(\texttt{(not}\ a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}) \vee P \\ (a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee P\end{array}}{\texttt{(if}\ a\ b\ c\texttt{)} \neq \texttt{nil} \vee P}$$

Our goal for this line is to derive $(\texttt{(if}\ a\ b\ c\texttt{)} \neq \texttt{nil} \vee T_{2\ldots n}) \vee D_{1\ldots m}$ when given proofs of $(\texttt{(not}\ a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \vee T_{2\ldots n}) \vee D_{1\ldots m}$ and $(a \neq \texttt{nil} \vee c \neq$

$\texttt{nil} \vee T_{2\dots n}) \vee D_{1\dots m}$. When $n = 1$ and $m = 0$, we may use the aux split positive rule and when $n > 1$ and $m > 0$, we may use the aux split positive 1 rule. When $n = 1$ or $m = 0$, the goal from the aux split positive 2 rule, but note that if $m = 0$ the premises must first be coerced into the appropriate form via the associativity rule.

Line 5. $\textsc{cs-aux}([\overline{a}, t_{2\dots n}], [d_{1\dots m}]) \triangleq \textsc{cs-aux}([t_{2\dots n}], [\texttt{(not } a\texttt{)}, d_{1\dots m}])$

We make use of two rules from Appendix B.

Derived Rule 148. Aux split default 1

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \vee Q \\ a = b \end{array}}{(a \neq \texttt{nil} \vee P) \vee Q}$$

Derived Rule 149. Aux split default 2

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \\ a = b \end{array}}{a \neq \texttt{nil} \vee P}$$

In this line, we need to establish $(\overline{a} \neq \texttt{nil} \vee T_{2\dots n}) \vee D_{1\dots m}$ when given $T_{2\dots n} \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee D_{1\dots m}$. As a lemma, we may derive $\overline{a} = \texttt{(not } a\texttt{)}$ using the standardize negative term rule. Now, if $n = 1$ and $m = 0$, our goal follows from our premise, our lemma, and the substitute into $\neq$ rule. Otherwise, if $n > 1$ and $m > 0$, we use aux split default 1. Finally, if only one of $n = 1$ or $m = 0$ holds, we use aux split default 2, but note that in the case where $m = 0$, we must first prepare our premise using the commute or rule.

Line 6. $\textsc{cs-aux}([a, t_{2\dots n}], [d_{1\dots m}]) \triangleq \textsc{cs-aux}([t_{2\dots n}], [a, d_{1\dots m}])$

Our goal for this line is to derive $(a \neq \texttt{nil} \vee T_{2\dots n}) \vee D_{1\dots m}$ when given a proof of $T_{2\dots n} \vee a \neq \texttt{nil} \vee D_{1\dots m}$. If $n = 1$ and $m = 0$, our premise is the same as our goal so there is nothing to do. Otherwise, if $n = 1$ and $m > 0$, we only need to commute our premise with commute or. Finally, if $n > 1$ and $m > 0$, we can derive our goal with the following rule.

Derived Rule 150. Aux split default 3

$$\frac{P \vee A \vee Q}{(A \vee P) \vee Q}$$

Derivation. (24)

$P \vee A \vee Q$	Given
$P \vee Q \vee A$	Dj. commute or
$(P \vee Q) \vee A$	Associativity
$A \vee P \vee Q$	Commute or
$(A \vee P) \vee Q$	Associativity

□

The basic CS-AUX algorithm presented above can lead to an exponential increase in the number of goal clauses. One way to reduce this is to first check whether there is an easy way to prove either of the new goals before recurring. In particular, we check whether one of the newly produced literals (1) is obvious, or (2) is the complement of another literal. When this occurs, it is straightforward to prove the new goal, which obviates the need to recur down that branch.

Even with this improvement, CS-AUX sometimes generates too many subgoals. It is useful to add a counter to CS-AUX which can be used to force it to stop splitting after a certain number of clauses have been generated. This way, other routines (such as our cleaning and rewriting) may be able to prove some of the clauses before further splitting is done.

7.6 If Lifting

Our clause-splitting algorithm only considers `if`-expressions at the top of each literal, so we now introduce a routine that lifts more deeply occurring `if`-expressions to the top of a term. For instance, the term (*f* (if *a* *b* *c*)) can be lifted to produce the provably equal term (if *a* (*f* *b*) (*f* *c*)).

We say a term is *simple* when it is `if`-free; more formally, constants and variables are always simple, `(`f t_1 ... t_n`)` is simple when each t_i is simple, and `((lambda (`x_1 ... x_n`)` β`)` t_1 ... t_n`)` is simple when each t_i is simple. Note that we do not consider `if`-expressions which occur within the body of a lambda, and generally throughout this section we treat lambda bodies as opaque.

We say x is a *subterm* of y whenever x occurs within y (ignoring lambda bodies). For instance, the subterms of `(cons a b)` are `a`, `b`, and `(cons a b)`, itself. The tests of a term, t, are the set of all a such that `(if` a b c`)` is a subterm of t. For instance, the tests of `(if (if a t nil) b c)` are `a` and `(if a t nil)`.

We say a term is *lifted* when, ignoring lambda bodies, the actuals of every lambda and the arguments of every function application besides `if` are simple. Given any term, our lifting algorithm produces a provably equal, lifted term; given any lifted term, our clause-splitting algorithm produces clauses that contain only simple literals. In this sense, lifting followed by clause splitting is complete, and produces clauses with no `if`-expressions remaining.

How does our lifting algorithm proceed?

To begin, we say a *splitting assignment* is a mapping from terms to truth values. Given a splitting assignment, A, we may *factor* a term, t, to produce a new term, which we denote $t|_A$. Intuitively, $t|_A$ is a simplification of t where we assume the splitting assignment's bindings are satisfied, and reduce any `if`-expressions having to do with the bound terms. In particular, if t is a constant or a variable then $t|_A \triangleq t$, for `if`-expressions

$$\texttt{(if } a \ b \ c\texttt{)}|_A \triangleq \begin{cases} b|_A & \text{if } A \text{ binds } a|_A \text{ to } \texttt{t}, \\ c|_A & \text{if } A \text{ binds } a|_A \text{ to } \texttt{nil}, \text{ or} \\ \texttt{(if } a|_A \ b|_A \ c|_A\texttt{)} & \text{otherwise,} \end{cases}$$

for any other function applications

$$(f\ \ t_1\ \ \ldots\ \ t_n)|_A \triangleq (f\ \ t_1|_A\ \ \ldots\ \ t_n|_A),$$

and for lambda abbreviations

$$\begin{aligned}&(\texttt{(lambda}\ (x_1\ \ldots\ x_n)\ \beta)\ t_1\ \ldots\ t_n)|_A \triangleq\\ &\quad(\texttt{(lambda}\ (x_1\ \ldots\ x_n)\ \beta)\ t_1|_A\ \ldots\ t_n|_A).\end{aligned}$$

To justify factoring, we will show that $t = t|_A$ is provable when the the bindings made by A are satisfied. It is easy to see that when A is empty, $t|_A$ is the same as t, so $t = t|_A$ follows by reflexivity. But when A is non-empty, we want to show something like "A" $\rightarrow t = t|_A$, or in other words, "$\neg A$" $\vee\, t = t|_A$. To make this more precise, suppose A binds the terms $t_1, \ldots, t_n$ to the truth values $v_1, \ldots, v_n$, respectively. Then, we first define $\textsc{bhyp}(t_i)$, the binding hypothesis of each t_i, as

$$\textsc{bhyp}(t_i) \triangleq \begin{cases} t_i = \texttt{nil} & \text{if } v_i \text{ is } \texttt{t}\text{, or}\\ t_i \neq \texttt{nil} & \text{otherwise,}\end{cases}$$

and we define $\textsc{ahyps}(A)$, the assignment hypothesis for A, as

$$\textsc{ahyps}(A) \triangleq \textsc{bhyp}(t_1) \vee \cdots \vee \textsc{bhyp}(t_n).$$

Intuitively, $\textsc{ahyps}(A)$ describes "$\neg A$," and is true only when one of the bindings from A is violated. So when A is non-empty, we want to be able to derive $\textsc{ahyps}(A) \vee t = t|_A$. We begin with a couple of auxiliary rules that assist in the derivation.

Derived Rule 151. Factor lemma 1

$$\frac{\begin{array}{l} P \vee a2 \neq \texttt{nil}\\ P \vee a1 = a2\\ P \vee b1 = b2\end{array}}{P \vee (\texttt{if}\ \ a1\ \ b1\ \ c) = b2}$$

Derivation. (79)

$P \lor a2 \neq \texttt{nil}$	Given	
$P \lor a1 = a2$	Given	
$P \lor a1 \neq \texttt{nil}$	Dj. sub. into $\neq$	(*1)
$\texttt{x} = \texttt{nil} \lor \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$a1 = \texttt{nil} \lor \texttt{(if } a1\ b1\ \texttt{c)} = b1$	Instantiation	
$P \lor a1 = \texttt{nil} \lor \texttt{(if } a1\ b1\ \texttt{c)} = b1$	Expansion	
$P \lor \texttt{(if } a1\ b1\ \texttt{c)} = b1$	Dj. mp2 *1	
$P \lor b1 = b2$	Given	
$P \lor \texttt{(if } a1\ b1\ \texttt{c)} = b2$	Dj. trans. $=$	$\square$

Derived Rule 152. Factor lemma 2

$$\begin{array}{l} P \lor a2 = \texttt{nil} \\ P \lor a1 = a2 \\ P \lor c1 = c2 \\ \hline P \lor \texttt{(if } a1\ b\ c1\texttt{)} = c2 \end{array}$$

Derivation. (79)

$P \lor a2 = \texttt{nil}$	Given	
$P \lor a1 = a2$	Given	
$P \lor a1 = \texttt{nil}$	Dj. trans. $=$	(*1)
$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$a1 \neq \texttt{nil} \lor \texttt{(if } a1\ b\ c1\texttt{)} = c1$	Instantiation	
$P \lor a1 \neq \texttt{nil} \lor \texttt{(if } a1\ b\ c1\texttt{)} = c1$	Expansion	
$P \lor \texttt{(if } a1\ b\ c1\texttt{)} = c1$	Dj. mp *1	
$P \lor c1 = c2$	Given	
$P \lor \texttt{(if } a1\ b\ c1\texttt{)} = c2$	Dj. trans. $=$	$\square$

Derived Rule 153. Factor

$$\frac{}{\textsc{ahyps}(A) \lor t = t|_A}, \text{ where } A \text{ is non-empty}$$

Derivation.

If t is a constant or a variable, then $t|_A$ is just t and by reflexivity we may conclude $t = t|_A$; then by expansion we have $\textsc{ahyps}(A) \lor t = t|_A$.

If t is a function application other than `if`, say $(f\ \ a_1\ \ \ldots\ \ a_n)$, then $t|_A$ is $(f\ \ a_1|_A\ \ \ldots\ \ a_n|_A)$. We may recursively derive $\textsc{ahyps}(A) \vee a_i = a_i|_A$ for each i. Now our goal, $\textsc{ahyps}(A) \vee (f\ \ a_1\ \ \ldots\ \ a_n) = (f\ \ a_1|_A\ \ \ldots\ \ a_n|_A)$, follows from the disjoined = by args rule.

If t is a lambda abbreviation, then the situation is analogous except that we use the disjoined lambda = by args rule instead.

Otherwise, t is $(\texttt{if}\ \ a\ \ b\ \ c)$, and there are three cases. If A does not give a binding for $a|_A$, then $t|_A$ is $(\texttt{if}\ \ a|_A\ \ b|_A\ \ c|_A)$ and we can use the same argument as for other function applications.

Next, if A binds $a|_A$ to `t`, then $t|_A$ is $b|_A$, so our goal is to show $\textsc{ahyps}(A) \vee (\texttt{if}\ \ a\ \ b\ \ c) = b|_A$. We may recursively derive $\textsc{ahyps}(A) \vee a = a|_A$ and $\textsc{ahyps}(A) \vee b = b|_A$. Then,

$a|_A \neq \texttt{nil} \vee a|_A = \texttt{nil}$	Propositional schema	
$a|_A = \texttt{nil} \vee a|_A \neq \texttt{nil}$	Commute or	
$\textsc{ahyps}(A) \vee a|_A \neq \texttt{nil}$	Multi-assoc expansion	(*1)
$\textsc{ahyps}(A) \vee a = a|_A$	Recursive construction	(*2)
$\textsc{ahyps}(A) \vee b = b|_A$	Recursive construction	(*3)
$\textsc{ahyps}(A) \vee (\texttt{if}\ \ a\ \ b\ \ c) = b|_A$	Factor lemma 1, *1–3	

Finally, if A binds $a|_A$ to `nil`, then $t|_A$ is $c|_A$, so our goal is to show $\textsc{ahyps}(A) \vee (\texttt{if}\ \ a\ \ b\ \ c) = c|_A$. Now we may recursively derive two formulas: $\textsc{ahyps}(A) \vee a = a|_A$ and $\textsc{ahyps}(A) \vee c = c|_A$. Then,

$a|_A \neq \texttt{nil} \vee a|_A = \texttt{nil}$	Propositional schema	
$\textsc{ahyps}(A) \vee a|_A = \texttt{nil}$	Multi-assoc expansion	(*1)
$\textsc{ahyps}(A) \vee a = a|_A$	Recursive construction	(*2)
$\textsc{ahyps}(A) \vee c = c|_A$	Recursive construction	(*3)
$\textsc{ahyps}(A) \vee (\texttt{if}\ \ a\ \ b\ \ c) = c|_A$	Factor lemma 2, *1–3	□

Now that factoring is in place, we are ready to introduce the main term transformation for if-lifting, which we call CASES. Given a term x we want to transform, and list of terms, $cs = [c_1, \ldots, c_n]$, which are the various cases to consider, we can create a new term, x', which reorganizes x into its factorings under the possible splitting assignments to the cases. Intuitively, x' is something like

```
(if c₁
    ⋱
      (if cₙ
          x|[c₁←t, ..., cₙ₋₁←t, cₙ←t]
        x|[c₁←t, ..., cₙ₋₁←t, cₙ←nil])
    ...)
  x|[c₁←nil, ..., cₙ₋₁←nil, cₙ←nil]),
```

except that we collapse `if`-expressions whose true and false branches are the same. More precisely, x' is the result of the algorithm $\textsc{cases}(x, cs, A)$, which as inputs takes a term x, the list of cases, cs, and a splitting assignment, A, which is ordinarily empty to begin with. The base case is

$$\textsc{cases}(x, [], A) \triangleq x|_A,$$

and otherwise, we say

$$\textsc{cases}(x, [c_1, \ldots, c_n], A) \triangleq \begin{cases} xt & \text{if } xt \text{ is } xf\text{, or} \\ \texttt{(if } c_1 \;\; xt \;\; xf\texttt{)} & \text{otherwise,} \end{cases}$$

where xt and xf are recursively defined as follows

$$xt \triangleq \textsc{cases}(x, [c_2, \ldots, c_n], c_1 \leftarrow \texttt{t} :: A), \text{ and}$$

$$xf \triangleq \textsc{cases}(x, [c_2, \ldots, c_n], c_1 \leftarrow \texttt{nil} :: A).$$

The term produced by CASES is provably equal to x when the bindings made by A are satisfied. More precisely, if A is empty then we can prove $x = \textsc{cases}(x, cs, A)$ and otherwise we can prove $\textsc{ahyps}(A) \vee x = \textsc{cases}(x, cs, A)$. The main auxiliary rules we need are the following.

Formal Theorem 44. Cases lemma

$$\neg(\texttt{x} = \texttt{nil} \vee \texttt{a} = \texttt{y}) \vee \neg(\texttt{x} \neq \texttt{nil} \vee \texttt{a} = \texttt{z}) \vee \texttt{a} = \texttt{(if x y z)}$$

Proof.

$x = \mathtt{nil} \lor (\mathtt{if}\ x\ y\ z) = y$	Ax. if when nnil	
$x = \mathtt{nil} \lor y = (\mathtt{if}\ x\ y\ z)$	Dj. commute =	
$(\neg(x = \mathtt{nil} \lor a = y) \lor x = \mathtt{nil}) \lor y = (\mathtt{if}\ x\ y\ z)$	Multi assoc exp.	(*1a)
$\neg(x = \mathtt{nil} \lor a = y) \lor x = \mathtt{nil} \lor a = y$	Prop. schema	
$(\neg(x = \mathtt{nil} \lor a = y) \lor x = \mathtt{nil}) \lor a = y$	Associativity	
$(\neg(x = \mathtt{nil} \lor a = y) \lor x = \mathtt{nil}) \lor a = (\mathtt{if}\ x\ y\ z)$	Dj. trans. = *1a	
$\neg(x = \mathtt{nil} \lor a = y) \lor x = \mathtt{nil} \lor a = (\mathtt{if}\ x\ y\ z)$	Right assoc.	
$(x = \mathtt{nil} \lor a = (\mathtt{if}\ x\ y\ z)) \lor \neg(x = \mathtt{nil} \lor a = y)$	Commute or	
$x = \mathtt{nil} \lor a = (\mathtt{if}\ x\ y\ z) \lor \neg(x = \mathtt{nil} \lor a = y)$	Right assoc.	(*1)
$x \neq \mathtt{nil} \lor (\mathtt{if}\ x\ y\ z) = z$	Axiom if when nil	
$x \neq \mathtt{nil} \lor z = (\mathtt{if}\ x\ y\ z)$	Dj. commute =	
$(\neg(x \neq \mathtt{nil} \lor a = z) \lor x \neq \mathtt{nil}) \lor z = (\mathtt{if}\ x\ y\ z)$	Multi assoc exp.	(*2a)
$\neg(x \neq \mathtt{nil} \lor a = z) \lor x \neq \mathtt{nil} \lor a = z$	Prop. schema	
$(\neg(x \neq \mathtt{nil} \lor a = z) \lor x \neq \mathtt{nil}) \lor a = z$	Associativity	
$(\neg(x \neq \mathtt{nil} \lor a = z) \lor x \neq \mathtt{nil}) \lor a = (\mathtt{if}\ x\ y\ z)$	Dj. trans. = *2a	
$\neg(x \neq \mathtt{nil} \lor a = z) \lor x \neq \mathtt{nil} \lor a = (\mathtt{if}\ x\ y\ z)$	Right assoc.	
$(x \neq \mathtt{nil} \lor a = (\mathtt{if}\ x\ y\ z)) \lor \neg(x \neq \mathtt{nil} \lor a = z)$	Commute or	
$x \neq \mathtt{nil} \lor a = (\mathtt{if}\ x\ y\ z) \lor \neg(x \neq \mathtt{nil} \lor a = z)$	Right assoc.	(*2)
$(a = (\mathtt{if}\ x\ y\ z) \lor \neg(x = \mathtt{nil} \lor a = y))$ $\lor a = (\mathtt{if}\ x\ y\ z) \lor \neg(x \neq \mathtt{nil} \lor a = z)$	Cut *1, *2	
$(a = (\mathtt{if}\ x\ y\ z) \lor a = (\mathtt{if}\ x\ y\ z))$ $\lor \neg(x \neq \mathtt{nil} \lor a = z) \lor \neg(x = \mathtt{nil} \lor a = y)$	Dj. assoc lm. 3	
$(a = (\mathtt{if}\ x\ y\ z) \lor a = (\mathtt{if}\ x\ y\ z))$ $\lor \neg(x = \mathtt{nil} \lor a = y) \lor \neg(x \neq \mathtt{nil} \lor a = z)$	Dj. commute or	
$(\neg(x = \mathtt{nil} \lor a = y) \lor \neg(x \neq \mathtt{nil} \lor a = z))$ $\lor a = (\mathtt{if}\ x\ y\ z) \lor a = (\mathtt{if}\ x\ y\ z)$	Commute or	
$(\neg(x = \mathtt{nil} \lor a = y) \lor \neg(x \neq \mathtt{nil} \lor a = z))$ $\lor a = (\mathtt{if}\ x\ y\ z)$	Dj. contraction	
$\neg(x = \mathtt{nil} \lor a = y)$ $\lor \neg(x \neq \mathtt{nil} \lor a = z) \lor a = (\mathtt{if}\ x\ y\ z)$	Right assoc.	□

Derived Rule 154. Cases lemma1

$$\frac{\begin{array}{c} x = \mathtt{nil} \lor a = b \\ x \neq \mathtt{nil} \lor a = c \end{array}}{a = (\mathtt{if}\ x\ b\ c)}$$

Derivation. (12)

$\neg(x = \mathtt{nil} \lor a = y)$ $\lor \neg(x \neq \mathtt{nil} \lor a = z) \lor a = (\mathtt{if}\ x\ y\ z)$	Cases lemma

$\neg(x = \texttt{nil} \vee a = b) \vee \neg(x \neq \texttt{nil} \vee a = c) \vee a = (\texttt{if}\ \ x\ \ b\ \ c)$	Instantiation
$x = \texttt{nil} \vee a = b$	Given
$\neg(x \neq \texttt{nil} \vee a = c) \vee a = (\texttt{if}\ \ x\ \ b\ \ c)$	Modus ponens
$x \neq \texttt{nil} \vee a = c$	Given
$a = (\texttt{if}\ \ x\ \ b\ \ c)$	Modus ponens $\square$

Derived Rule 155. Disjoined cases lemma1

$$\frac{\begin{array}{l} P \vee x = \texttt{nil} \vee a = b \\ P \vee x \neq \texttt{nil} \vee a = c \end{array}}{P \vee a = (\texttt{if}\ \ x\ \ b\ \ c)}$$

Derivation. (31)

$\neg(\mathsf{x} = \texttt{nil} \vee \mathsf{a} = \mathsf{y}) \vee \neg(\mathsf{x} \neq \texttt{nil} \vee \mathsf{a} = \mathsf{z}) \vee \mathsf{a} = (\texttt{if x y z})$	Cases lemma
$\neg(x = \texttt{nil} \vee a = b) \vee \neg(x \neq \texttt{nil} \vee a = c) \vee a = (\texttt{if}\ \ x\ \ b\ \ c)$	Instantiation
$P \vee \neg(x = \texttt{nil} \vee a = b) \vee \neg(x \neq \texttt{nil} \vee a = c) \vee a = (\texttt{if}\ \ x\ \ b\ \ c)$	Expansion
$P \vee x = \texttt{nil} \vee a = b$	Given
$P \vee \neg(x \neq \texttt{nil} \vee a = c) \vee a = (\texttt{if}\ \ x\ \ b\ \ c)$	Dj. modus ponens
$P \vee x \neq \texttt{nil} \vee a = c$	Given
$P \vee a = (\texttt{if}\ \ x\ \ b\ \ c)$	Dj. modus ponens $\square$

To produce smaller proofs, it is also useful to somewhat optimize the simple propositional manipulation, below.

Derived Rule 156. Lhs commute or then rassoc

$$\frac{(A \vee B) \vee C}{B \vee A \vee C}$$

Derivation. (25)

$(A \vee B) \vee C$	Given
$C \vee A \vee B$	Commute or
$(C \vee A) \vee B$	Associativity
$B \vee C \vee A$	Commute or
$B \vee A \vee C$	Dj. commute or $\square$

We can now justify the CASES rule.

Derived Rule 157. Cases

$$\frac{}{[\textsc{ahyps}(A) \vee]\ x = \textsc{cases}(x, cs, A)}$$

By this notation, we mean if A is empty we may derive $x = \textsc{cases}(x, cs, A)$, and otherwise we may derive $\textsc{ahyps}(A) \vee x = \textsc{cases}(x, cs, A)$.

Derivation.

As a basis, if cs is empty then $\textsc{cases}(x, cs, A)$ is $x|_A$. When A is empty, $x|_A$ is just x, so our goal follows from reflexivity. Otherwise, we want to show $\textsc{ahyps}(A) \vee x = x|_A$, which we may do with the factor rule.

Otherwise, let cs be $[c_1, \ldots, c_n]$, and let xt and xf be as above. To begin with, if A is empty, we may recursively derive

$$(*1) \quad c_1 = \texttt{nil} \vee x = xt, \text{ and}$$
$$(*2) \quad c_1 \neq \texttt{nil} \vee x = xf.$$

Now, if xt and xf are the same, then $\textsc{cases}(x, cs, A)$ is xt and our goal is to show $x = xt$, which we can do as follows.

$x = xt \vee x = xt$	Cut *1, *2
$x = xt$	Contraction

Otherwise, xt and xf are distinct, so $\textsc{cases}(x, cs, A)$ is $\texttt{(if } c_1\ xt\ xf\texttt{)}$, and our goal is to show $x = \texttt{(if } c_1\ xt\ xf\texttt{)}$, which follows directly from *1 and *2 using cases lemma1.

Otherwise, suppose A is non-empty, so we may recursively derive

$$(*1) \quad (c_1 = \texttt{nil} \vee \textsc{ahyps}(A)) \vee x = xt, \text{ and}$$

$$(*2) \quad (c_1 \neq \texttt{nil} \vee \textsc{ahyps}(A)) \vee x = xf.$$

Now, if xt and xf are the same, our goal is to show $\textsc{ahyps}(A) \vee x = xt$, which we can do as follows.

$c_1 = \texttt{nil} \vee \textsc{ahyps}(A) \vee x = xt$	Right associativity	*1
$c_1 \neq \texttt{nil} \vee \textsc{ahyps}(A) \vee x = xt$	Right associativity	*2
$(\textsc{ahyps}(A) \vee x = xt) \vee \textsc{ahyps}(A) \vee x = xt$	Cut	
$\textsc{ahyps}(A) \vee x = xt$	Contraction	

Finally, if xt and xf are distinct, our goal is $\textsc{ahyps}(A) \vee x = \texttt{(if } c_1\ xt\ xf\texttt{)}$. Then,

$\textsc{ahyps}(A) \vee c_1 = \texttt{nil} \vee x = xt$	Lhs commute or then rassoc	*1
$\textsc{ahyps}(A) \vee c_1 \neq \texttt{nil} \vee x = xf$	Lhs commute or then rassoc	*2
$\textsc{ahyps}(A) \vee x = \texttt{(if } c_1\ xt\ xf\texttt{)}$	Disjoined cases lemma1	$\square$

We say the *unlifted subterms* of a term are the top-level subterms which cause the term not to be lifted. For instance, the unlifted subterms of

```
(if a
    (if b
        (f c (if x nil t))
      nil)
  (g (if y nil t)))
```

are `(f c (if x nil t))` and `(g (if y nil t))`. Our if-lifting routine is an iterative process which, in each pass, transforms every unlifted subterm, u, by applying CASES to u, using the simple tests of u as the cases. For the example term above, in one pass we would transform the subterm `(f c (if x nil t))` by splitting it into cases on `x`, and transform `(g (if y nil t))` by splitting it into cases on `y`. The resulting term would be

```
(if a
    (if b
        (if x (f c nil) (f c t))
      nil)
  (if y (g nil) (g t))),
```

and since this resulting term is lifted, no additional passes would be necessary.

We implement each pass of our if-lifting routine with the operation $\textsc{lift1}$. For constants and variables we define $\textsc{lift1}(x) \triangleq x$, for `if`-expressions,

$$\textsc{lift1}(\texttt{(if}\ \ a\ \ b\ \ c\texttt{)}) \triangleq \texttt{(if}\ \ \textsc{lift1}(a)\ \ \textsc{lift1}(b)\ \ \textsc{lift1}(c)\texttt{)},$$

and for any other function application or lambda abbreviation,

$$\textsc{lift1}(x) \triangleq \begin{cases} x & \text{if every argument is simple, or} \\ \textsc{cases}(x, \textsc{stests}(x), []) & \text{otherwise,} \end{cases}$$

where $\textsc{stests}(x)$ gathers the simple tests of x.

Derived Rule 158. Lift1

$$\frac{}{\mathrm{x} = \textsc{lift1}(\mathrm{x})}$$

Derivation. If x is a constant or a variable, this is trivial by reflexivity.

If x is `(if` a b c`)`, we may recursively derive $a = \textsc{lift1}(a)$, $b = \textsc{lift1}(b)$, and $c = \textsc{lift1}(c)$, and our goal follows from the = by args rule.

Otherwise, if x is any other function application or lambda abbreviation, then there are two cases. If every argument is simple, we only need to show $x = x$, which follows from reflexivity. Otherwise, we need to show $x = \textsc{cases}(x, \textsc{stests}(x), [])$, which follows from the cases rule. □

Our full if-lifting algorithm, $\textsc{lift}(x)$, repeatedly applies $\textsc{lift1}$ until a fixed-point is reached. Since we can prove $x = \textsc{lift1}(x)$ for each step we take, it is easy to prove $x = \textsc{lift}(x)$ inductively via the transitivity of $=$. To `if`-lift a clause, we apply $\textsc{lift}$ to each literal; this reduction can be justified via the update clause rule.

Showing that $\textsc{lift}(x)$ terminates is somewhat involved. The main idea is to first define a measure, IDEPTH, as follows. The IDEPTH of any constant or variable is zero. For `if`-expressions

$$\textsc{idepth}((\texttt{if}\ \ a\ \ b\ \ c)) \triangleq 1 + \max\{\textsc{idepth}(a), \textsc{idepth}(b), \textsc{idepth}(c)\},$$

and for other functions applications or lambda abbreviations

$$\textsc{idepth}(x) \triangleq \max\{\textsc{idepth}(t_i)\},$$

where the t_i are the arguments or actuals. To admit LIFT, we show that whenever $\textsc{lift1}(x) \neq x$, then maximum IDEPTH of any unlifted subterm in $\textsc{lift1}(x)$ is less than that for x.

Like clause splitting, full `if`-lifting is sometimes takes too much time or produces proofs that are too large. It is straightforward to develop a version of LIFT1 which considers at most the first n terms from $\textsc{stests}(x)$, and similarly we can develop a restricted version of LIFT which applies this limited LIFT1 only until some limit is reached. This allows other techniques such as rewriting to reduce the intermediate terms before further lifting.

Chapter 8

Assumptions

Suppose $C = [t_1, \ldots, t_n]$ is a clause we are trying to prove. Our rewriter works literal by literal, walking through each t_i and using assumptions, evaluation, and user-supplied rewrite rules to produce a supposedly simpler term, $t_i{}'$. In this chapter, we explain how assumptions are made and used.

Where do assumptions come from? One source is the clause itself. As each literal is being rewritten, we can assume the other literals are false. That is, let $T_1, \ldots, T_n$ be the term formulas for $t_1, \ldots, t_n$, so the formula for C is $T_1 \vee \cdots \vee T_n$. Suppose we would like to rewrite t_n to $t_n{}'$. Now, our original goal is propositionally equivalent to $(\neg T_1 \wedge \cdots \wedge \neg T_{n-1}) \rightarrow t_n \neq \texttt{nil}$, and our simplified goal is equivalent to

$$(\neg T_1 \wedge \cdots \wedge \neg T_{n-1}) \rightarrow t_n{}' \neq \texttt{nil}.$$

So, if we can prove this simplified goal and we can also establish

$$(\neg T_1 \wedge \cdots \wedge \neg T_{n-1}) \rightarrow \texttt{(iff } t_n \ t_n{}'\texttt{)} = \texttt{t},$$

then we can recover a proof of our original goal via the disjoined substitute iff into literal rule. In other words, the rewriter need not show t_n and $t_n{}'$ are always and exactly equal, but only that they are `iff`-equivalent when the other literals are assumed to be false.

Another source of assumptions is `if`-expressions. That is, when we are rewriting `(if` a b c`)`, we may assume a is true while we rewrite b, and that a is false while we rewrite c.

The fundamental operations of an assumptions system are to ASSUME new facts, to identify when CONTRADICTORY assumptions have been made, and to SIMPLIFY terms using the facts which have been assumed.

How are these operations used by our rewriter? Suppose we want to simplify some goal clause, $C = [t_1, \ldots, t_n]$, by rewriting the literal t_i. Before rewriting begins, we create an empty assumptions structure, then extend it by assuming `(not` t_j`)` for each $j \neq i$. If some contradiction is observed, then we can use the contradiction to immediately prove the clause and there will be no need to rewrite t_i. Otherwise, the rewriter begins simplifying t_i; as it works, it encounters various subterms which it asks the assumptions system to SIMPLIFY. Ideally, the assumptions system will know something about these terms, and will produce simpler replacement terms which will help the rewriter to make more progress.

Probably the simplest way to record assumptions would be to put them in a list: ASSUME could just cons the new assumption into the list, and SIMPLIFY could just reduce subterms to `t` when they are found in the list. But such a system would not be very helpful to the rewriter. For instance, suppose we knew the `subsetp` function was reflexive, which might be expressed as the rewrite rule "terms matching `(subsetp` x x`)` may be rewritten to `t`." Suppose further that we are attempting to rewrite `(subsetp` a b`)` after assuming `(equal` a b`)` is true. Now, since `(equal` a b`)` does not occur as a subterm of `(subsetp` a b`)`, the rewriter's requests to simplify subterms would always fail, and we would not see that the rule could be applied.

Instead, our assumptions system uses simple disjoined-set structures to track `equal`- and `iff`-equivalences which have been assumed and inferred. For instance, if we begin by assuming `(equal` a b`)`, we will create an equality set, $\{a, b\}$. If we then additionally assume `(equal` b c`)`, this set will be extended to $\{a, b, c\}$. This way, when our assumptions system is asked to simplify any of a, b, or c, it can produce a distinguished representative for this set, solving the problem described above.

It is easy to imagine more sophisticated assumptions systems that infer more information and employ other kinds of forward-direction reasoning. For instance, in the ACL2 theorem prover, an extensible type-reasoning algorithm keeps track of facts such as "a is a positive natural" or "b cannot be a cons," and these observations can be used to simplify terms such as `(natp` a`)` to `t` and `(car` b`)` to `nil`. ACL2 also builds tables of inequalities for arithmetic reasoning, and allows the user to provide forward-chaining rules that are used to make additional inferences before rewriting begins.

But forward-direction reasoning takes time and may not produce useful inferences, so the trick is to strike a good balance. Anecdotally, early in this project we decided to eliminate the forward-chaining rules from our ACL2 proofs; this was not difficult and led to an approximately $\frac{1}{3}$ improvement in the speed of some proofs. Of course, this is only one data point and we do not intend to claim that such rules are never useful. Our approach is probably overly minimalist, but we have found it to be adequate and not difficult to work with.

8.1 Term Ordering

A major function of our assumptions system is to simplify equivalent terms into some canonical form. We choose the distinguished member for each equivalence set based upon whichever term is simplest according to an ordering on terms.

To begin with, we introduce a total order over all objects, which we name `<<` after a similar function that Manolios and Kaufmann [61] implemented for ACL2. Since we already have an ordering of the natural numbers, `<`, and an ordering of the symbols, `symbol-<`, to implement `<<` we simply say naturals come before symbols,

symbols come before conses, and conses are ordered first by their `car`s, then by their `cdr`s.

While straightforward, `<<` is not a very good ordering for choosing "simple" terms. For instance, constants intuitively seem to be simpler than variables, yet `<<` reverses this, e.g., `0` is larger than `x` since we represent `x` as a symbol and represent `0` as `(quote . (0 . nil))`. Lexicographic ordering can also lead to oddities such as `(f a (f b c))` being considered simpler than `(f b c)`.

Our term order, `logic.term-<`, is taken with only slight modification from ACL2's function `term-order`. First, we say terms with fewer variables are smaller. When the number of variables is the same, we say terms with fewer total function applications and lambda abbreviations are smaller. When this, too, is the same, we count up the total SIZE of all constants in the term, where

$$\textsc{size}(x) = \begin{cases} 1 & \text{if } x \text{ is a symbol} \\ x & \text{if } x \text{ is a natural number, and} \\ 1 + \textsc{size}(a) + \textsc{size}(b) & \text{if } x \text{ is } (a\ .\ b), \end{cases}$$

and say that terms with a lower total size are smaller. Finally, if even this is the same, we just use `<<` to determine which term is smaller. In practice, `logic.term-<` seems to work well.

8.2 Hypboxes

Our assumptions system stores two kinds of information: (1) the actual terms which have been assumed, which we will call the *hyps*, and (2) the sets of equivalences which have been inferred from these hyps. Roughly speaking, when our assumptions system claims some term, a, is equivalent to a simpler term, b, we will need to be able to prove $\mathit{hyps} \to (\mathit{equiv}\ \ a\ \ b) = \mathtt{t}$, where *equiv* is either `equal` or `iff`.

How are the hyps recorded? For simplicity, we would have preferred to use a single list of the terms we had assumed, and that is how we started out. But later, when we were using the fully expansive version of our prover to rewrite clauses, we found that this led proofs to become too large. (See Section 9.11 for details). This led us to adopt a slightly more complex structure, which we call a *hypbox*. Each hypbox contains two lists of terms, called *left* and *right*, which are together used to store the assumptions. Instead of a single ASSUME operation, we have ASSUME-LEFT and ASSUME-RIGHT, which add the new assumption to the suggested list. When we are rewriting t_i in the clause $[t_1, \ldots t_n]$, we ASSUME-LEFT $t_1, \ldots, t_{i-1}$ and ASSUME-RIGHT $t_{i+1}, \ldots, t_n$. This separation allows us to stitch in the new $t_i{}'$ in far fewer proof steps.

It is also convenient to store the negation of each assumption, rather than the assumption itself. After all, before we begin rewriting t_i, we need to assume $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ are all false, so the negated assumptions are readily available in the clause. Further, when we are interested in using assumptions to simplify a term, we want to establish, e.g., $hyps \rightarrow (equiv\ \ a\ \ b) = \mathtt{t}$, but this is really $\neg hyp_1 \vee \cdots \vee \neg hyp_n \vee (equiv\ \ a\ \ b) = \mathtt{t}$.

Given a non-empty hypbox, we define the *hypbox formula* as follows. Let the negated hypotheses in the left list be $[l_1, \ldots, l_n]$ and in the right list be $[r_1, \ldots, r_m]$. Furthermore, let $L_1, \ldots, L_n$ be the term formulas for the l_i, and let $R_1, \ldots, R_m$ similarly be the term formulas for the r_i. Then, the hypbox formula is $L_{1 \ldots n} \vee R_{1 \ldots m}$. To justify our assumptions system, our proof obligation is to show that whenever it is used to simplify some term a to b, then $P \vee (equiv\ \ a\ \ b) = \mathtt{t}$ is provable,where P is the hypbox formula.

8.3 Equivalence Traces

We implement two versions of our assumptions system, one which is "slow" and one which is "fast." The main difference between the two is how the sets of equivalent terms are represented. The slow version remembers how its inferences were made, and this information can be used to justify any claim it makes with a fully expansive proof. The fast version omits this information, which makes it somewhat more efficient, but as a result its claims cannot be directly justified. To justify the fast version, we will prove (Section 8.6) it only makes the same claims as the slow version.

But for now, we turn our attention to the slow assumptions system. Here, each equivalence we have inferred is represented as an *equivalence trace*. Each equivalence trace is an aggregate of five components,

- *method*, a symbol describing what kind of trace this is,
- *iffp*, a flag indicating if the equivalence is `iff` or `equal`,
- *lhs*, a term, called the left-hand side,
- *rhs*, a term, called the right-hand side, which must be larger than the lhs according to the term order, and
- *subtraces*, which are recursively a list of any equivalence traces needed to justify this trace (e.g., for transitivity).

We say the *conclusion* of an equivalence trace is the term `(`*equiv lhs rhs*`)`, where *equiv* is either `equal` or `iff`, depending upon the value of iffp. We think of each trace as an assertion that this conclusion holds when the assumption system's hypotheses are satisfied. That is, each well-formed trace claims that the formula $P \vee (\mathit{equiv}\ \ \mathit{lhs}\ \ \mathit{rhs}) = \mathtt{t}$ is provable, where P is the hypbox formula. It is sometimes useful to ignore the details of the term order, so when we say that a trace *equates* a and b, what we mean is that its lhs is the smaller and its rhs is the greater of these terms.

When a new hypothesis, *hyp*, is added to our assumptions system via ASSUME-LEFT or ASSUME-RIGHT, we construct equivalence traces that (1) capture the direct meaning of the new hypothesis, and (2) connect these new traces with any previously constructed traces. For example, suppose we have previously assumed the hypothesis `(equal` *a* *b*`)`, and are now assuming `(equal` *b* *c*`)`. We begin by constructing a trace that captures the meaning of this new hypothesis, i.e., which equates *b* and *c*. We then combine this new trace with the previously constructed traces to build a new trace that equates *a* with *c*.

To capture the direct meaning of a new assumption, *hyp*, we attempt to construct four kinds of equivalence traces.

- *Primary* equivalence traces. If *hyp* is of the form `(equal` *a* *b*`)`, we can usually construct a Primary equivalence trace which equates *a* and *b*. The exception is when *a* and *b* are the same term: the term order requirement does not allow a trace to have the same lhs and rhs. The method of a well-formed Primary equivalence trace is the symbol PRIMARY, there are no subtraces, and the iffp flag is NIL since this trace represents an equality. The lhs and rhs of the trace are respectively the smaller and larger of *a* and *b*, per the term order.

- *Secondary* equivalence traces. If *hyp* has the form `(not` *a*`)`, we can usually construct a Secondary equivalence trace that equates *a* with `nil`. The exception is when *a* happens to be `nil`, in which case we again cannot construct the trace due to the term order. The method of a well-formed Secondary equivalence trace is the symbol SECONDARY. There are no subtraces, iffp is NIL, and the lhs and rhs of the trace are respectively the smaller and larger *a* and `nil`, per the term order.

– *Direct Iff* equivalence traces. If *hyp* has the form `(iff` *a* *b*`)`, we can usually construct a Direct Iff equivalence trace that captures the Boolean equivalence of *a* and *b*. The exception is when *a* and *b* are the same term. The method of a well-formed Direct Iff equivalence trace is DIRECT-IFF. There are no subtraces, iffp is T, and the lhs and rhs are the smaller and larger of *a* and *b*.

– *Negative Iff* equivalence traces. We can usually construct a Negative Iff equivalence trace that shows *hyp* is non-`nil`. The method of a well-formed Negative Iff equivalence trace is NEGATIVE-IFF. There are no subtraces, iffp is T, and the lhs and rhs are the smaller and larger of *hyp* and `t`. In practice, we do not construct Negative Iff equivalence traces when *hyp* is any non-nil constant, since they are not useful assumptions.

If we can generate a Primary or Secondary equivalence trace from *hyp*, then we can also construct a "weaker" version of this trace. For x to be a well-formed *Weakening* equivalence trace, its method must be WEAKEN and it must have exactly one subtrace, say y. The `iff` of x must be T while the iffp of y is NIL, and the lhs and rhs of x must be the lhs and rhs from y, respectively. In other words, a Weakening trace allows us to conclude `(iff` *lhs* *rhs*`)` from another trace which concludes `(equal` *lhs* *rhs*`)`.

The above traces allow us to capture the meaning of the new hypothesis, but we would also like to construct traces which connect the new hypothesis with our previous assumptions. That is, suppose we have previously constructed a trace which concludes `(equal` *a* *b*`)`. If, when we assume a new hypothesis, we construct a Primary equivalence trace that concludes `(equal` *b* *c*`)`, we would like to additionally create a new equivalence trace that establishes `(equal` *a* *c*`)`.

Toward this end, we have three kinds of *Transitivity* equivalence traces that may be used to combine compatible equivalence traces. Written in an inference-rule style, these traces might be described as follows.

$$\begin{array}{ccc} \textit{Trans1} & \textit{Trans2} & \textit{Trans3} \\ \begin{array}{c} (\textit{equiv}_1\ \ a\ \ b) \\ (\textit{equiv}_1\ \ b\ \ c) \\ \hline (\textit{equiv}_1\ \ a\ \ c) \end{array} & \begin{array}{c} (\textit{equiv}_1\ \ a\ \ b) \\ (\textit{equiv}_1\ \ a\ \ c) \\ \hline (\textit{equiv}_1\ \ b\ \ c) \end{array} & \begin{array}{c} (\textit{equiv}_1\ \ a\ \ c) \\ (\textit{equiv}_1\ \ b\ \ c) \\ \hline (\textit{equiv}_1\ \ a\ \ b) \end{array} \end{array}$$

More precisely, for x to be a well-formed Trans1 equivalence trace, its method must be TRANS1, and it must have two subtraces, say y and z. The iffp fields of x, y, and z must all agree. The lhs of x must be the lhs of y; the rhs of x must be the rhs of z, and the rhs of y must be the lhs of z. The requirements for the other transitivity traces are analogous.

Equivalence traces allow us to justify the inferences made by our assumptions system. Each trace serves as a high-level sketch that explains how its formula can be proven. It is straightforward to prove the formula for any Primary, Secondary, Direct Iff, or Negative Iff trace. As an example, we now show how this may be done for Primary equivalence traces.

Suppose a and b are different terms and we are given `(equal` a b`)` as a hypothesis. This means that `(not (equal` a b`))` is among the left or right terms of the hypbox. We will show that $P \vee$ `(equal` lhs rhs`)` $=$ `t` is provable, where P is the hypbox formula, lhs is the lesser of a and b according to the term order, and rhs is the greater of a and b. As a lemma, the following theorem about `not` is helpful.

Formal Theorem 45. Not when nil

$$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} = \texttt{t}$$

Proof.

$\texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$ Axiom if when nil

$\texttt{x} \neq \texttt{nil} \vee \texttt{(if x nil t)} = \texttt{t}$	Instantiation	(*1)
$\texttt{(not x)} = \texttt{(if x nil t)}$	Definition of not	
$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} = \texttt{(if x nil t)}$	Expansion	
$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} = \texttt{t}$	Dj. trans. = *1	$\square$

The main part of the proof is to show that either the term in the hypbox is true, or that the conclusion of the trace holds. In the case of Primary equivalence traces, we may carry out this derivation as follows.

$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} = \texttt{t}$	Th. not when nil
$\texttt{(equal } a\ b\texttt{)} \neq \texttt{nil} \vee \texttt{(not (equal } a\ b\texttt{))} = \texttt{t}$	Instantiation
$\texttt{(equal } a\ b\texttt{)} \neq \texttt{nil} \vee \texttt{(not (equal } a\ b\texttt{))} \neq \texttt{nil}$	Dj. not nil from t
$\texttt{(not (equal } a\ b\texttt{))} \neq \texttt{nil} \vee \texttt{(equal } a\ b\texttt{)} \neq \texttt{nil}$	Commute or
$\texttt{(not (equal } a\ b\texttt{))} \neq \texttt{nil} \vee \texttt{(equal } a\ b\texttt{)} = \texttt{t}$	Dj. equal t from not nil
$\texttt{(not (equal } a\ b\texttt{))} \neq \texttt{nil} \vee \texttt{(equal } \mathit{lhs}\ \mathit{rhs}\texttt{)} = \texttt{t}$	[*] See below

[*] This last step depends upon the order of a and b: if a is smaller than b in the term order, we can just reuse the proof from the previous line; otherwise, *lhs* is b and *rhs* is a, and we can derive the following formula via disjoined commute equal.

Now, to finish the proof, we just need to expand the above with our other hypotheses. Recall that (not (equal a b)) is either in the left or right side of the hypbox. If the other side is empty, then by multi assoc expansion we can obtain our goal. Otherwise, if the hypothesis is in the left side and the right side is non-empty, we have:

$L_{1\dots n} \vee \texttt{(equal } \mathit{lhs}\ \mathit{rhs}\texttt{)} = \texttt{t}$	Multi-assoc expansion
$P \vee \texttt{(equal } \mathit{lhs}\ \mathit{rhs}\texttt{)} = \texttt{t}$	Disjoined left expansion

Otherwise, the hypothesis is on the right side and the left side is non-empty, and we have:

$R_{1\dots m} \vee \texttt{(equal } \mathit{lhs}\ \mathit{rhs}\texttt{)} = \texttt{t}$	Multi-assoc expansion
$L_{1\dots n} \vee R_{1\dots m} \vee \texttt{(equal } \mathit{lhs}\ \mathit{rhs}\texttt{)} = \texttt{t}$	Expansion
$P \vee \texttt{(equal } \mathit{lhs}\ \mathit{rhs}\texttt{)} = \texttt{t}$	Associativity

In the case of Trans1, Trans2, Trans3, and Weakening traces, it is quite straightforward to derive the formula for the trace if we are given proofs of the formulas for the subtraces, using our rules about `iff` and `equal`. Hence, by induction, we may derive the formula for any well-formed equivalence trace. We introduce a function, the "trace compiler," that can construct the fully expansive proof for any valid trace.

Traces are a useful abstraction which separate the process of making assumptions from justifying them. That is, when we implement the assumptions system, we only need to ensure that all of the traces we create are valid. This allows us to work at the level of traces instead of proofs.

We can also use traces to prove clauses which contain contradictory assumptions. We say an equivalence trace is *contradictory* when it concludes

- $\texttt{(equal}\ c_1\ c_2\texttt{)}$, where c_1 and c_2 are distinct constants,
- $\texttt{(equal}\ a\ \texttt{(not}\ a\texttt{))}$,
- `(iff nil t)`, which per the term order also addresses `(iff t nil)`, or
- $\texttt{(iff}\ a\ \texttt{(not}\ a\texttt{))}$.

Given a contradictory equivalence trace, we can prove the hypbox formula. For instance, let P be the hypbox formula and suppose we have a contradictory equivalence trace of the form $\texttt{(equal}\ c_1\ c_2\texttt{)}$, where c_1 and c_2 are distinct constants. To begin, we may prove the formula for this trace, namely $P \vee \texttt{(equal}\ c_1\ c_2\texttt{)} = \texttt{t}$, using the trace compiler. Now, to derive P,

$P \vee \texttt{(equal}\ c_1\ c_2\texttt{)} = \texttt{t}$	Trace compiler	
$\texttt{(equal}\ c_1\ c_2\texttt{)} = \texttt{t} \vee P$	Commute or	(*1)
$\texttt{(equal}\ c_1\ c_2\texttt{)} = \texttt{nil}$	Base evaluation	
$\texttt{(equal}\ c_1\ c_2\texttt{)} \neq \texttt{t}$	Not t from nil	
P	Modus ponens 2, *1	

For the other kinds of contradictory equivalence traces, the approach is similar except for details of showing that the conclusion is false, viz. the base evaluation step, above.

8.4 Equivalence Databases

We organize our equivalence traces into two simple disjoined-set (a.k.a. union-find) structures—one for `equal`-equivalences, and one for `iff`-equivalences. An *equivalence set* is an aggregate of three components,

- *iffp*, a flag indicating the equivalence for this set,
- *head*, the term that is the distinguished member of this set, and
- *tail*, a list of iffp-compatible equivalence traces, whose every lhs is the same as head, and whose rhses are distinct.

We think of equivalence sets as sets of terms: the terms in the set are its head, and also the rhs of every trace in its tail. By the term order, the terms in each equivalence set are unique.

Equivalence sets are grouped into *equivalence databases*, which are aggregates of three components,

- *equalsets*, a list of `equal` equivalence sets,
- *iffsets*, a list of `iff` equivalence sets, and
- *contradiction*, which is `nil`, or a contradictory equivalence trace,

where the terms in the equalsets are mutually disjoint (i.e., no term occurs in more than one equalset), and similarly the terms in the iffsets are mutually disjoint.

Using such a database is straightforward. Our USE operation takes as inputs x, the term the rewriter would like to have simplified; *iffp*, the equivalence that must be preserved; and *db*, the database to use. We look through whichever of equalsets or iffsets is appropriate for *iffp*. If we can find x in the tail of some set, we return the trace which establishes (*equiv* *h* *x*), where h is the head of the set. The rewriter, then, will replace x with h, which is the "simplest" term we know is equivalent to x. If every trace in the database is valid, the returned trace is also valid and can be compiled into a proof of its claim.

Constructing these databases is more involved. The initial equivalence database has no equalsets, no iffsets, and no contradiction. Each time we assume a new hypothesis, we construct the various equivalence traces that are appropriate for the assumption and add them to the database. Adding these traces may also lead us to infer new transitivity and weakening traces, which we also add. This can get somewhat tricky because of all the invariants being maintained, so we make use of some utility routines.

1. UPDATE-HEAD(t, x).

Given an equivalence set x whose head is h, and an iffp-compatible equivalence trace, t, that concludes (*equiv* *lhs* *h*), UPDATE-HEAD creates a new set, x', that has all of the terms in x and has *lhs* as its new head. Note that by the term order, *lhs* is smaller than h.

2. MAYBE-EXTEND(t, x).

Given an equivalence set x whose head is h, and an iffp-compatible equivalence trace, t, that concludes (*equiv* *lhs* *rhs*), MAYBE-EXTEND produces a new set, x', which adds the information in t to x if it makes sense to do so. That is, if both *lhs* and *rhs* are in x, then their equivalence is "already known" by the set and x' is simply

x. If neither term is found, then this equivalence is "not relevant" to the set, and again x' is simply x. The interesting case is when exactly one of *lhs* or *rhs* is found, and here we add the missing term to x'.

3. JOIN-SETS(t, x, y).

Given two mutually disjoint, iffp-compatible equivalence sets, x and y, and an iffp-compatible equivalence trace, t, that concludes `(`*equiv lhs rhs*`)`, where *lhs* is in x and *rhs* is in y, JOIN-SETS produces a new equivalence set, n, which contains all of the members of x and y.

4. EXTEND-SETS(t, x).

Given a list of iffp-compatible, mutually disjoint equivalence sets, x, and an iffp-compatible trace, t, that concludes `(`*equiv lhs rhs*`)`, EXTEND-SETS creates a new list of equivalence sets, x', which extend x with the information in t.

5. EXTEND-DB(*nhyp*, *db*, *primaryp*, *secondaryp*, *directp*, *negativep*).

Given a negated hypothesis, *nhyp*, and an equivalence database, *db*, and four flags *primaryp*, *secondaryp*, *directp*, *negativep* which control the kinds of traces to create, EXTEND-DB creates a new database by adding the traces which can be inferred from this hypothesis.

The flags allow us to control the kinds of inferences that are made from the hypothesis. For instance, if *primaryp* is `nil`, then we will not try to create any primary equivalence traces from *nhyp*.

If the flags allow it, we first attempt to construct a primary and a secondary trace from *nhyp*. When this is successful, we use EXTEND-SETS to add the new traces to the equalsets; we also use weakening to produce `iff`-based versions of these traces which we add to the iffsets, again via EXTEND-SETS. Next, we attempt to construct direct iff and negative iff traces from *nhyp*, and add them to the iffsets. Finally, when no contradiction has been found, we may sweep through the new equivalence sets, looking for contradictory traces, and appropriately update the database's contradiction field.

Constructing an equivalence database is not particularly cheap. The EXTEND-DB operation is linear in the total size of the equalsets and iffsets, but since it must be called for each assumption the overall cost of constructing the database is quadratic. This has been adequate for our project, but should probably be improved upon. In particular, it seems wasteful to scan for contradictions every time a new assumption is added, and it should be straightforward to defer that until all clause assumptions have been added. It is probably also unnecessary to replicate primary and secondary traces in both the equalsets and iffsets, and this might be avoided through a more advanced USE operation.

8.5 Assumptions Structures

We group the hypbox and the corresponding equivalence database into an *assumptions structure* for the rewriter to interface with. Each assumptions structure is an aggregate of the following components:

- *hypbox*, the hypbox for the actual terms that have been assumed,
- *eqdatabase*, the equivalence database we have constructed,
- *ctrl*, an assumptions control structure, and
- *trueterms*, a list of terms used heuristically in free-variable matching,

where every trace in the eqdatabase is valid with respect to the hypbox.

We have already covered hypboxes and equivalence databases. Assumptions control structures are simple aggregates of the *primaryp*, *secondaryp*, *directp*, and *negativep* flags which are passed to EXTEND-DB. We had hoped this level of control would occasionally be useful as a way to reduce the amount of time needed to make assumptions, but in practice it seems that our rewriter is not very effective without all four kinds of assumptions, so we rarely use this feature. This kind of control might be more useful in a richer, ACL2-like assumptions system as a way to sometimes disable type reasoning, arithmetic reasoning, or forward-chaining rules.

The trueterms are a list of the terms which we have inferred to be non-`nil`. When our rewriter tries to use a conditional rewrite rule whose hypotheses contain free variables, this list of terms is used heuristically to try to identify terms that might satisfy the hypotheses. This is only incidental to the rest of the assumptions system, and is covered in Section 9.5.

Bringing it all together, suppose we are using the fully expansive version of our rewriter and we would like to simplify the clause $[t_1, \ldots, t_n]$ by rewriting the literal t_i. Before rewriting begins, we use the function `rw.empty-assms` to construct an initial assumptions structure which has an empty hypbox, an empty eqdatabase, no trueterms, and a ctrl structure which is supplied (usually implicitly) by the user. A trivial but important fact is that given any valid ctrl structure (recognized by `rw.assmctrlp`), this function produces a valid assumptions structure (recognized by `rw.assmsp`). In our ACL2 proof sketch, this is expressed as follows.

ACL2 Code
```
(defthm rw.assmsp-of-rw.empty-assms
  (implies (rw.assmctrlp ctrl)
           (rw.assmsp (rw.empty-assms ctrl))))
```

We then add the negations of the literals to the left of t_i via ASSUME-LEFT, and similarly we add the negations of the literals to the right via ASSUME-RIGHT. Both ASSUME-LEFT and ASSUME-RIGHT take, as inputs, the new negated hypothesis to add and the current assumptions structure. They return a new assumptions structure, where the hypbox is extended with the hypothesis (either on the left or right, as appropriate), the equivalence database is updated via EXTEND-DB, the ctrl remains the same, and the trueterms are updated to agree with the equivalence database.

For correctness, our goal is to show that ASSUME-LEFT and ASSUME-RIGHT produce valid assumptions structures when given a valid assumptions structure as input. One lemma toward this is that if we are given any equivalence trace which is valid in some hypbox, then the trace is still valid in any extended hypbox. Another key lemma is that if all of the traces in the database are valid with respect to a hypbox, and the new *nhyp* being added to the database via EXTEND-DB is among the left or right terms in the hypbox, then all of the traces in the resulting, extended database are also valid. In our ACL2 proof sketch, these theorems are as follows.

ACL2 Code
```
(defthm rw.assmsp-of-rw.assume-left
  (implies (and (logic.termp nhyp)
                (rw.assmsp assms))
           (rw.assmsp (rw.assume-left nhyp assms))))

(defthm rw.assmsp-of-rw.assume-right
  (implies (and (logic.termp nhyp)
                (rw.assmsp assms))
           (rw.assmsp (rw.assume-right nhyp assms))))
```

Our SIMPLIFY operation is named `rw.try-assms`, and takes three arguments: the assumptions structure to use, the term to simplify, and a flag indicating whether equal- or iff-equivalence should be maintained. It either returns the simplified version of the term, or nil when no simplification is possible.

To justify the use of rw.try-assms, we introduce rw.try-assms-bldr, which takes the same arguments. When rw.try-assms is successful, rw.try-assms-bldr can produce a proof that $P \vee (\mathit{equiv}\ \ \mathit{term}\ \ \mathit{term}') = \mathtt{t}$, where P is the formula for the hypbox. To construct this proof, we simply compile the trace found in the database and then commute the equivalence.

Recall from page 154 that when we introduce proof-building functions, we prove that it is (1) "well-typed"—it is a valid appeal, (2) "relevant"—it has the desired conclusion, and (3) "faithful"—it is accepted by logic.proofp. Below, we show the ACL2 statements of these theorems for rw.try-assms-bldr.

ACL2 Code
```
(defthm logic.appealp-of-rw.try-assms-bldr
  (implies (and (rw.try-assms assms term iffp)
                (logic.termp term)
                (rw.assmsp assms))
           (logic.appealp (rw.try-assms-bldr assms term iffp))))

(defthm logic.conclusion-of-rw.try-assms-bldr
  (implies
   (and (rw.try-assms assms term iffp)
        (logic.termp term)
        (rw.assmsp assms))
   (equal (logic.conclusion (rw.try-assms-bldr assms term iffp))
          (logic.por
           (rw.hypbox-formula (rw.assms->hypbox assms))
           (logic.pequal (logic.function
                          (if iffp 'iff 'equal)
                          (list term
                                (rw.try-assms assms term iffp)))
                         ''t)))))
```

```
(defthm logic.proofp-of-rw.try-assms-bldr
  (implies (and (rw.try-assms assms term iffp)
                (logic.termp term)
                (rw.assmsp assms)
                (rw.assms-atblp assms atbl)
                (logic.term-atblp term atbl)
                (equal (cdr (lookup 'not atbl)) 1)
                (equal (cdr (lookup 'iff atbl)) 2)
                (equal (cdr (lookup 'equal atbl)) 2)
                ... various formulas are thms ...
                ... various formulas are axioms ... )
           (logic.proofp (rw.try-assms-bldr assms term iffp)
                         axioms thms atbl)))
```

In the faithfulness theorem, the predicate `rw.assms-atblp` is used to ensure that the terms throughout the traces are valid with respect to the arity table. The “various formulas” which we omit are the simple axioms and theorems such as the reflexivity and commutativity of equal that are needed by the builder functions that `rw.try-assms-bldr` calls upon, as described in Section 6.1.

Together, these lemmas establish that it is legitimate to use `rw.try-assms` to simplify terms as long as we know we are using a valid assumptions structure. As we will see in the next chapter, our fully expansive rewriter begins with an empty assumptions structure (which is valid) and extends it only via `rw.assume-left` and `rw.assume-right` (which are validity preserving). Accordingly, it can justify its uses of `rw.try-assms` by calling upon `rw.try-assms-bldr`.

8.6 Fast Assumptions

Constructing an equivalence trace involves consing its components together into a structure. This overhead may not seem too bad since only four conses are required to construct each trace. But we may need to build many equivalence traces to construct an assumptions system, and we construct an assumptions system before rewriting each literal in the clause, so this cost multiplies. To avoid this overhead, we develop and verify a "fast" version of our assumptions system which does away with these traces.

Recall that an ordinary, "slow" equivalence set is an aggregate of three components, iffp, head, and tail, where the head is a term and the tail is a list of equivalence traces. A *fast equivalence set* is also such an aggregate, except that the tail is just a list of terms. The basic idea is that we only want to record the terms which would have been in the rhs of each trace.

Similarly, recall that an ordinary equivalence database is an aggregate of equalsets, iffsets, and contradiction, where the equalsets and iffsets are lists of equivalence sets, and contradiction is either `nil` or a contradictory equivalence trace. A *fast equivalence database* is also such an aggregate, except that fast equivalence sets are used and the contradiction field is a Boolean flag that indicates whether a contradiction has been identified.

Finally, an ordinary assumptions structure is an aggregate of a hypbox, eqdatabase, ctrl, and trueterms. A *fast assumptions structure* contains the same components, except that the eqdatabase is a fast equivalence database.

To implement our fast assumptions system, we provide new analogues of our various routines for extending equivalence sets and databases that operate on these stripped down structures. To verify the fast version, our basic approach is to show these routines mirror the operation of the slow versions, which we have already justified.

We begin by developing three imaging functions to relate our fast and slow structures. Given a slow equivalence set, x, SET-IMAGE(x) creates the corresponding fast equivalence set: the iffp and head of SET-IMAGE(x) are the same as the iffp and head for x, and the tail of SET-IMAGE(x) is the list of the rhses of the tail of x. Given a slow equivalence database, db, DB-IMAGE(db), creates the corresponding fast equivalence database: the equalsets and iffsets of DB-IMAGE(db) are formed by taking the SET-IMAGE of the equalsets and iffsets for db, and the contradiction of DB-IMAGE(db) is `t` when db has a contradiction, and `nil` otherwise. Finally, given a slow assumptions structure, x, ASSM-IMAGE(x) creates the corresponding fast assumptions structure. The eqdatabase of ASSM-IMAGE(x) is the DB-IMAGE of the eqdatabase of x, while the hypbox, ctrl, and trueterms of ASSM-IMAGE(x) are just copied from x.

Next, we show that the fast version of each operation is "correct" with respect to this imaging function. Since we write the fast operations by copying the slow versions and making the appropriate updates, these proofs are quite straightforward.

1. FAST-UPDATE-HEAD(h, x).

Given a fast equivalence set x and a new term, h, which is smaller than the head of x, FAST-UPDATE-HEAD creates a new equivalence set, n, whose head is h and whose tail is formed by adding the head of x into its tail. Given valid inputs,

$$\begin{aligned}&\textsc{set-image}(\textsc{update-head}(t, x)) = \\ &\qquad \textsc{fast-update-head}(\textsc{lhs}(t), \textsc{set-image}(x)).\end{aligned}$$

2. FAST-MAYBE-EXTEND(l, r, x).

Given a fast equivalence set, x, and the terms l and r, FAST-MAYBE-EXTEND creates a new fast equivalence set which reflects the equivalence of l and r if this fact is relevant to x, as in MAYBE-EXTEND. Given valid inputs,

$$\begin{aligned}&\textsc{set-image}(\textsc{maybe-extend}(t, x)) = \\ &\qquad \textsc{fast-maybe-extend}(\textsc{lhs}(t), \textsc{rhs}(t), \textsc{set-image}(x)).\end{aligned}$$

3. FAST-JOIN-SETS(x, y).

Given two fast equivalence sets, x and y, FAST-JOIN-SETS creates a new fast equivalence set containing all of the terms in x and y and with the appropriate head. Given valid inputs,

$$\textsc{set-image}(\textsc{join-sets}(t, x, y)) = \\ \textsc{fast-join-sets}(\textsc{set-image}(x), \textsc{set-image}(y)).$$

4. FAST-EXTEND-SETS(l, r, iffp, x).

Given a list of fast equivalence sets, x, the terms l and r, and the Boolean flag *iffp*, FAST-EXTEND-SETS produces a new list of fast equivalence sets where the equivalence of l and r is known. The *iffp* flag is needed in case neither l nor r are present, since in this case we need to create a new fast equivalence set relating l and r, and we need to know what to use for the iffp of this new set. Given valid inputs,

$$\textsc{set-list-image}(\textsc{extend-sets}(t, x)) = \\ \textsc{fast-extend-sets}(\textsc{lhs}(t), \textsc{rhs}(t), \textsc{iffp}(t), \textsc{set-list-image}(x)),$$

where SET-LIST-IMAGE(x) just applies SET-IMAGE to every member of a list of equivalence sets.

5. FAST-EXTEND-DB$(\mathit{nhyp}, \mathit{db}, \mathit{primaryp}, \mathit{secondaryp}, \mathit{directp}, \mathit{negativep})$.

Given a fast equivalence database, *db*, a negated hypothesis *nhyp*, and the various flags as in EXTEND-DB, FAST-EXTEND-DB creates a new fast equivalence database which extends *db* with the assumptions which can be made from this hypothesis. Given valid inputs,

$$\textsc{db-image}(\textsc{extend-db}(\mathit{nhyp}, \mathit{db}, \mathit{primaryp}, \mathit{secondaryp}, \mathit{directp}, \mathit{negativep})) = \\ \textsc{fast-extend-db}(\mathit{nhyp}, \textsc{db-image}(\mathit{db}), \mathit{primaryp}, \mathit{secondaryp}, \mathit{directp}, \mathit{negativep}).$$

6. $\textsc{fast-empty-assms}(\mathit{ctrl})$.

Given an assumptions control structure, *ctrl*, FAST-EMPTY-ASSMS creates an empty fast assumptions structure, and we have

$$\textsc{assm-image}(\textsc{empty-assms}(\mathit{ctrl})) = \textsc{fast-empty-assms}(\mathit{ctrl}).$$

7. $\textsc{fast-assume-}\{\textsc{left},\textsc{right}\}(\mathit{nhyp}, \mathit{assms})$.

Given a negated hypothesis, *nhyp*, and a fast assumptions system, *assms*, FAST-ASSUME-LEFT and FAST-ASSUME-RIGHT add the hypothesis to the appropriate side of the hypbox, extend the database by calling FAST-EXTEND-DB, and update the trueterms as appropriate. Given valid inputs,

$$\textsc{assm-image}(\textsc{assume-}\{\textsc{left},\textsc{right}\}(\mathit{nhyp}, \mathit{assms})) =$$
$$\textsc{fast-assume-}\{\textsc{left},\textsc{right}\}(\mathit{nhyp}, \textsc{assm-image}(\mathit{assms})).$$

8. $\textsc{fast-try-assms}(\mathit{assms}, \mathit{term}, \mathit{iffp})$.

Given a fast assumptions system, a term to simplify, and a flag indicating whether `equal`- or `iff`-equivalence should be maintained, FAST-TRY-ASSMS returns either a simplified term or `nil` to indicate failure. Given valid inputs,

$$\textsc{try-assms}(\mathit{assms}, \mathit{term}, \mathit{iffp}) =$$
$$\textsc{fast-try-assms}(\textsc{assm-image}(\mathit{assms}), \mathit{term}, \mathit{iffp}).$$

We have slow and fast versions of our rewriter which are analogous to our slow and fast assumptions systems. The slow rewriter uses the slow assumptions system and keeps other information necessary to produce fully expansive proofs to justify the rewriting it performs, while the fast rewriter uses the fast assumptions system and also cannot justify the other steps it takes. Accordingly, the correspondence of our fast and slow assumptions system is an important lemma in relating the fast and slow rewriters.

Chapter 9

Rewriting

Rewriting with lemmas is the driving force in our style of theorem proving. Our rewriter uses the assumptions system we developed in Chapter 8 for keeping track of the equivalences which are known, and also uses the evaluator we developed in Section 6.4 to simplify ground terms.

More interestingly, it can apply rewrite rules. Each rewrite rule has some hypotheses and a conclusion, typically an equality, which is used to direct the replacement of one term with another. For rewriting to be justified, each rewrite rule must correspond to some theorem. Also, the replacement may only be carried out after the hypotheses are shown to be true. Our rewriter uses backchaining (recursive rewriting) to attempt to relieve hypotheses. Through his choice of rules, the user can guide the rewriter and "train" it to reason about the functions in his domain.

Our rewriter is quite complicated. The reader is reminded that we ultimately justify its use, both with ACL2 and, more importantly, with the core program developed in Chapter 4.

9.1 Rewrite Traces

As in our assumptions system, we have two versions of our rewriter, one which is "slow" but can produce a trace that explains how the term was rewritten, and one which is "fast" but cannot justify its work. To justify the fast version, we will establish (Section 9.10) that it produces the same results as the slow version. But until then, we will focus on the slow rewriter.

To justify the use of our rewriter, our basic approach is quite similar to our slow assumptions system. When our rewriter simplifies some term x to x', it produces a *rewrite trace* that provides a high-level explanation of how the rewriting was performed. For instance, one step of a trace might say, "the subterm y was rewritten to y' by applying the rewrite rule r, whose hypotheses were relieved as described by the following subtraces." These traces allow us to address the justification of our rewriter in two phases: first we show how to prove the claim made by any well-formed trace, then we show the rewriter always produces well-formed traces.

This decoupling has allowed us to modify how our rewriter works and implement many of its features without substantially needing to reconsider its justification. We wish to stress this point. When implementing a rewriter, one must make a number of decisions about the order in which various simplifications are attempted. For instance, upon encountering a new term, should we first (1) recursively try to rewrite its subterms, (2) try to use the assumptions system, (3) try to evaluate it, or (4) try to apply some rewrite rules? It is difficult to tell which strategy will be most effective ahead of time, but regardless of which order is finally settled upon, the basic steps are the same. Traces allow us to justify each kind of step separately, so a large part of the overall proof can be done without any regard to the order in which rewriting steps are tried.

Traces also lead to certain efficiencies when building fully expansive proofs. In conditional rewriters, rules may only be applied when their hypotheses can be relieved, and relieving the hypotheses may recursively require a lot of rewriting. When we fail to apply some rewrite rule because we cannot relieve some hypothesis, this failed attempt will be left out of the final trace. Because of this, when we construct a proof from the trace, we only consider "useful" steps. This is much like Boulton's [10] technique of separating proof search from construction in LCF theorem provers.

Of course, the end-user of our system does not need to worry about such efficiencies since he can simply use our fast, verified rewriter, which does not build these traces. But when we are working to transform our Milawa proofs into a form that `logic.proofp` can check, we make extensive use of our slow rewriter, so its efficiency is important.

Rewrite traces are similar to equivalence traces. Each rewrite trace is an aggregate of the following components:

– *method*, a symbol explaining what kind of trace this is,

– *hypbox*, the assumptions the rewrite is occurring under,

– *lhs*, the term we rewrote (e.g., x),

– *rhs*, the term we produced (e.g., x'),

– *iffp*, indicating whether an `equal`- or `iff`-equivalence is maintained,

– *subtraces*, which are (recursively) a list of rewrite traces needed to justify this trace, and

– *extras*, which hold any additional information needed to justify this step.

Associated with each trace is a formula that conveys the logical meaning of the trace. This formula is built from the hypbox, iffp, lhs, and rhs of the given trace, and is called the *trace formula*. If the hypbox of the trace is empty, the trace formula is `(`*equiv lhs rhs*`)` $=$ `t`, where *equiv* is either `equal` or `iff`, per the iffp field of the trace. Otherwise, the trace formula is $P \vee$ `(`*equiv lhs rhs*`)` $=$ `t`, where P is the hypbox formula.

We denote a generic trace formula by $[\mathit{assms} \longrightarrow]\ \mathit{lhs} \equiv \mathit{rhs}$. When describing a trace where iffp must be `t`, we may write $[\mathit{assms} \longrightarrow]$ `(iff` *lhs* *rhs*`)`, and similarly when iffp must be `nil`, we may write $[\mathit{assms} \longrightarrow]$ `(equal` *lhs* *rhs*`)`. Often we will use a trace formula to suggest the traces we build, leaving the reader to fill in the rest of the trace by context, intuition, or clarifications in the surrounding text.

Each call of the slow rewriter must produce a trace, and this trace is built partly from the arguments to the rewriter and partly from the actions taken internally by the rewriter. Among the relevant arguments of the rewriter, used implicitly below in the description of the traces generated, are the term to be rewritten, x, the assumptions system, *assms*, and more specifically the hypbox for that assumptions system, and an iffp flag indicating which sense of equivalence is to be maintained by that call of the rewriter.

When the rewriter cannot (or, for whatever reason, does not) further simplify its input term, it may produce a Failure trace. We use an inference-rule style notation to describe the construction of traces. It should be understood that the method of the returned trace is a symbol indicating that this is a Failure trace, and that the subtraces and extras of the trace are empty.

Failure Trace

$$\frac{}{[\mathit{assms} \longrightarrow]\ x \equiv x}$$

When the rewriter encounters a ground term, it may attempt to evaluate it to produce a constant. If this succeeds, the rewriter may construct an Evaluation trace. Here, x' is constructed by (1) calling upon the evaluator from Section 6.4, then (2) canonicalizing the result to `t` or `nil` if we are only maintaining `iff`-equivalence. The stack depth used for evaluation is recorded in the *extras* of the trace.

Evaluation Trace
(Justified by evaluation)

$$\frac{}{[\mathit{assms} \longrightarrow]\ x \equiv x'}$$

One of the arguments to the rewriter is an assumptions structure, and the rewriter may use assumptions to simplify a term. When this is successful, we may construct an Assumptions trace. Here, x' is the result of simplifying x using the assumptions system. So that we may later justify this simplification, we record the equivalence trace we have used in the extras of the rewrite trace.

Assumptions Trace
(Justified by assumptions)

$$\frac{}{[\mathit{assms} \longrightarrow]\ \mathit{lhs} \equiv \mathit{rhs}}$$

Transitivity traces are the basic mechanism for combining rewrite steps. For instance, if we use assumptions to reduce x to x', then use evaluation to simplify x' to `t`, we can combine these steps with a Transitivity trace which concludes x is `t`. It should be understood that the traces shown above the line are the subtraces of the generated trace.

Transitivity Trace

$$\frac{\begin{array}{c}[\mathit{assms} \longrightarrow]\ x \equiv y \\ [\mathit{assms} \longrightarrow]\ y \equiv z\end{array}}{[\mathit{assms} \longrightarrow]\ x \equiv z}$$

To rewrite functions applications and lambda abbreviations, it is generally necessary to (recursively) simplify each argument. This may be done with an Equiv By Args or a Lambda Equiv By Args trace.

Equiv by Args Trace

$$\frac{\begin{array}{l}[\mathit{assms} \longrightarrow]\ a_1 = {a_1}' \\ \vdots \\ [\mathit{assms} \longrightarrow]\ a_n = {a_n}'\end{array}}{[\mathit{assms} \longrightarrow]\ \texttt{(}f\ \ a_1\ \ \ldots\ \ a_n\texttt{)} \equiv \texttt{(}f\ \ {a_1}'\ \ \ldots\ \ {a_n}'\texttt{)}}$$

Lambda Equiv by Args Trace

$$\cfrac{\begin{array}{l}[assms \longrightarrow]\ a_1 = a_1{}' \\ \vdots \\ [assms \longrightarrow]\ a_n = a_n{}'\end{array}}{[assms \longrightarrow]\ \texttt{((lambda (}x_{1\ldots n}\texttt{) }\beta\texttt{) }a_{1\ldots n}\texttt{)} \equiv \texttt{((lambda (}x_{1\ldots n}\texttt{) }\beta\texttt{) }a_{1\ldots n}{}'\texttt{)}}$$

Beta-Reduction traces allow us to expand away lambda abbreviations.

Beta-Reduction Trace

$$\cfrac{}{[assms \longrightarrow]\ \texttt{((lambda (}x_1\ \ldots\ x_n\texttt{) }\beta\texttt{) }t_1\ \ldots\ t_n\texttt{)} \equiv \beta/[x_{1\ldots n} \leftarrow t_{1\ldots n}]}$$

We also have a few traces to support special handling of `if`-expressions. One reason for this is that when we rewrite `(if` x y z`)`, we would additionally like to assume x while rewriting y, and to assume `(not` x`)` while rewriting z. This would not be possible using an equiv by args trace, which requires that the assumptions are the same across all arguments. Furthermore, it is generally useful to rewrite x while only maintaining `iff`-equivalence. Finally, we also prefer to rewrite `if` lazily—that is, we do not want to spend any time rewriting y when x rewrites to `nil`, and similarly we do not want to rewrite z when x rewrites to `t`.

If False Trace

$$\cfrac{\begin{array}{l}[assms \longrightarrow]\ \texttt{(iff }x_1\texttt{ nil)} \\ [assms \longrightarrow]\ z_1 \equiv z_2\end{array}}{[assms \longrightarrow]\ \texttt{(if }x_1\ y_1\ z_1\texttt{)} \equiv z_2}$$

If True Trace

$$\cfrac{\begin{array}{l}[assms \longrightarrow]\ \texttt{(iff }x_1\texttt{ t)} \\ [assms \longrightarrow]\ y_1 \equiv y_2\end{array}}{[assms \longrightarrow]\ \texttt{(if }x_1\ y_1\ z_1\texttt{)} \equiv y_2}$$

If General Trace

$$\cfrac{\begin{array}{l}[assms \longrightarrow]\ \texttt{(iff }x_1\ x_2\texttt{)} \\ x_2, assms \longrightarrow y_1 \equiv y_2 \\ \texttt{(not }x_2\texttt{)}, assms \longrightarrow z_1 \equiv z_2\end{array}}{[assms \longrightarrow]\ \texttt{(if }x_1\ y_1\ z_1\texttt{)} \equiv \texttt{(if }x_2\ y_2\ z_2\texttt{)}}$$

Our rewriter does not apply rewrite rules to `if`-expressions, so we provide a couple of additional traces which can be used to eliminate `(if` x y y`)` and to rewrite `(if` x `nil t)` into our preferred normal form, `(not` x`)`.

If Same Trace

$$\frac{\begin{array}{l}[\mathit{assms} \longrightarrow]\ \texttt{(iff}\ x_1\ x_2\texttt{)} \\ x_2, \mathit{assms} \longrightarrow y \equiv w \\ \texttt{(not}\ x_2\texttt{)}, \mathit{assms} \longrightarrow z \equiv w\end{array}}{[\mathit{assms} \longrightarrow]\ \texttt{(if}\ x_1\ y\ z\texttt{)} \equiv w}$$

If Not Trace

$$\frac{}{[\mathit{assms} \longrightarrow]\ \texttt{(if}\ x\ \texttt{nil t)} \equiv \texttt{(not}\ x\texttt{)}}$$

Our rewriter also has a special case for rewriting `not`-expressions. As with `if`-expressions, it is useful to know that only `iff`-equivalence needs to be maintained while rewriting the argument to `not`.

Not Congruence Trace

$$\frac{[\mathit{assms} \longrightarrow]\ \texttt{(iff}\ x\ x'\texttt{)}}{[\mathit{assms} \longrightarrow]\ \texttt{(not}\ x\texttt{)} \equiv \mathit{rhs},}$$

where

$$\mathit{rhs} = \begin{cases} \texttt{nil} & \text{if } x' \text{ is } \texttt{t}, \\ \texttt{t} & \text{if } x' \text{ is } \texttt{nil}, \text{ or} \\ \texttt{(not}\ x'\texttt{)} & \text{otherwise.} \end{cases}$$

There are also two more kinds of traces, rule application and forcing, which we will introduce later. Along with the above, these are the only well-formed traces. Much like equivalence traces, we can directly prove the formula for any well-formed atomic rewrite trace, and we can prove the formula for a compound trace if we are given proofs of the formulas for its subtraces. Hence, by induction, we can prove the formula for any well-formed trace. We implement a function, called the rewrite trace compiler, which does exactly this.

Justifying each kind of trace is routine. Failure traces can be proven via our various reflexivity rules and expansion. Evaluation traces are proven with the evaluation rule and our rules about `iff`. For Assumption traces, we just need to compile the equivalence trace as in `rw.try-assms-bldr`. Equiv By Args and Lambda Equiv By Args are trivial to justify using = by args or lambda = by args. Beta-Reduction traces follow from the beta reduction rule, and the traces for `if` and `not` are easy to derive using the rules and theorems we have already developed.

9.2 Controlling the Rewriter

The user guides the rewriter by selecting rules for it to apply. When we wish to discuss a particular rewrite rule, we will present it in an ACL2-like format. For example, the following rule says that terms matching `(car (cons x y))` should be unconditionally rewritten to `x`.

Rule:
```
(equal (car (cons x y))
       x)
```

More sophisticated rules are conditional, and only apply when certain hypotheses can be established. For instance, the following is a conditional rule which says that terms matching `(cons (car x) (cdr x))` should be rewritten to `x`, but only if we can first show that `(consp x)` holds.

Rule:
```
(implies (consp x)
         (equal (cons (car x) (cdr x))
                x))
```

We represent each rewrite rule as a simple aggregate of the following components,

- *name*, a symbol which is used to identify the rule,
- *type*, a description of what kind of rule this is (inside-out or outside-in),
- *hyps*, a list of hypotheses,
- *lhs*, a term which is the “target” of this rule,
- *rhs*, a term which is the “replacement” of this rule,
- *equiv*, the equivalence relation being maintained (e.g., `iff` or `equal`),
- *syntax*, syntactic restrictions on the rule’s application (Section 9.6), and
- *crithyps*, a list of “critical” terms for free-variable matching (Section 9.5).

How is a rule’s type used? Rewriting can be done inside-out or outside-in. In the former, to rewrite (f a_1 … a_n), we first simplify the terms on the “inside” by rewriting each a_i to $a_i{}'$, and afterward we move “out” and consider the rules that apply to (f $a_1{}'$ … $a_n{}'$). But our rewriter can also apply rules in an outside-in direction. Here, we begin by considering rules that could apply to the “outside,” (f a_1 … a_n). If some rule applies, it produces some new term which we then recursively attempt to simplify; otherwise we move “in” and begin simplifying the a_i, and afterwards we may consider the rules that apply to (f $a_1{}'$ … $a_n{}'$).

Both directions of rewriting have their strengths. Inside-out rewriting has the advantage that simplified $a_i{}'$ are more likely to be in a reduced form that will match with rewrite rules, while outside-in rewriting has the advantage that some subterms may be completely skipped when the rule eliminates a variable. For instance, consider rewriting the term `(car (cdr (cons` a `(cons` b c`))))`. If we apply a single pass of outside-in rewriting, we would reduce the term to `(car (cons` b' c'`))` without rewriting a. On the other hand, a single pass of inside-out rewriting would yield b', but in the process we would need to rewrite a.

The logical meaning of a rule is captured by its hyps, lhs, rhs, and equiv; the other fields are only heuristic annotations that influence how the rewriter will make use of the rule. We associate a clause with every rule,

$$[\texttt{(not}\ \ hyp_1\texttt{)}, \ldots, \texttt{(not}\ \ hyp_n\texttt{)}, \texttt{(}equiv\ \ lhs\ \ rhs\texttt{)}],$$

and we say the *rule formula* is the formula for this clause. Informally, the rule formula can be understood to mean $hyps \rightarrow \texttt{(}equiv\ \ lhs\ \ rhs\texttt{)}$.

In our tracing mechanism, rewrite rule applications are handled with Rule traces. The rule and substitution list being used are stored in the extras of the trace. For the trace to be well-formed, the rule formula must be among the theorems of the current history. Also, the equiv for the rule must be compatible with the iffp for the trace. That is, if the rule's equivalence is `equal`, then it may be used to construct traces where iffp is either `t` or `nil`. But if the rule's equivalence is `iff`, then it may only be used to construct traces where iffp is `t`.

Rule Trace
(Justified by a rewrite rule)

$$\begin{array}{l} [assms \rightarrow]\ \texttt{(iff}\ \ hyp_1/\sigma\ \ \texttt{t)} \\ \vdots \\ [assms \rightarrow]\ \texttt{(iff}\ \ hyp_n/\sigma\ \ \texttt{t)} \\ \hline [assms \rightarrow]\ lhs/\sigma \equiv rhs/\sigma \end{array}$$

To justify Rule traces, we make use of a couple of supporting derivations. The name "crewrite" in the lemma stands for "conditional rewrite."

Formal Theorem 46. Not when not nil

$$\texttt{x} = \texttt{nil} \vee \texttt{(not x)} = \texttt{nil}$$

Proof.

$\texttt{x} = \texttt{nil} \lor \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$\texttt{x} = \texttt{nil} \lor \texttt{(if x nil t)} = \texttt{nil}$	Instantiation	(*1)
$\texttt{(not x)} = \texttt{(if x nil t)}$	Definition of not	
$\texttt{x} = \texttt{nil} \lor \texttt{(not x)} = \texttt{(if x nil t)}$	Expansion	
$\texttt{x} = \texttt{nil} \lor \texttt{(not x)} = \texttt{nil}$	Dj. trans. = *1	$\square$

Formal Theorem 47. Crewrite rule lemma

$$\texttt{(iff x t)} \neq \texttt{t} \lor \texttt{(not x)} = \texttt{nil}$$

Proof.

$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x t)} = \texttt{nil}$	Th. iff t when nil	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x t)} \neq \texttt{t}$	Dj. not t from nil	
$\texttt{x} = \texttt{nil} \lor \texttt{(not x)} = \texttt{nil}$	Th. not when nnil	
$\texttt{(not x)} = \texttt{nil} \lor \texttt{(iff x t)} \neq \texttt{t}$	Cut	
$\texttt{(iff x t)} \neq \texttt{t} \lor \texttt{(not x)} = \texttt{nil}$	Commute or	$\square$

With these lemmas in place, Rule traces can be justified as follows. Since the formula for the rule is a theorem, we may begin with a proof of the rule's formula. Let $equiv_R$ be the equivalence relation for this rule. Then, this formula is

$$\texttt{(not } hyp_1\texttt{)} \neq \texttt{nil} \lor \cdots \lor \texttt{(not } hyp_n\texttt{)} \neq \texttt{nil} \lor \texttt{(}equiv_R \ lhs \ rhs\texttt{)} \neq \texttt{nil},$$

and by instantiation we may derive (*1),

$$\texttt{(not } hyp_1/\sigma\texttt{)} \neq \texttt{nil} \lor \cdots \lor \texttt{(not } hyp_n/\sigma\texttt{)} \neq \texttt{nil} \lor \texttt{(}equiv_R \ lhs/\sigma \ rhs/\sigma\texttt{)} = \texttt{t}.$$

We may also assume that we have been given proofs of the subtraces. That is, for each i, we have a proof of $[assms \rightarrow]$ $\texttt{(iff } hyp_i/\sigma \texttt{ t)} = \texttt{t}$, from which we may easily obtain proofs of (*2),

$$[assms \rightarrow] \ \texttt{(not } hyp_i/\sigma\texttt{)} = \texttt{nil},$$

by instantiating crewrite rule lemma and using modus ponens or disjoined modus ponens.

Combining the proofs from *1 and *2, it is straightforward to derive

$$[\mathit{assms} \rightarrow]\ \texttt{(}\mathit{equiv}_R\ \ \mathit{lhs}/\sigma\ \ \mathit{rhs}/\sigma\texttt{)} \neq \texttt{nil}.$$

Then, using equal t from not nil, iff t from not nil, or their disjoined versions, we have

$$[\mathit{assms} \rightarrow]\ \texttt{(}\mathit{equiv}_R\ \ \mathit{lhs}/\sigma\ \ \mathit{rhs}/\sigma\texttt{)} = \texttt{t}.$$

When iffp matches equiv_R, the above is exactly our goal for this trace. Otherwise, equiv_R is `equal` while iffp is `t`. In this case, the above is a proof of

$$[\mathit{assms} \rightarrow]\ \texttt{(equal}\ \ \mathit{lhs}/\sigma\ \ \mathit{rhs}/\sigma\texttt{)} = \texttt{t},$$

and by iff from equal or disjoined iff from equal, we may obtain a proof of our goal,

$$[\mathit{assms} \rightarrow]\ \texttt{(iff}\ \ \mathit{lhs}/\sigma\ \ \mathit{rhs}/\sigma\texttt{)} = \texttt{t}.$$

So far, we have implied that the hyps of each rule are simply terms. Actually, each hypothesis is an aggregate with the following fields:

– *term*, the actual term for this hypothesis,

– *fmode*, a "forcing mode" (Section 9.8),

– *limitp*, a flag indicating whether backchaining should be limited, and

– *limit*, how much to limit backchaining when limitp is set.

What are these limits? We can only apply a conditional rewrite rule when we can show that all of its hypotheses are true. To do this, we typically recursively call the rewriter on each hypothesis to see if it can rewrite the hypothesis to true. To ensure the rewriter terminates, we use a counter called blimit to cap the number of times we can backchain. Ordinarily, when rewriting begins, we initialize blimit with a large number that we do not expect to reach, say 1000. Then, when it comes time to relieve a hypothesis, we decrement the blimit in the recursive call. Once the blimit is exhausted, we are no longer allowed to backchain.

It is useful to give the user more control over the backchain limit, and we do this on a per-hypothesis basis via the limit and limitp annotations. When the rewriter backchains to relieve a hypothesis for which limitp is set, it either decrements the blimit as described above, or sets it to the limit specified by this hypothesis, whichever is smaller. This allows the user to introduce "cheap" rules where the rewriter is not allowed to work very hard at relieving a particular hypothesis. For instance, consider the following rule:

Rule:
```
(implies (not (consp x))
         (equal (car x) nil))
```

We would not expect this rule to be applicable very often, since we usually would not apply `car` to objects which are not conses. But terms matching `(car x)` occur quite frequently in proofs, and this rule will lead us to consider `(consp x)` each time we encounter such a term. Since we have many rules about `consp`, the rewriter could take some time to decide whether the rule applies. To prevent this, we annotate the hypothesis with a limit of one, so the rewriter can backchain at most once more in its effort to show that `(consp x)` is false.

Here we are making a bet: although the limit may prevent the rewriter from applying the rule in certain cases, we think these cases will probably be rare enough that they are not worth looking for. In other words, we want the rewriter to use our rule only opportunistically when it can easily see that it is applicable, and not to spend much time on this rule otherwise. We often use these limits in our work, and they are supported by ACL2's rewriter as well.

We organize rewrite rules into theories. A *theory* is a binary search tree where the keys are symbols, ordered by `symbol-<`, and where the values associated with each key are lists of rewrite rules. We define the leading symbol of a term as follows: the leading symbol of a constant, variable, or lambda abbreviation is `nil`, and the leading symbol of the function application $(f\ t_1\ \ldots\ t_n)$ is f. In a well-formed theory, for each key k, the leading symbol of the lhs of each rule associated with k is also k.

Theories provide an efficient way to filter the set of rules which might apply to a particular term. That is, suppose we want to use rewrite rules to simplify some term, x. We begin by performing a binary search through the theory to find the list of rules which share the same leading symbol as x. These are the only rules we need to consider, because they are the only rules which can match x. This filtering is quite effective and leaves us with only a few rules to try instead of thousands. We expect that rewrite rules target only function applications, so the use of `nil` as the leading symbol for constants, variables, and lambda abbreviations does not cause a problem. ACL2 uses a similar scheme, but stores rules via property lists in its "logical world" instead of using a binary search tree.

The theory is given to the rewriter as part of a larger control structure. A *rewriter control structure* is an aggregate with the following components:

– *theory*, the collection of rewrite rules to use,

– *assmctrl*, the control settings (ctrl) for the assumptions structure,

– *defs*, definitions to use during evaluation,

– *depth*, the stack depth to use for evaluation,

– *noexec*, a list of functions not to evaluate,

– *forcingp*, a flag to control whether forcing is permitted, and

– *betamode*, a way to control when beta-reduction is permitted.

The purpose of the theory, assmctrl, defs, and depth should be evident.

The noexec list is just a list of function names which the rewriter should not evaluate. Preventing evaluation is sometimes useful for maintaining desired abstractions, and we frequently use this feature in the case of constructor functions such as `logic.function` and `logic.pequal`. That is, a rule that targets `(logic.function name args)` will match with a term like `(logic.function 'if '(a b c))`, but not `'(if a b c)`. Another common case of this is for zero-ary functions like `logic.initial-arity-table`; placing these functions on the noexec list allows us to work with them by name instead of by value.

The forcingp flag is either `t` or `nil` and allows the user to globally disable forcing; see Section 9.8 for details.

Finally, betamode is either `nil`, `once`, or `t`, and controls how the rewriter beta-reduces lambda abbreviations. When the mode is `nil`, no beta-reduction is permitted. This is often useful in the early stages of large proofs, because we may be able to avoid expanding some lambdas that are in irrelevant branches of `if`-expressions. Also, since beta-reduction can replicate the actuals of a lambda, avoiding beta-reduction until the actuals have been further simplified may allow us to avoid repeating work. When the mode is `once`, beta-reduction is permitted but we do not recursively rewrite the result. This strategy allows us to expand lambdas incrementally. That is, during each rewriting pass, we may beta-reduce only the outermost lambda. Other techniques, such as case-splitting, can then be used before the next rewriting pass. Finally, when the betamode is `t`, we beta-reduce lambdas and recursively rewrite the result.

9.3 The Rewriter

We now give an overview of our conditional rewriter, CRW. CRW is a fairly complex flag function of eleven arguments:

– *flag*, which is the current mode of operation, and is one of

`term`	rewrite a single term,
`list`	rewrite a list of terms,
`rule`	try to apply a rewrite rule,
`rules`	try to apply a list of rewrite rules,
`match`	try to apply a rule under a particular substitution list,
`matches`	try to apply a rule under a list of substitution lists,
`hyp`	try to relieve a hypothesis, or
`hyps`	try to relieve a list of hypotheses,

– *assms*, which is the (slow) assumptions structure being used,

– x, which is the term, term list, hypothesis, or list of hypotheses we are currently trying to rewrite or relieve, as appropriate for this mode of operation,

– *rule*[s], which is the rule or list of rules we are currently trying to apply, or is `nil` in modes such as `term` and `list` where we are not dealing with particular rules,

– *sigma*[s], which is the substitution list, or list of substitution lists, which we are currently trying to instantiate the rule with, or is `nil` in modes such as `term`, `list`, `rule`, and `rules` where we are not dealing with particular substitutions,

– *cache*, which is used to avoid repeatedly rewriting terms, discussed in Section 9.7,

- *iffp*, which is a flag indicating whether `equal`- or `iff`-equivalence needs to be maintained,

- *blimit*, which is a counter that limits backchaining to ensure that CRW always terminates,

- *rlimit*, which is a counter that limits repeated rewriting to ensure termination,

- *anstack*, which is used to implement a heuristic called the ancestors check, which is discussed in Section 9.4, and

- *control*, which is the control structure being used, and remains fixed throughout the course of rewriting.

Regardless of the mode of operation being used, we think of CRW as producing three values, which are returned as an aggregate:

- *data*, which is the main result and depends upon the mode:

  - In `term` mode, *data* is a rewriter trace that establishes $[assms \longrightarrow]\ x \equiv x'$, where x' is the term that has been x rewritten to. If x cannot be simplified any further, x' will simply be x, via a Failure trace.

  - In `list` mode, *data* is just a list of the traces produced by rewriting each term in the list.

  - In `rule`, `rules`, `match`, or `matches` modes, *data* may either be `nil` indicating that the attempt has failed, or may be a trace that establishes $[assms \longrightarrow]\ x \equiv x'$, as in the term mode.

  - In `hyp` mode, on success *data* is a trace that establishes $[assms \longrightarrow]\ term \equiv$ `t`, where *term* is the term from the hypothesis x; on failure, *data* is nil.

  · In `hyps` mode, *data* is a cons whose `car` is `t` or `nil` indicating whether every hypothesis was successfully relieved. On success, the `cdr` is a list of the traces produced by relieving each hypothesis, and on failure the `cdr` is also `nil`.

– *cache*$'$, which is an updated version of the cache, and

– *alimitedp*, which is a flag used in our caching scheme.

In order to focus on the main part of the rewriter's operation, in this section we will essentially ignore the cache, ancestors stack, and alimitedp flags. To the extent possible, we also ignore forcing, syntactic restrictions, and free variable matching.

The core of CRW is found in the `term` mode, and the other modes are comparatively simple. In `term` mode, CRW operates recursively over the structure of the input term, x, with base cases to handle constants and variables, and recursive cases for `if`-expressions, `not`-expressions, other functions, and lambda abbreviations.

In each of these cases, we might try to use evaluation, assumptions, or rewrite rules to simplify x, or we might instead try to first focus on simplifying the subterms of x. So, the basic questions to answer are, "which approaches should we try?" and "in which order should we try them?" We now explain how each case is handled.

1. Constants.

Constants are very simple. They have no subterms that CRW could recursively simplify. They evaluate to themselves, so there is no need to try evaluation. They come first in the term order, so our assumptions system will not be able to simplify them further. We think it would be strange to target a constant with a rewrite rule, so we do not try to use rewrite rules.

The only thing CRW does with constants is to canonicalize them to `t` or `nil` when we are maintaining `iff`-equivalence. In our implementation, we have built this into our Evaluation mechanism. So, when x is a non-`t`, non-`nil`, constant and *iffp* is `t`, CRW produces an Evaluation trace showing that $[\mathit{assms} \rightarrow]\ x \equiv$ `t`; otherwise, it produces a Failure trace that establishes $[\mathit{assms} \rightarrow]\ x \equiv x$.

2. Variables.

Like constants, variables have no subterms that CRW could recursively simplify. We cannot use evaluation since variables are not ground terms. Like ACL2's rewriter, CRW does not try to apply rewrite rules to variables; we think that if a rewrite rule was allowed to target every variable, it would be tried so frequently that rewriting would be unacceptably slow.

The assumptions system might have inferred that x is equivalent to some simpler term. Accordingly, CRW attempts to use the assumptions system to simplify x. If this is successful, an appropriate Assumptions trace is returned, and otherwise we produce a Failure trace which leaves x unchanged.

3. `If`-expressions.

Suppose x is of the form `(if` a b c`)`. CRW treats this as a special case instead of using our usual strategy for other functions. Even if our goal is to maintain equality while rewriting x, we only maintain `iff`-equivalence while rewriting a to a', since doing so may allow additional rules to apply. We avoid rewriting one of b or c when a' is a constant. Finally, when we rewrite b and c, we additionally assume a' and `(not` a'`)`, respectively, since these assumptions may allow more progress to be made.

We begin by recursively rewriting a to a', maintaining `iff`-equivalence. If a' is a constant, we can check to see whether it is `nil`. If not, we do not need to consider c; we recursively rewrite b to b' using the *iffp* we are given for the entire `if`-expression, and produce an If True Trace which establishes $[\mathit{assms} \longrightarrow]$ `(if` a b c`)` $\equiv b'$. On the other hand, if a' is `nil`, we do not need to consider b; we recursively rewrite c to c' under *iffp*, and return an If False Trace which establishes $[\mathit{assms} \longrightarrow]$ `(if` a b c`)` $\equiv c'$.

When a' is not a constant, we recursively rewrite both b and c under *iffp*. We rewrite b to b' in an extended assumptions structure where we assume a' as a new hypothesis. Similarly, we assume `(not` a'`)` when we rewrite c to c'.

We do not expect rewrite rules to target `if`, yet we would like to reduce terms of the form `(if` y z z`)` to z, and terms of the form `(if` y `nil t)` to `(not` y`)`, so we explicitly check for these cases. When b' and c' are equal, we produce an If Same Trace which establishes $[\mathit{assms} \longrightarrow]$ `(if` a b c`)` $\equiv b'$. Otherwise, we construct an If General Trace which establishes $[\mathit{assms} \longrightarrow]$ `(if` a b c`)` $\equiv$ `(if` a' b' c'`)`; most of the time this is the trace we will return, but if b' is `nil` and c' is `t`, we can construct an If Not Trace which establishes $[\mathit{assms} \longrightarrow]$ `(if` a' b' c'`)` $\equiv$ `(not` a'`)`, and by Transitivity we return a trace of $[\mathit{assms} \longrightarrow]$ `(if` a b c`)` $\equiv$ `(not` a'`)`.

4. `Not`-expressions.

We consider `(not` a`)` to be our canonical form for negative terms whose guts are a, and we prefer this form over variants such as `(if` a `nil t)`, `(equal` a `nil)`, and `(iff` a `nil)` since it is the most compact. Above, we mentioned how `if`-expressions of the form `(if` a `nil t)` are rewritten to `(not` a`)`, and we use ordinary rewrite rules to convert the other kinds of negative terms into the `not`-based form.

When we were first developing CRW, we did not have any special handling for `not`, but we later found that we wanted to allow `iff`-based rules to apply to its argument. To support this, we now handle `not` separately from other functions. We begin by rewriting a to a' under `iff`, then construct a Not Congruence Trace. Usually, this trace establishes $[\mathit{assms} \longrightarrow]$ `(not` a`)` $\equiv$ `(not` a'`)`, but if a' is a constant then the right-hand side will be t or nil, instead.

At this point, we might attempt to further simplify the resulting right-hand side. If we have obtained t or nil, there is really nothing to do. But if we instead have (not a'), where a' is not a constant, what are our options?

Since a' is the result of rewriting, we think of it as being already simplified and do not wish to rewrite it recursively. Like ACL2, we prefer not to allow rewrite rules to target not-expressions, since a more powerful approach is to target the argument, perhaps using iff as the equivalence relation. Evaluation would not be useful here since if a' was a constant then our Not Congruence trace would already have dealt with it. This leaves only assumptions. We ask the assumptions system if (not a') can be simplified, and if so we extend our Not Congruence Trace with the resulting Assumptions trace, via Transitivity.

5. Other functions.

Our most complicated case is for functions other than if and not. Suppose we would like to simplify (f a_1 ... a_n). Here we have a number of options—we could simplify subterms, try rewrite rules, consult our assumptions system, or perhaps use evaluation.

Originally, we did not support outside-in rules. When we added them, we found that they sometimes caused loops with certain "constant gathering" rules, and we mention some details in Section 9.6. Now, as a special consideration to address these kinds of loops, our first step is to check whether each a_i is a constant; if so, we try to use Evaluation to simplify (f a_1 ... a_n) to a constant, unless f is on the noexec list of the control structure.

If such evaluation is not possible, is not permitted, or fails (e.g., because of stack depths or calls of witnessing functions), our next step is to try outside-in rewriting. We begin by looking up f in the current theory, and we then try to apply any outside-in rules which are associated with f. This is done by recursively calling CRW using the `rules` mode, and passing in the outside-in rules for f as the *rule*[*s*] argument. If any rule applies, this produces a Rule trace which establishes

$$[\mathit{assms} \longrightarrow]\ \texttt{(}f \ \ a_1 \ \ \ldots \ \ a_n\texttt{)} \equiv x',$$

where x' is the result of applying the rule. We then recursively rewrite x', which results in a trace that establishes $[\mathit{assms} \longrightarrow]\ x' \equiv x''$. Finally, we combine the two traces using Transitivity, and return the result.

Otherwise we work inside-out. We begin by rewriting the arguments, maintaining equality, by recursively calling CRW in the `list` mode. This produces a list of traces, $[\mathit{assms} \longrightarrow]\ a_1 = {a_1}'$, …, $[\mathit{assms} \longrightarrow]\ a_n = {a_n}'$, which are used as the subtraces of an Equiv By Args trace that establishes

$$[\mathit{assms} \longrightarrow]\ \texttt{(}f \ \ a_1 \ \ \ldots \ \ a_n\texttt{)} \equiv \texttt{(}f \ \ {a_1}' \ \ \ldots \ \ {a_n}'\texttt{)}.$$

We think of this as the first step in an "evolving" trace. That is, let *rhs* be $\texttt{(}f \ \ {a_1}' \ \ \ldots \ \ {a_n}'\texttt{)}$. We now turn now turn our attention to simplifying *rhs*. If, through whatever method, we can show that $[\mathit{assms} \longrightarrow]\ \mathit{rhs} \equiv \mathit{rhs}'$, then by Transitivity we can conclude $[\mathit{assms} \longrightarrow]\ \texttt{(}f \ \ a_1 \ \ \ldots \ \ a_n\texttt{)} \equiv \mathit{rhs}'$. We can then turn our attention to simplifying rhs' to rhs'', and so on.

We begin by trying evaluation. If the resulting ${a_i}'$ are constants, we try to evaluate $\texttt{(}f \ \ {a_1}' \ \ \ldots \ \ {a_n}'\texttt{)}$ unless f occurs in the noexec list of the control structure. If this is permitted and successful, we stop since we have obtained a constant. Our Evaluation trace mechanism ensures that if we are only maintaining `iff`-equivalence, then the result will be `t` or `nil`.

Next, we try inside-out rewrite rules. This is done by recursively calling CRW in the `rules` mode, using the inside-out rules for f as the *rule*[*s*] argument. If some rule applies, this produces a Rule trace which establishes $[\mathit{assms} \rightarrow]\ \mathit{rhs} \equiv \mathit{rhs}'$, and we use it to extend our evolving trace.

Next, we try assumptions. If our assumptions system can simplify the right-hand side, we extend our trace with the resulting Assumptions trace.

Finally, if using rules or assumptions was successful, and the rlimit is nonzero, we try to make additional progress by recursively rewriting the new *rhs* and add the result to our trace.

6. Lambda abbreviations.

The final case in `term` mode is that x is a lambda abbreviation. Suppose x is `((lambda (`v_1 `...` v_n`)` β`)` a_1 `...` a_n`)`. Again there are a number of ways to proceed—we could simplify subterms, use rewrite rules, consult our assumptions system, try evaluation, or use beta-reduction to eliminate the abbreviation.

We begin by recursively simplifying the actuals. We recursively call CRW in the `list` mode, maintaining equality, producing a list of traces, $[\mathit{assms} \rightarrow]\ a_1 = a_1'$, $\ldots$, $[\mathit{assms} \rightarrow]\ a_n = a_n'$, which become the subtraces of a Lambda Equiv By Args trace that establishes

$$[\mathit{assms} \rightarrow]\ \texttt{((lambda (}v_1\ \ldots\ v_n\texttt{)}\ \beta\texttt{)}\ a_1\ \ldots\ a_n\texttt{)} \equiv$$
$$\texttt{((lambda (}v_1\ \ldots\ v_n\texttt{)}\ \beta\texttt{)}\ a_1'\ \ldots\ a_n'\texttt{)}.$$

If all of the resulting a_i' are constants, we have produced a ground term which we try to evaluate. If evaluation is successful, we have reduced the lambda to a constant. In this case, we combine the Lambda Equiv By Args trace with the resulting Evaluation trace, using Transitivity, and return this combined trace.

Otherwise, if evaluation is not possible or fails, we check whether beta-reduction is permitted by this control structure. If the betamode is `once` or `t`, we construct a Beta-Reduction trace which establishes

$$[\mathit{assms} \longrightarrow]\ \texttt{((lambda (}v_1\ \ldots\ v_n\texttt{)}\ \beta\texttt{)}\ {a_1}'\ \ldots\ {a_n}'\texttt{)} \equiv \beta/[v_{1\ldots n} \ldots a_{1\ldots n}'].$$

We can then combine this trace with the Lambda Equiv By Args trace, using Transitivity, to conclude

$$[\mathit{assms} \longrightarrow]\ \texttt{((lambda (}v_1\ \ldots\ v_n\texttt{)}\ \beta\texttt{)}\ a_1\ \ldots\ a_n\texttt{)} \equiv \beta/[v_{1\ldots n} \ldots a_{1\ldots n}']$$

If the betamode is `once`, we return the trace above. But otherwise, when the betamode is `t`, we recursively rewrite $\beta/[v_{1\ldots n} \ldots a_{1\ldots n}']$, and construct a Transitivity to combine its result with the trace above.

This concludes our discussion of the `term` mode. The other modes are more straightforward.

In `list` mode, x is a list of terms which we want to rewrite. We rewrite each term in the list by recursively calling CRW in the `term` mode, and `cons` together all of the traces produced along the way.

In `rules` mode, x is a term we want to rewrite and *rule*$[s]$ is a list of the rewrite rules we would like to try. We try each rule in turn, by recursively calling CRW in the `rule` mode and passing in the particular rule to use as the value for *rule*$[s]$. Such an attempt either produces a trace explaining how the rule was applied to x, or `nil` when the rule could not be applied; we return the trace produced by the first successful attempt, or `nil` if none of the attempts were successful.

In `rule` mode, x is the term we want to rewrite and *rule*$[s]$ is a particular rewrite rule we would like to try. We check that the rule is compatible with the equivalence relation we are maintaining, then try to pattern match the lhs of the rule against x. If this is successful, a substitution list, σ, is produced, which satisfies lhs$/\sigma = x$. At this point, our free variable matching algorithm (see Section 9.5) is used to extend σ into a list of substitution lists, $[\sigma_1, \ldots, \sigma_n]$, each of which satisfies lhs$/\sigma_i = x$. We then try to apply the rule using each of these substitution lists by recursively calling CRW in `matches` mode, using $[\sigma_1, \ldots, \sigma_n]$ for $sigma[s]$.

In `matches` mode, x is the term we want to rewrite, $rule[s]$ is the rule we are trying to use, and $sigma[s]$ are a list of substitution lists which we think may allow us to apply the rule. We try each substitution list in turn, by recursively calling CRW in the `match` mode, passing in the particular σ_i to try for $sigma[s]$. As in the `rules` mode, each such attempt either produces a trace explaining how the rule was applied to x, or `nil` when the rule could not be applied; we return the trace produced by the first successful attempt, or `nil` if none of the attempts were successful.

In `match` mode, x is the term we want to rewrite, $rule[s]$ is the rule we are trying to use, and $sigma[s]$ is the particular substitution list, σ_i, which we want to try. We attempt to relieve all of the hypotheses for the rule by recursively calling CRW in the `hyps` mode, passing in the hyps of the rule for x, the rule we are using as $rule[s]$, and the substitution list we are using as $sigma[s]$. If all of the hypotheses are successfully relieved, then the traces returned by this recursive call may be used as the subtraces for a Rule trace which establishes $[assms \rightarrow]$ lhs$/\sigma_i \equiv$ rhs$/\sigma_i$. Since each σ_i satisfies lhs$/\sigma_i = x$, this is the same as $[assms \rightarrow]\ x \equiv$ rhs$/\sigma_i$, so we return this trace. Otherwise, when some of the hypotheses cannot be relieved, we return `nil` to indicate that the attempt has failed.

In `hyps` mode, x is a list of hypothesis which we want to relieve, $rule[s]$ is the rule we are trying to apply, and $sigma[s]$ is the substitution list, σ_i, we are considering. We attempt to relieve each hypothesis in turn, by recursively calling CRW in the `hyp` mode. Each such attempt either produces a trace establishing that

$$[\mathit{assms} \rightarrow] \; \texttt{(iff} \;\; \mathit{term}/\sigma_i \;\; \texttt{t)},$$

or `nil` to indicate failure. If all of the hypotheses can be relieved successfully, we return a flag indicating success and the list of traces. Otherwise, as soon as any hypothesis fails, we return failure.

This leaves only the `hyp` mode. Here, x is the hypothesis we want to relieve, $\mathit{rule}[s]$ is the rule we are trying to use, and $\mathit{sigma}[s]$ is the particular substitution list, σ_i, which we are trying to use. Let g be x/σ_i, so that our goal is to construct and return a trace which establishes $[\mathit{assms} \rightarrow]$ `(iff` g `t)`. If we cannot do this, we fail by returning `nil`.

Our most general mechanism for constructing this trace is backchaining. The idea is to call CRW recursively in the `term` mode on g, maintaining `iff`. To ensure this terminates, we either decrement the blimit or set it to the limit specified by this rule, whichever is smaller. If the resulting trace establishes $[\mathit{assms} \rightarrow]$ `(iff` g `t)`, then we have successfully relieved the hypothesis.

But there are a couple of reasons we may not be able to backchain. In particular, the blimit may have already been exhausted (as is often the case due to "cheap" hypotheses), or we may have run afoul of the ancestors check heuristic (see Section 9.4). To allow some of these cases to succeed, before backchaining we first try to use evaluation or assumptions to simplify g. If either of these successfully produces a trace of the form $[\mathit{assms} \rightarrow]$ `(iff` g `t)`, we do not need to backchain.

9.4 Ancestors Checking

It is easy to give a rewriter rules that will cause it to loop. A common example is the associativity of `app`. By convention, we ordinarily write this rule so that nested calls of `app` are grouped up to the right, e.g.,

Rule:
```
(equal (app (app x y) z)
       (app x (app y z)))
```

But an alternate rule could be used to group the `app` calls to the left,

Rule:
```
(equal (app x (app y z))
       (app (app x y) z))
```

If both of these rules are given to the rewriter, it will loop when a term such as `(app (app a b) c)` is encountered. That is, applying the first rule will yield `(app a (app b c))`, then applying the second rule will lead us back to `(app (app a b) c)`, and so on. We regard this kind of loop as a failure by the user to provide a good rewriting strategy, and accordingly we do not try to recover from these loops—instead, we simply wait for them to exhaust the rlimit, and at that point we report the situation to the user.

Another kind of loop is more insidious and can occur during backchaining. As an example, we say that the following rule is a "pump" because it can lead us to consider a sequence of "inflating" terms.

Rule:
```
(implies (consp (cdr x))
         (equal (consp x) t))
```

To apply this rule to `(consp a)`, we will need to relieve the hypothesis `(consp (cdr a))`. But the rule matches this hypothesis as well, so if we apply it again we will be led to the hypothesis `(consp (cdr (cdr a)))`, and so on until the blimit is reached. In rare or contrived cases, perhaps `(consp (cdr (cdr (... (cdr x)))))` is known and applying the rule would be useful, but in practice this kind of inflation tends to be useless and expensive.

Another kind of backchaining loop can be caused by a combination of rules. The following rules are not individually pumps, but will destructively interact with one another to lead us into a backchaining loop.

Rule:
```
(implies (true-listp x)
         (equal (consp x) (if x t nil)))
```

Rule:
```
(implies (not (consp x))
         (equal (true-listp x) (not x)))
```

To see the loop, suppose we give both of these rules to the rewriter and begin rewriting `(consp a)`. Using the first rule, we backchain to `(true-listp a)`. Then, using the second rule, we backchain to `(not (consp a))`. As the rewriter descends to `(consp a)`, the loop repeats.

In ACL2, these kinds of loops are avoided using a heuristic called *ancestors checking*, which we reimplement nearly verbatim. The basic approach, dating back to the Boyer-Moore theorem prover [18], is to maintain an *ancestors stack* (the anstack argument to CRW). Each time we backchain, we add a frame to the stack which records

– the term we are trying to relieve,

– the rule we are backchaining on behalf of,

– the GUTS of the term (to avoid recomputation), and

– the number of function occurrences in the term (to avoid recomputation).

Each time we are about to backchain, we compare the new hypothesis to the ancestors stack, and in certain cases we heuristically decide that no further backchaining should be permitted. Backchaining may be prohibited for two reasons:

- If this exact term (or its guts) are already anywhere on the stack, we do not allow the backchain. This catches loops caused by rules such as consp-when-true-listp and true-listp-when-not-consp.
- For every frame caused by the same rule, if (1) the new term looks heuristically "worse" than the old term and (2) the guts of both terms are applications of the same function, we do not allow the backchain. This catches loops introduced by pumps such as consp-when-consp-of-cdr.

The implementation of "worse" is rather subtle, and we do not wish to cover it in detail. Over time, the way in which ACL2 makes this decision has been tweaked to be more efficient and to permit certain kinds of backchaining which were previously prevented. ACL2's current implementation of "worse" has been in use for over seven years, and we have reimplemented it as closely as possible. Since ancestors checking is only used to decide whether or not we will try to relieve a hypothesis, the particular criteria considered are not important from a logical perspective.

9.5 Free-Variable Matching

Many useful rewrite rules have *free variables*—variables which occur in a hypothesis but not in the lhs of the rule. A typical example is a transitivity rule. Here, the lhs only mentions `x` and `z`, but the hypotheses also mention the variable `y`, so `y` is a free variable:

Rule:
```
(implies (and (subsetp x y)
          (subsetp y z))
     (equal (subsetp x z)
            t))
```

Free variables are problematic since they will not be bound in the substitution list created by matching the lhs against particular terms to be rewritten.

For instance, suppose we have assumed `(subsetp a b)` and `(subsetp b c)`, and we would now like to rewrite `(subsetp a c)`. We begin by matching `(subsetp x z)` against `(subsetp a c)`, producing the substitution list $\sigma = [\texttt{x} \leftarrow \texttt{a}, \texttt{z} \leftarrow \texttt{c}]$. Since σ does not mention `y`, if we use it to instantiate our hypotheses, we will think our obligations are to show `(subsetp a y)` and `(subsetp y c)` instead of `(subsetp a b)` and `(subsetp b c)`.

To correct for this, we would like to extend σ with a binding for `y`, say γ, producing $\sigma' = [\texttt{x} \leftarrow \texttt{a}, \texttt{z} \leftarrow \texttt{c}, \texttt{y} \leftarrow \gamma]$. Since `y` is not mentioned in the lhs of the rule, no matter which γ we pick, we will have lhs/σ' = `(subsetp a c)`. So, if we can find any γ which will allow us to relieve the hypotheses, we can use it apply the rule.

Which choices for γ should we try? Here, there is some tension. For each term we try, we will need to try to relieve the hypotheses under the new substitution list. This may be expensive, so we would like to suggest relatively few candidates for γ. On the other hand, if we fail to suggest a workable binding when one exists, the rule will not be applied and we will fail to make progress.

Our approach is fairly conservative. For each free variable v, we say the first hypothesis that mentions v is critical. (We determine which hyps are critical when we create each rule, and record them in the crithyps field so that this computation need not be repeated.) We will try all of the bindings which, in a fairly trivial way, can be sure to satisfy all of the critical hypotheses. In particular, recall that our assumptions system includes trueterms, a list of terms which are known to be non-`nil`. To generate the σ' to try, we try to match each critical hypothesis against these trueterms, using the partial substitution σ as a constraint.

For example, continuing our `subsetp` scenario from above, the trueterms in our assumptions system will be `(subsetp a b)` and `(subsetp b c)`, and the critical hypothesis is `(subsetp x y)`. We will try to match `(subsetp x y)` with each of these trueterms, under the substitution $\sigma = [\texttt{x} \leftarrow \texttt{a}, \texttt{z} \leftarrow \texttt{c}]$. Here,

– we successfully match `(subsetp x y)` with `(subsetp a b)`, producing $\sigma' = [\texttt{x} \leftarrow \texttt{a}, \texttt{z} \leftarrow \texttt{c}, \texttt{y} \leftarrow \texttt{b}]$, but

– we fail to match `(subsetp x y)` with `(subsetp b c)`, because σ requires `x` to be bound to `a`,

so the only σ' generated binds `y` to `b`, which is exactly what we wanted.

Our approach to free-variable matching can be regarded as a simplification of the default behavior in ACL2 (`:match-free :all`). But ACL2 also allows the user to specify other behaviors, such as only considering the first potential match (`:match-free :once`), or calling upon a user-defined function [46] which can inspect the goal and make its own suggestion (`bind-free`). We have not needed to implement these features, but it should not be difficult to do so.

In particular, our free variable matching code is implemented as a function, `(rw.create-sigmas-to-try rule sigma trueterms)`, where `rule` is the rule we are using, `sigma` is the partial substitution which unifies the target term with our rule's lhs, and `trueterms` are the trueterms from the assumptions system. This function produces a list of the σ' which should be attempted. The only logical constraints upon this function are (1) that it produces a list of well-formed substitution lists, and (2) that for every v which is bound in σ, v has the same binding in each σ'. Accordingly, implementing `:match-free :once` would only require adding a field to our rule structures and checking its value.

Implementing `bind-free` would be only slightly more difficult: we would similarly need to annotate our rules, extend `rw.create-sigmas-to-try` with a parameter for the definitions to use for evaluation, and then check that the result of evaluating the bind-free criteria yielded valid substitution lists.

9.6 Syntactic Restrictions

More powerful rewriting strategies are possible when the user can syntactically restrict the application of rewrite rules. In ACL2, this is done with the `syntaxp` [46] mechanism, and our approach is quite similar.

Why are syntactic restrictions useful? Consider an associative, commutative function like `+`. We might like to express the commutativity of `+` as an unconditional rewrite rule,

Rule:
```
(equal (+ a b)
       (+ b a))
```

Normally this rule would loop, e.g., rewriting `(+ x 1)` to `(+ 1 x)`, then back to `(+ x 1)`, and so on. To prevent such loops, we syntactically restrict the rule so that it may only be applied when the term matching `b` is smaller than the term matching `a`, according to the term order. With this restriction, `(+ x 1)` can still be rewritten to `(+ 1 x)`, but `(+ 1 x)` cannot be rewritten back into `(+ x 1)` since `x` is a larger term than `1`.

The commutativity rule above is useful in that it normalizes any single addition so that the smaller term is always on the left. Now, consider another rule,

Rule:
```
(equal (+ a (+ b c))
       (+ b (+ a c)))
```

Without any syntactic restrictions, this rule would cause loops as before, so we restrict it to apply only when the match for `b` is smaller than that for `a`, per the term order. Together, these two rules are sufficient to normalize any right-associated sum so that the smallest terms come first. For instance, here is how `(+ z (+ y x))` would be rewritten, working inside-out. (We avoid using syntactic restrictions for outside-in rules, since the syntactic nature of the subterms we are matching may change during rewriting.)

$$\begin{aligned}
\texttt{(+ z (+ y x))} &\mapsto \texttt{(+ z (+ x y))} && \text{by commutativity-of-+,}\\
&\mapsto \texttt{(+ x (+ z y))} && \text{by commutativity-of-+-two,}\\
&\mapsto \texttt{(+ x (+ y z))} && \text{by commutativity-of-+.}
\end{aligned}$$

It is straightforward to convert any sum into a right-associated sum with the following rule. So, along with the rules above, we can normalize any sum into a right-associated form where the operands are sorted by the term order.

Rule:
```
(equal (+ (+ a b) c)
       (+ a (+ b c)))
```

We also often make use of syntactic restrictions to break normal forms when this will allow us to obtain new ground terms for evaluation. For instance, using the above rules, we would normalize `(+ 1 (+ b (+ a 2)))` to `(+ 1 (+ 2 (+ a b)))`. At this point, we would like to combine the constant terms. In our term order, the constants are contiguous and are smaller than any non-constant terms, so the above rules move them to the front of the sum. We introduce a new rule which looks for constant arguments at the front of the sum which can be grouped for evaluation.

Rule:
```
(equal (+ a (+ b c))
       (+ (+ a b) c))
```

As a syntactic restriction, we may only apply the rule when the terms matching `a` and `b` are constants. In the case of `(+ 1 (+ 2 (+ a b)))`, the rule is allowed to apply and produces `(+ (+ 1 2) (+ a b))`. Then, working inside-out, we see that `(+ 1 2)` is a ground term and evaluate it, producing `3`. Hence, no loop is caused with associativity-of-+ even though the two rules disagree about which normal form to use.

How do we implement syntactic restrictions? Suppose we have matched a term with a rewrite rule, and let σ be a substitution list we would like to try using. Usually, there is only one such σ to try, but when we are using rules with free variables there may be many extensions of the initial substitution list. Suppose that $\sigma = [x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n]$.

Our rewrite rule's syntax field contains a list of terms which we interpret as syntactic restrictions, and only if all of these restrictions are satisfied may attempt to apply the rule. To decide whether some restriction, R, is satisfied, we begin by creating a grounding substitution from σ,

$$\textsc{ground}(\sigma) = [x_1 \leftarrow \text{'}s_1, \ldots, x_n \leftarrow \text{'}s_n],$$

and we apply this substitution to R. We then try to evaluate the resulting term, $R/\textsc{ground}(\sigma)$, using the definitions and stack depth specified in the rewriter's control structure. For R to be satisfied, the evaluation must not fail and must produce a non-`nil` constant. We usually expect that R only mentions the variables involved in the rule, in which case $R/\textsc{ground}(\sigma)$ is a ground term and can be evaluated as long as the stack depth is large enough and all of the functions mentioned are defined.

As a special consideration, we do not use the evaluator described in Section 6.4 to evaluate syntactic criteria. Instead, we use a slightly modified evaluator, called the syntax evaluator, which can evaluate our term order function, `logic.term-<`, and also `logic.constantp`, as primitives in the style of `logic.base-evaluator`. There are two reasons for this.

One is a bootstrapping problem. When we begin to recreate our ACL2 proofs in Milawa, we want to be able to put syntactic restrictions on rules about functions like `+` and `equal` before we have even defined `logic.term-<` and `logic.constantp`. By building these functions into the evaluator, we avoid needing to maintain some separate list of definitions for evaluating syntactic restrictions.

The second is efficiency. Many of our syntactic restrictions are about the term order, but deciding whether terms are in order requires us to count the variables, constants, and function symbols in the two terms. This can be a somewhat expensive computation when the terms involved are large, and our usual evaluator is not very efficient. Building `logic.term-<` into the evaluator allows us to begin using ordinary Lisp evaluation quickly, avoiding this overhead. As a simple benchmark, we recorded the following times and memory usages using ordinary Lisp evaluation, the generic evaluator presented in Section 6.4, and our custom evaluator for syntactic restrictions, when comparing the term `(+ a (+ b (+ c (+ d e))))` against `(+ a (+ b (+ c (+ d f))))`, ten thousand times.

	Lisp evaluation	Generic evaluator	Syntax evaluator
Time	.18 seconds	115 seconds	.34 seconds
Memory Usage	5.6 MB	3.7 GB	7.3 MB

Like the ancestors check, syntactic restrictions are only used to decide whether we should attempt to apply some rule. Hence, from a logical perspective, the particulars of how we make the decision, which evaluators we use, and so on, are not important.

9.7 Rewriter Caching

A recent extension to ACL2 by Boyer and Hunt [16] provides hash-consing, automatic function memoization, and fast association lists where the lookup and update operations use hashing for greater efficiency. These features are quite useful and would be welcome in a more industrially focused version of Milawa, yet they are somewhat at odds with the goals of our project. In particular, we would like to keep the story of execution as simple and believable as possible by avoiding any sophisticated execution tricks.

Using fast association lists, we have implemented a cache for our rewriter that allows us to avoid repeatedly rewriting commonly occurring terms. We would like to stress that this cache is optional and we can disable its use entirely, or treat it as an ordinary association list. We can still carry out all of our proofs with the cache disabled, although more time is required. Also, the proof checking system we developed in Chapter 4, which is used to check all of the proofs of Milawa's fidelity, does not include any of Boyer and Hunt's extensions and cannot make efficient use of the cache.

We implement caches as simple aggregates of *blockp*, a flag indicating whether the cache may be written to, and *data*, a fast association list which maps terms to cache lines. Data can be thought of as an ordinary association list which is accessed via `lookup` and extended by consing, but when we permit fast alists to be used, these operations are instead implemented using a hash table for efficiency.

Data associates with every term, x, a cache line which records up to two traces. One of these traces records how x was rewritten under `equal`, and the other records how x was rewritten under `iff`. Either trace may be omitted if x has not yet been rewritten while maintaining the corresponding equivalence relation. Each line is represented as a pair of the form `(`*eqltrace* `.` *ifftrace*`)`, where each entry is either `nil` or is a trace.

The blockp flag is used in our caching scheme to avoid installing certain "poor" traces into the cache. At various points in its execution, CRW will extend the cache. This is always done with the function `(rw.cache-update term trace iffp cache)`, which returns the extended cache. When blockp is `nil`, we say that the cache is open and `rw.cache-update` installs the given trace into the cache line for this term. But when blockp is `t`, we say the cache is blocked and no such update is actually performed; the given cache is returned unchanged.

It is not too difficult to justify our use of caching. Our basic idea is that every trace we put into the cache should be well-formed and should be carried out using a fixed set of assumptions. One consequence of this is that we must use fresh caches when we recursively rewrite b and c in `(if` *a b c*`)`, since the assumptions we are working with have changed. At any rate, if all of the traces in the cache are well-formed, then any time we take a trace from the cache it will also be well-formed.

Implicit in the word "caching" is a notion of transparency: regardless of whether the cache is used, the results of rewriting should be the same. This is a subtle matter, and to develop a more effective caching scheme we are willing to sacrifice some degree of transparency. Despite this, our rewriter is still a function in the mathematical sense. The cache is not "hidden" as some kind of variable in imperative programming, but is instead given to the rewriter as an argument. Because of this, the rewriter always produces a unique output for any inputs. When we say our caching scheme is not entirely transparent, we only mean to convey that if all of the other inputs are held constant, the output can vary based on the cache.

The simplest example of this pertains to the rewrite limit. To ensure that CRW terminates, we decrease the rlimit parameter each time we recursively rewrite a term, and no further rewriting is permitted once the rlimit is exhausted. This counter ruins simple attempts to memoize calls of the rewriter, e.g., knowing that we were able to rewrite `(consp x)` to `t` using an rlimit of 97 does not necessarily tell us what it will rewrite to with an rlimit of 96.

In practice, we expect the rlimit will never be hit in the ordinary course of a proof—in fact, we print a message if this happens, to warn the user that his rules are probably looping. Accordingly, our caching scheme ignores the rlimit. This violates transparency in that the sense that there are some terms which will be rewritten differently when caching is enabled than they would when it is disabled, but allows us to have a much more useful cache where the stored results may be used throughout many levels of recursive invocation.

The backchain limit is similar. We decrement the blimit parameter each time we attempt to relieve a hypothesis, and no further backchaining is permitted once the blimit reaches zero. As with the rlimit, rewriting `(consp x)` to `t` using a backchain limit of 998 does not necessarily mean we can do the same with a limit of 997.

But the backchain limit is more subtle to handle well, because we do expect it to be encountered in the course of relieving "cheap" hypotheses (page 303). Because of this, simply ignoring the backchain limit, as we ignore the rewrite limit, is not a good option. Imagine that the rewriter first considers `(consp x)` in the context of a very low backchain limit, and fails to rewrite it. It would be a shame to remember this fact and give up on rewriting `(consp x)` in a later, less-restricted setting, where more work could have been attempted.

Our approach is to ignore the backchain limit only until we begin working to relieve a hypothesis with an explicit limit. The idea is that the backchain limit is unlikely to be hit unless a cheap hypothesis has lowered it. Once a cheap hypothesis is encountered, we prevent CRW from writing to the cache while it attempts to relieve the hypothesis. In particular,

1. we record whether or not the cache is currently blocked,
2. we put the cache into blocking mode,
3. we recursively invoke CRW to rewrite the hypothesis, and
4. we restore the original blocking mode.

The most delicate part of our caching scheme is the ancestors check. Much like the backchain limit, it would be dangerous to ignore the ancestors stack since we might "poorly" rewrite x in a context where we have many ancestors restrictions, then reuse this result in a less-restricted context.

To prevent this, we develop a notion of an ancestors-limited rewrite. The idea is to identify which rewrites may have been limited due to ancestors checking, and to avoid adding them to the cache. In particular:

– An attempt to relieve a hypothesis is ancestors-limited if (1) the ancestors check prevents this hypothesis from being pursued, or (2) when we rewrote the hypothesis, the rewrite was ancestors-limited and the result was not a constant.

– An attempt to apply a rule is ancestors-limited if every potential match we considered failed, and at least one of the attempts failed due to an ancestors-limited attempt to relieve a hypothesis.

- An attempt to rewrite a term is ancestors-limited if (1) none of the rules we attempted were successful, (2) at least one of the rules we attempted to apply was ancestors-limited, and (3) other simplification methods such as evaluation and assumptions were not successful.

We perform this computation in CRW and pass the result along in the alimitedp flag of the return value.

When should CRW try to use the cache, and when should traces be added? Like deciding when to use evaluation, assumptions, rewrite rules, and subterm simplification, there are many options, and the best course is not necessarily clear without experimentation.

Our approach is to use and extend the cache only in the `term` and `hyp` modes; in other modes, we are only concerned with properly passing the cache around and performing the alimitedp computation.

In `term` mode, we think of rewriting constants and variables as being relatively cheap, and accordingly we do not consult the cache or extend it with the traces we construct. For `(if` a b c`)`, we do not try to make use of the cache, but we take care to create new, empty caches to use when rewriting b and c, since different sets of assumptions are used. We do not use the cache when rewriting `(not` a`)`, since this only involves a little work beyond rewriting a. We also do not consult the cache for lambda abbreviations.

In fact, our only use of the cache is our handling of functions besides `if` and `not`, say `(`f a_1 ... a_n`)`. Here, after the a_i are rewritten, we check whether `(`f $a_1{}'$ ... $a_n{}'$`)` is cached, and reuse its trace if so. Otherwise, after we finish rewriting `(`f $a_1{}'$ ... $a_n{}'$`)` into *rhs*, we add the trace which establishes `(`f $a_1{}'$ ... $a_n{}'$`)` $\equiv$ *rhs* to the cache, unless this rewrite was ancestors limited. Of course, if the cache is blocked, our attempt to extend it may not produce any changes.

In the hyp mode, we are somewhat more aggressive. Let g be the instantiated term for the hypothesis. Here, we immediately consult the cache to see whether the trace for g has already been computed, and if so we use the result. When no such trace exists, we follow the approach outlined in Section 9.3. Barring ancestors limitations, we add the result of rewriting g to the cache before we return.

The strategy just outlined has a interesting interaction with cheap hypotheses. For instance, suppose we are attempting to apply some rule with (consp x) as a hypothesis with a backchain limit of zero. Although we are not allowed to backchain, we still consult the cache. If in the course of applying previous rules, (consp x) was successfully rewritten to t in a non-cheap context, we can use the cached result to apply the rule.

9.8 Forcing Hypotheses

In many conditional rewrite rules, there is no reason to expect that the hypotheses will be satisfied. For instance, consider the following rule.

Rule:
```
(implies (subsetp x y)
         (equal (disjointp x y)
                (not (consp x))))
```

During proof attempts, the lhs of this rule, (disjointp x y), will match with any term of the form (disjointp a b). When this match occurs, we do not necessarily expect to be able to show that a is a subset of b. Indeed, this will often not be the case.

On the other hand, there are certain hypotheses which we expect to always be true. A good example of these are type-like hypotheses. For instance, recall that we have introduced `logic.function`, `logic.function-name`, and `logic.function-args` as aliases for `cons`, `car`, and `cdr`, respectively. Since our logic is untyped and total, nothing forces us to "properly" use these aliases; we are always free to write, say, `(logic.function-args 5)`, or to call `car` instead of `logic.function-name` when we are working with a function application. But we adopt a discipline whereby we always use these aliases when working with function application terms, and we never use them improperly. In fact, we make use of ACL2's guard mechanism [52] to mechanically enforce this discipline.

Now, consider the following rule.

Rule:
```
(implies (and (logic.functionp x)
              (logic.termp x))
         (equal (logic.term-listp (logic.function-args x))
                t))
```

Because of our discipline, if we encounter a term of the form `(logic.function-args` *a*`)` during a proof, we think it is reasonable to expect that *a* is a valid function application term. Accordingly, we think that any time the lhs of this rule matches some term, the hypotheses should be true.

In a statically typed logic, this rule would probably not even have hypotheses. Instead, we would have introduced terms as a sum type with function applications as one of the disjuncts, and the type system would not allow us to apply `logic.function-args` to a natural number or to take the `car` of a function application. In fact, the rule itself would be unnecessary since `(logic.function-args x)` would have type `term list`.

But type-like hypotheses are only one example, and often there are additional requirements on the use of our functions which cannot be expressed as simple types.

As an example, recall the faithfulness theorem for build.reflexivity,

Rule:
```
(implies (and (logic.termp a)
              (logic.term-atblp a atbl)
              (memberp (axiom-reflexivity) axioms))
         (logic.proofp (build.reflexivity a) axioms thms atbl))
```

In a typed logic, the signature of build.reflexivity would be term → appeal, so the first hypothesis would be taken care of by the type system. It is difficult to imagine expressing the other hypotheses as types. Even so, they express conditions which are necessary for our use of build.reflexivity to be sensible, and we expect them to hold any time that we are concerned with the validity of the proof created by build.reflexivity.

In ACL2, the user can instruct the rewriter to *force* such hypotheses. Ordinarily, if a hypothesis cannot be rewritten to t, ACL2 simply fails to apply the rule since its application cannot be justified. But when ACL2's rewriter fails to relieve a forced hypothesis, it will instead "pretend" the hypothesis could be rewritten to t. Later, if the rest of the proof has been successful, we must return to these pretended steps and show that each forcibly assumed hypothesis is justified.

For many reasons, this deferral can be useful. [27, 42, 87]

Upon seeing a forcibly assumed hypothesis, the user often realized he has not properly stated his theorem. In fact, he often needs to add the very hypothesis being forced to make his conjecture true. As an example, many of our proof-building functions like build.reflexivity only produce valid proofs when certain theorems and axioms have been established, and when the arities of certain functions like if and equal are as expected. Having so many of these functions, it is easy to forget the precise requirements for using them.

In other cases, the forcibly assumed hypothesis may indeed be provable, but may simply not be possible to establish through rewriting using the currently available rules. Here, forcing effectively allows us to apply other techniques (e.g., induction, generalization, case splitting, etc.), or to try rewriting with different rules.

Finally, forcing allows for a certain optimization. In the course of rewriting a clause or a list of clauses, we may force a number of hypotheses, leaving us with a list of forcing obligations, say $[\mathit{assms}_1 \rightarrow \texttt{(iff}\ h_1\ \texttt{t)}, \ldots, \mathit{assms}_n \rightarrow \texttt{(iff}\ h_n\ \texttt{t)}]$. This list often contains duplicates, and as an optimization we can remove the duplicates before we begin proving these new goals. In some of our proofs, hundreds of duplicate goals are eliminated this way.

How do we implement forcing? Recall that each of our hypothesis structures includes an fmode field that specifies the forcing mode to be used while relieving the hypothesis. When a hypothesis should not be forced, its fmode is `nil`. Otherwise, its fmode is either `weak` or `strong`, and we may force it.

The difference between `weak` and `strong` forcing is somewhat subtle. To relieve a hypothesis h, we need to rewrite it to `t`. One way we might fail to do this is by rewriting h to some variable or function application which we do not know how to simplify further. Weak and strong forcing handle this case identically, by forcibly assuming h. But another way we can fail is by rewriting h to `nil`. Here, forcing h is more questionable, since we have effectively "disproved" h from these hypotheses. On one hand, we may still be able to prove the forced goal for h if, in the course of further rewriting and other techniques, we can identify some contradictory hypotheses. On the other, if this is really our only way to prove the goal, what is gained by forcing the hypothesis? After all, the other assumptions are already present in the clause we are rewriting, so the contradiction should be evident without forcing.

Rather than globally choose one behavior or the other, we leave it up to the user. If a hypothesis is only weakly forced, then we still allow it to fail when it is rewritten to `nil`. Strong forcing, on the other hand, forces the hypothesis even when it is rewritten to `nil`.

In practice, we only make use of weak forcing throughout our proofs, which is the behavior ACL2 uses. But we suspect strong forcing may be useful in the future. When we were developing our ACL2 proof sketch, we often ran into cases where hypotheses were not being forced as we expected. These problems led to minor changes in ACL2 to improve forcing. It may be that there are similar corner cases in Milawa, and that strong forcing is, in fact, desirable in certain cases.

In our tracing mechanism, we relieve each forced hypothesis using a new kind of trace,

Forcing Trace
(Must be justified later)

$$\frac{}{[\mathit{assms} \longrightarrow]\ \texttt{(iff}\ \mathit{hyp}\ \texttt{t)}}$$

How can we justify the use of Forcing traces? To begin with, we write a function, `rw.collect-forced-goals`, which walks over a trace and gathers the formulas for every subtrace whose method is `force`. This function is used in two ways. First, our trace compiler, `rw.compile-trace`, expects to be given a list of proofs of these formulas as an argument. This makes it trivial for the compiler to construct a proof for any Forcing trace—we simply use the given proof of its formula. Second, when we call upon our rewriter to simplify some term, we also use `rw.collect-forced-goals` to gather up a list of the forced obligations we have incurred.

Adding Forcing traces to CRW is straightforward. We attempt to relieve hypotheses in a uniform way, regardless of whether they are forced. But before failing to relieve a hypothesis, we check whether forcing should be used. This decision is made by checking the fmode for the hypothesis and the forcingp flag in the control structure. This latter flag allows the user to disable forcing globally, which can be useful in the early stages of a large proof. Then, if forcing is permitted, we create a Forcing trace for the hypothesis instead of failing.

9.9 Justifying the Rewriter

We now summarize our ACL2 proof of the justification of CRW. The proof has two steps. First, we show that any well-formed trace can be compiled into a proof of its formula. Then, we show that CRW always produces a well-formed trace.

Our trace compiler, `rw.compile-trace`, takes three arguments: the trace to compile, the list of function definitions for evaluation, and the proofs of any forced goals. Before we describe its implementation, here are the ACL2 theorems which establish that it is well-typed, relevant, and faithful.

ACL2 Code
```
(defthm logic.appealp-of-rw.compile-trace
  (implies (and (rw.tracep x)
                (rw.trace-okp x defs)
                (definition-listp defs)
                (logic.appeal-listp fproofs)
                (subsetp (rw.collect-forced-goals x)
                         (logic.strip-conclusions fproofs)))
           (logic.appealp (rw.compile-trace x defs fproofs))))

(defthm logic.conclusion-of-rw.compile-trace
  (implies (and (rw.tracep x)
                (rw.trace-okp x defs)
                (definition-listp defs)
                (logic.appeal-listp fproofs)
                (subsetp (rw.collect-forced-goals x)
                         (logic.strip-conclusions fproofs)))
```

```
          (equal (logic.conclusion
                  (rw.compile-trace x defs fproofs))
                 (rw.trace-formula x))))

(defthm logic.proofp-of-rw.compile-trace
  (implies (and (rw.tracep x)
                (rw.trace-okp x defs)
                (rw.trace-atblp x atbl)
                (rw.trace-env-okp x defs thms atbl)
                (definition-listp defs)
                (logic.formula-list-atblp defs atbl)
                (subsetp defs axioms)
                (logic.appeal-listp fproofs)
                (logic.proof-listp fproofs axioms thms atbl)
                (subsetp (rw.collect-forced-goals x)
                         (logic.strip-conclusions fproofs))
                ... various arities are correct ...
                ... various formulas are axioms ...
                ... various formulas are thms ...
                )
           (logic.proofp (rw.compile-trace x defs fproofs)
                         axioms thms atbl)))
```

Our notion of what constitutes a well-formed trace is captured by the four predicates about x which are mentioned in the proof of faithfulness.

- (rw.tracep x) determines if x is a structurally well-formed trace. That is, x must be a cons tree of a certain shape, with a symbol for its method field, terms for its lhs and rhs fields, a Boolean for its iffp field, a structurally well-formed hypbox for its hypbox field, and, recursively, a list of structurally well-formed traces for its subtraces field.

- (rw.trace-atblp x atbl) determines if x is well-formed with respect to the arity table atbl. That is, the lhs and rhs of x, and every term in the hypbox of x, must have proper arities with respect to atbl, and every subtrace must also satisfy these criteria.

– (rw.trace-okp x defs) determines if x is well-formed with respect to most of the requirements for each kind of trace introduced in Section 9.1. For instance, if x is a Failure trace, its lhs and rhs must be the same; if x is a Transitivity trace, it must have two subtraces which agree with its equivalence relation, its lhs and rhs must be the lhs and rhs of the first and second subtraces, respectively, and so on. The defs parameter is needed to ensure that evaluation traces have the correct conclusion. Every subtrace must also satisfy these criteria.

– (rw.trace-env-okp x defs thms atbl) determines if x meets the criteria for rewrite rules from Section 9.2. That is, for every Rule trace in x, the formula for the rule must be among the given thms. Additionally, the rule and substitution list being used must be well-formed with respect to the given arity table, atbl.

We implement our trace compiler as a flag function with two modes, one to compile an individual trace, and one to compile a list of traces.

Definition: rw.flag-compile-trace

```
(pequal*
 (rw.flag-compile-trace flag x defs fproofs)
 (if (equal flag 'trace)
     (let* ((subtraces (rw.trace->subtraces x))
            (subproofs (rw.flag-compile-trace 'list subtraces defs
                                               fproofs)))
       (rw.compile-trace-step x defs subproofs fproofs))
   (if (consp x)
       (cons (rw.flag-compile-trace 'trace (car x) defs fproofs)
             (rw.flag-compile-trace 'list (cdr x) defs fproofs))
     nil)))
```

The work of compiling each individual step is handled by rw.compile-trace-step, which simply inspects the method of the trace and invokes a separate function for each kind of trace. We can easily extend this function to add new kinds of traces without changing the proofs for the current traces. This function can be understood as a first-order approximation of a polymorphic call.

Definition: rw.compile-trace-step
```
(pequal* (rw.compile-trace-step x defs proofs fproofs)
         (let ((method (rw.trace->method x)))
           (cond ((equal method 'fail)
                   (rw.compile-fail-trace x))
                 ((equal method 'transitivity)
                   (rw.compile-transitivity-trace x proofs))
                 ((equal method 'equiv-by-args)
                   (rw.compile-equiv-by-args-trace x proofs))
                 ... and so on ...
                 ((equal method 'force)
                   (rw.compile-force-trace x fproofs))
                 )))
```

Finally, we have a compiler function for each kind of trace. We will only show a few examples. The simplest compiler is for Failure traces, where no subproofs need to be considered. Even here we have cases for iffp and for whether there are assumptions.

Definition: rw.compile-fail-trace
```
(pequal* (rw.compile-fail-trace x)
         (let* ((hypbox (rw.trace->hypbox x))
                (iffp   (rw.trace->iffp x))
                (lhs    (rw.trace->lhs x)))
           (if (and (not (rw.hypbox->left hypbox))
                    (not (rw.hypbox->right hypbox)))
               ;; no assms: just conclude (equiv x x) by reflexivity
               (if iffp
                   (build.iff-reflexivity lhs)
                 (build.equal-reflexivity lhs))
             ;; assms: start with (equiv x x), then expand
             (if iffp
                 (build.expansion (rw.hypbox-formula hypbox)
                                  (build.iff-reflexivity lhs))
               (build.expansion (rw.hypbox-formula hypbox)
                                (build.equal-reflexivity lhs))))))
```

Many traces require subtraces. In the compiler for the trace, we assume we are given proofs of the formulas for these subtraces. As an example, here is our compiler for Transitivity traces.

Definition: `rw.compile-transitivity-trace`

```
(pequal*
 (rw.compile-transitivity-trace x proofs)
 (let* ((hypbox (rw.trace->hypbox x))
        (iffp   (rw.trace->iffp x))
        (proof1 (first proofs))
        (proof2 (second proofs)))
   (if (and (not (rw.hypbox->left hypbox))
            (not (rw.hypbox->right hypbox)))
       (if iffp
           (build.transitivity-of-iff proof1 proof2)
         (build.transitivity-of-equal proof1 proof2))
     (if iffp
         (build.disjoined-transitivity-of-iff proof1 proof2)
       (build.disjoined-transitivity-of-equal proof1 proof2)))))
```

Forcing traces are unique. Since the caller of `rw.compile-trace` must provide proofs of all of the formulas for the forced traces, compiling such a trace only involves finding the provided proof in this list.

Definition: `rw.compile-force-trace`

```
(pequal* (rw.compile-force-trace x fproofs)
          (logic.find-proof (rw.trace-formula x) fproofs))
```

We establish that each of these individual compilers is well-typed, relevant, and faithful. The relevance theorem always shows that the compiler produces a proof of the formula for this trace. These results are then combined to prove the three theorems for `rw.compile-trace-step`:

ACL2 Code

```
(defthm logic.appealp-of-rw.compile-trace-step
  (implies (and (rw.tracep x)
                (rw.trace-step-okp x defs)
                (definition-listp defs)
                (logic.appeal-listp proofs)
                (equal (logic.strip-conclusions proofs)
                       (rw.trace-list-formulas
                        (rw.trace->subtraces x)))
                (logic.appeal-listp fproofs)
                (subsetp (rw.collect-forced-goals x)
                         (logic.strip-conclusions fproofs)))
           (logic.appealp
            (rw.compile-trace-step x defs proofs fproofs))))

(defthm logic.conclusion-of-rw.compile-trace-step
  (implies (and (rw.tracep x)
                (rw.trace-step-okp x defs)
                (definition-listp defs)
                (logic.appeal-listp proofs)
                (equal (logic.strip-conclusions proofs)
                       (rw.trace-list-formulas
                        (rw.trace->subtraces x)))
                (logic.appeal-listp fproofs)
                (subsetp (rw.collect-forced-goals x)
                         (logic.strip-conclusions fproofs)))
           (equal (logic.conclusion
                   (rw.compile-trace-step x defs proofs fproofs))
                   (rw.trace-formula x))))

(defthm logic.proofp-of-rw.compile-trace-step
  (implies (and (rw.tracep x)
                (rw.trace-atblp x atbl)
                (rw.trace-step-okp x defs)
                (rw.trace-step-env-okp x defs thms atbl)

                (definition-listp defs)
                (logic.formula-list-atblp defs atbl)
                (subsetp defs axioms)

                (logic.appeal-listp proofs)
                (logic.proof-listp proofs axioms thms atbl)
                (equal (logic.strip-conclusions proofs)
                       (rw.trace-list-formulas
                        (rw.trace->subtraces x)))

                (logic.appeal-listp fproofs)
                (logic.proof-listp fproofs axioms thms atbl)
                (subsetp (rw.collect-forced-goals x)
                         (logic.strip-conclusions fproofs))

                ... various arities are correct ...
                ... various formulas are thms ...
                ... various formulas are axioms ...
                )
           (logic.proofp
            (rw.compile-trace-step x defs proofs fproofs)
            axioms thms atbl)))
```

The three theorems for rw.compile-trace follow, by induction.

The second half of the justification of CRW is to show the trace it produces is well-formed in the sense of logic.proofp-of-rw.compile-trace, and hence can be compiled into a fully expansive proof. That is, we must show the trace returned by CRW satisfies rw.tracep, rw.trace-atblp, rw.trace-okp, and rw.trace-env-okp. This is more difficult to manage since it involves reasoning about the actual operation of CRW, a large and complicated function with many cases.

An important piece of groundwork for carrying out our proof is to introduce a constructor function for each kind of trace. Perhaps the simplest way to represent traces would be as 7-tuples. As an optimization, we instead use trees of a more compact shape, so that only 6 conses are required to construct a trace, and fewer calls of `car` or `cdr` are required to access the various fields. We introduce the constructor function

```
(rw.trace method hypbox lhs rhs iffp subtraces extras)
```

in the style of `logic.function`. That is, `rw.trace` conses together a trace object out of these components. But whereas this is a "general purpose" constructor, we also introduce a "special purpose" constructor for each kind of trace. For instance, here is our constructor for a Transitivity trace. Here, we expect that the arguments will have compatible iffp and assms fields, and that the rhs of `x` will be the lhs of `y`.

Definition: `rw.transitivity-trace`
```
(pequal* (rw.transitivity-trace x y)
         (let ((a      (rw.trace->lhs x))
               (c      (rw.trace->rhs y))
               (hypbox (rw.trace->hypbox x))
               (iffp   (rw.trace->iffp x)))
           (rw.trace 'transitivity hypbox a c iffp (list x y) nil)))
```

As another example, our constructor for If Same traces is shown below. Here, we expect that the arguments, `x`, `y`, and `z`, are traces which are suitable as subtraces. That is, we should be given

$$\texttt{x} : [\mathit{assms} \rightarrow]\ \texttt{(iff}\ x_1\ x_2\texttt{)},$$

$$\texttt{y} : x_2, \mathit{assms} \rightarrow y \equiv w,$$

$$\texttt{z} : \texttt{(not}\ x_2\texttt{)}, \mathit{assms} \rightarrow z \equiv w,$$

and we will produce a new If Same trace which establishes

$$[assms \longrightarrow]\ \texttt{(if}\ \ x_1\ \ y\ \ z\texttt{)} \equiv w.$$

Definition: `rw.crewrite-if-specialcase-same-trace`
```
(pequal* (rw.crewrite-if-specialcase-same-trace x y z)
         (rw.trace 'crewrite-if-specialcase-same
                   (rw.trace->hypbox x)
                   (logic.function 'if
                                   (list (rw.trace->lhs x)
                                         (rw.trace->lhs y)
                                         (rw.trace->lhs z)))
                   (rw.trace->rhs y)
                   (rw.trace->iffp y)
                   (list x y z)
                   nil))
```

We use these specialized constructors to build every trace throughout CRW. Accordingly, to show the trace produced by CRW satisfies our various well-formedness predicates, we mainly need to reason about the conditions under which these constructors produce well-formed traces. We introduce rewrite rules to accomplish this. For instance, in the case of Transitivity traces, we have:

ACL2 Code
```
(defthm rw.tracep-of-rw.transitivity-trace
  (implies (and (rw.tracep x)
                (rw.tracep y))
           (rw.tracep (rw.transitivity-trace x y))))

(defthm rw.trace-atblp-of-rw.transitivity-trace
  (implies (and (rw.trace-atblp x atbl)
                (rw.trace-atblp y atbl))
           (rw.trace-atblp (rw.transitivity-trace x y) atbl)))

(defthm rw.trace-okp-of-rw.transitivity-trace
  (implies (and (equal (rw.trace->iffp x) (rw.trace->iffp y))
                (equal (rw.trace->hypbox x) (rw.trace->hypbox y))
                (equal (rw.trace->rhs x) (rw.trace->lhs y))
                (rw.trace-okp x defs)
                (rw.trace-okp y defs))
           (rw.trace-okp (rw.transitivity-trace x y) defs)))

(defthm rw.trace-env-okp-of-rw.transitivity-trace
  (implies (and (rw.trace-env-okp x defs thms atbl)
                (rw.trace-env-okp y defs thms atbl))
           (rw.trace-env-okp (rw.transitivity-trace x y)
                             defs thms atbl)))
```

And similarly, for If Same traces, we have:

ACL2 Code

```
(defthm rw.tracep-of-rw.crewrite-if-specialcase-same-trace
  (implies (and (rw.tracep x)
                (rw.tracep y)
                (rw.tracep z))
           (rw.tracep
            (rw.crewrite-if-specialcase-same-trace x y z))))

(defthm rw.trace-atblp-of-rw.crewrite-if-specialcase-same-trace
  (implies (and (rw.trace-atblp x atbl)
                (rw.trace-atblp y atbl)
                (rw.trace-atblp z atbl)
                (equal (cdr (lookup 'if atbl)) 3))
           (rw.trace-atblp
            (rw.crewrite-if-specialcase-same-trace x y z)
            atbl)))

(defthm rw.trace-okp-of-rw.crewrite-if-specialcase-same-trace
  (implies
   (and (rw.tracep x)
        (rw.tracep y)
        (rw.tracep z)
        (rw.trace-okp x defs)
        (rw.trace-okp y defs)
        (rw.trace-okp z defs)
        (rw.trace->iffp x)
        (equal (rw.trace->iffp y) (rw.trace->iffp z))
        (equal (rw.trace->rhs y) (rw.trace->rhs z))
        (equal (rw.hypbox->left (rw.trace->hypbox y))
               (cons (logic.function 'not (list (rw.trace->rhs x)))
                     (rw.hypbox->left (rw.trace->hypbox x))))
        (equal (rw.hypbox->left (rw.trace->hypbox z))
               (cons (rw.trace->rhs x)
                     (rw.hypbox->left (rw.trace->hypbox x))))
        (equal (rw.hypbox->right (rw.trace->hypbox y))
               (rw.hypbox->right (rw.trace->hypbox x)))
        (equal (rw.hypbox->right (rw.trace->hypbox z))
               (rw.hypbox->right (rw.trace->hypbox x))))
   (rw.trace-okp
    (rw.crewrite-if-specialcase-same-trace x y z)
    defs)))

(defthm rw.trace-env-okp-of-rw.crewrite-if-specialcase-same-trace
  (implies (and (rw.trace-env-okp x defs thms atbl)
                (rw.trace-env-okp y defs thms atbl)
                (rw.trace-env-okp z defs thms atbl))
           (rw.trace-env-okp
            (rw.crewrite-if-specialcase-same-trace x y z)
            defs thms atbl)))
```

Some of these rules have many hypotheses, but since we only expect CRW to construct well-formed traces, during our main proof effort all of these hypotheses should be true. We force them all.

Rules like `rw.trace-okp-of-rw.transitivity-trace` have hypotheses that refer to the lhs, rhs, and iffp of `x` and `y`, the traces which are being combined. But sometimes, during the proofs for CRW, the matches for `x` and `y` are calls of other trace constructors. Because of this, we must be able to reason about, e.g., the lhs of the trace produced by each constructor. This is generally quite straightforward. For instance, for Transitivity traces, we can prove the following, unconditional rules.

ACL2 Code
```
(defthm rw.trace->hypbox-of-rw.transitivity-trace
  (equal (rw.trace->hypbox (rw.transitivity-trace x y))
         (rw.trace->hypbox x)))

(defthm rw.trace->lhs-of-rw.transitivity-trace
  (equal (rw.trace->lhs (rw.transitivity-trace x y))
         (rw.trace->lhs x)))

(defthm rw.trace->rhs-of-rw.transitivity-trace
  (equal (rw.trace->rhs (rw.transitivity-trace x y))
         (rw.trace->rhs y)))

(defthm rw.trace->iffp-of-rw.transitivity-trace
  (equal (rw.trace->iffp (rw.transitivity-trace x y))
         (rw.trace->iffp x)))
```

With these constructors and our rules for reasoning about them in place, we carry out the proofs for CRW.

Each proof involves a fairly typical induction over the definition of CRW, with cases for each flag. These kinds of proofs are large and must include separate cases for each mode and return value. For instance, in our `rw.tracep` proof, we must simultaneously show that

1. the trace returned in `term` mode is accepted by `rw.tracep`,
2. the cache returned in `term` mode contains traces which are all accepted by `rw.tracep`,

3. the list of traces returned in `list` mode are all accepted by `rw.tracep`,

4. the cache returned in `list` mode contains traces which are all accepted by `rw.tracep`,

and so on, proving similar results for the other modes.

In the end, we introduce `rw.crewrite` as a wrapper for CRW in `term` mode, and we establish the following theorems:

ACL2 Code

```
(defthm rw.tracep-of-rw.crewrite
  (implies (and (rw.assmsp assms)
                (logic.termp x)
                (booleanp iffp)
                (rw.controlp control))
           (rw.tracep
            (rw.crewrite assms x iffp blimit rlimit control))))

(defthm rw.trace-atblp-of-rw.crewrite
  (implies (and (rw.assmsp assms)
                (rw.assms-atblp assms atbl)
                (logic.termp x)
                (logic.term-atblp x atbl)
                (booleanp iffp)
                (rw.controlp control)
                (rw.control-atblp control atbl)
                (equal (cdr (lookup 'not atbl)) 1))
           (rw.trace-atblp
            (rw.crewrite assms x iffp blimit rlimit control)
            atbl)))

(defthm rw.trace-okp-of-rw.crewrite
  (implies (and (logic.termp x)
                (rw.assmsp assms)
                (booleanp iffp)
                (rw.controlp control))
           (rw.trace-okp
            (rw.crewrite assms x iffp blimit rlimit control)
            (rw.control->defs control))))

(defthm rw.trace-env-okp-of-rw.crewrite
  (implies (and (logic.termp x)
                (logic.term-atblp x atbl)
                (rw.assmsp assms)
                (rw.assms-atblp assms atbl)
                (booleanp iffp)
                (rw.controlp control)
                (rw.control-atblp control atbl)
                (rw.control-env-okp control axioms thms)
                (equal (cdr (lookup 'not atbl)) 1))
           (rw.trace-env-okp
            (rw.crewrite assms x iffp blimit rlimit control)
            (rw.control->defs control)
            thms atbl)))
```

We also establish that the hypbox, lhs, and iffp of the resulting trace are what we would expect:

ACL2 Code

```
(defthm rw.trace->hypbox-of-rw.crewrite
  (implies (and (rw.assmsp assms)
                (logic.termp x)
                (booleanp iffp)
                (rw.controlp control))
           (equal (rw.trace->hypbox (rw.crewrite assms x iffp blimit
                                                 rlimit control))
                  (rw.assms->hypbox assms))))

(defthm rw.trace->lhs-of-rw.crewrite
  (implies (and (rw.assmsp assms)
                (logic.termp x)
                (booleanp iffp)
                (rw.controlp control))
           (equal (rw.trace->lhs (rw.crewrite assms x iffp blimit
                                              rlimit control))
                  x)))

(defthm rw.trace->iffp-of-rw.crewrite
  (implies (and (rw.assmsp assms)
                (logic.termp x)
                (booleanp iffp)
                (rw.controlp control))
           (equal (rw.trace->iffp (rw.crewrite assms x iffp blimit
                                               rlimit control))
                  iffp)))
```

9.10 Fast Rewriting

Just as we have both slow and fast versions of our assumptions system, we have slow and fast versions of our rewriter. So far, we have described the slow version, CRW. One reason CRW is inefficient is that each assumption trace must include the equivalence trace that justifies its conclusions, and hence it must use the slow version of our assumptions system. Additionally, there is some overhead involved in constructing rewrite traces: we call `cons` six times to construct a trace, and since we may need to construct many traces during the course of any particular rewrite, this overhead can add up.

The fast version of our rewriter, FAST-CRW, avoids much of this overhead. It takes almost the same arguments as CRW, with two notable exceptions:

– FAST-CRW should be given a fast assumptions system, whereas CRW takes a slow assumptions system, and

– FAST-CRW should be given a fast cache, which we have not yet introduced but is described below, whereas CRW takes a slow (ordinary) cache.

Like CRW, FAST-CRW produces three outputs, *data*, *cache*$'$, and *alimitedp*. Whereas CRW produced a trace for its data, FAST-CRW produces a fast trace (also introduced below), and a updated fast cache. To keep FAST-CRW and CRW in agreement as we make changes, we programmatically generate the definition of FAST-CRW from CRW by simple rewriting.

In Section 8.6, we introduced our fast assumptions system, which uses ordinary lists of terms rather than lists of equivalence traces as the representation of each equivalence set. Fast traces and caches are similar. A *fast trace* is an aggregate of a term, *rhs*, which intuitively is the result of rewriting, and a list of formulas, *fgoals*, which are any formulas that were forcibly assumed during the rewrite. A *fast cache line* is like an ordinary cache line, except that it stores fast traces instead of slow traces; a *fast cache* is like a regular cache, except that terms are associated with fast cache lines instead of ordinary cache lines.

Like the fast version of our assumptions system, the fast traces produced by FAST-CRW do not contain enough information to produce fully expansive proofs. Instead, to justify FAST-CRW, we show that it produces the same results as CRW.

Just as we used the imaging functions SET-IMAGE, DB-IMAGE, and ASSM-IMAGE to relate our fast and slow equivalence sets, equivalence databases, and assumptions structures, we introduce new imaging functions to relate our fast and slow traces and caches. Given a slow rewrite trace, x, TRACE-IMAGE(x) creates the corresponding fast trace, whose rhs is the rhs of x, and whose fgoals are the result of `(rw.collect-forced-goals x)`. Given a slow cache line, x, CLINE-IMAGE(x) creates the corresponding fast cache line by applying TRACE-IMAGE to each trace. Given a slow cache, x, CACHE-IMAGE(x) creates a fast cache by applying CLINE-IMAGE to each cache line.

Showing that FAST-CRW produces the image of CRW is the most difficult proof we have carried out with Milawa. We present the actual ACL2 `defthm` command for our main result, below. To read this, note that:

– `rw.crewrite-core` is a simple wrapper for running CRW in `term` mode, which passes in `nil` for the unused rule[s] and sigma[s] parameters,

– `rw.fast-crewrite-core` is a similar wrapper for FAST-CRW,

– `rw.cresult->data` is an alias for `car`; it just extracts the *data* component of the return value of CRW or FAST-CRW

– `rw.assmsp`, `rw.controlp`, and `rw.cachep` are recognizers for (slow) assumptions systems, control structures, and (slow) caches, respectively, and

– `rw.trace-fast-image` is our implementation of TRACE-IMAGE.

ACL2 Code

```
(defthm rw.trace-fast-image-of-rw.crewrite-core
  (implies
   (and (logic.termp x)
        (rw.assmsp assms)
        (rw.controlp control)
        (rw.cachep cache)
        (booleanp iffp))
   (equal (rw.trace-fast-image
           (rw.cresult->data
            (rw.crewrite-core assms x cache iffp blimit rlimit
                              anstack control)))
          (rw.cresult->data
           (rw.fast-crewrite-core (rw.assms-fast-image assms)
                                  x
                                  (rw.cache-fast-image cache)
                                  iffp blimit rlimit anstack
                                  control)))))
```

But this result is only a corollary of a much more complicated proof.

A common difficulty when carrying out inductive proofs about flag functions (or mutually recursive functions) is that we must simultaneously prove something about each of the different flags. For instance, consider the theorem above. Here, we would like to show that in the `term` mode, the image of CRW is produced by FAST-CRW. But, in the `term` mode, when CRW and FAST-CRW rewrite function calls, they do so by recursively invoking themselves in the `list` mode. To explain how these results are related, we will need an inductive hypothesis about the `list` mode. Similar circumstances lead us to add additional conjuncts to explain how all of the other modes operate. Putting all of this together, the resulting lemma takes about eight pages to write down and involves establishing thirty properties at once. We present the ACL2 `defthm` command for this lemma in Appendix C.

Our use of imaging functions allows us to carry out this proof as an ordinary induction over the definition of CRW. This was not the case in our first attempt to verify FAST-CRW in ACL2. We had originally formulated our theorem in the following style:

```
(defthm rw.trace-fast-image-of-rw.crewrite-core
  (implies (and ...
                (assms-are-okayp assms fast-assms)
                (cache-is-okayp cache fast-cache))
           (equal (rw.trace-fast-image
                   (rw.cresult->data
                    (rw.crewrite-core assms x cache iffp blimit
                                      rlimit anstack control))
                  (rw.cresult->data
                   (rw.fast-crewrite-core fast-assms x fast-cache
                                          iffp blimit rlimit anstack
                                          control)))))
```

Unfortunately, this formulation required a more sophisticated induction. Since CRW and FAST-CRW use different kinds of assumptions systems and caches, they recur in different ways: where CRW adds a slow trace to its slow cache, FAST-CRW adds a fast trace to its fast cache, etc. To reconcile this difference, we would have needed to induct in such a way that `assms` is instantiated as in CRW while `fast-assms` is instantiated as in FAST-CRW, and similarly for `cache` and `fast-cache`.

Introducing an induction scheme like this in ACL2 is, in principle, easy. All we need to do is write a new function, say MERGED-CRW, to simultaneously mimic the behaviors of CRW and FAST-CRW. MERGED-CRW would take as arguments a slow and fast assumptions structure, a slow and fast cache, and all of other arguments that CRW and FAST-CRW share. It would return five values: the slow trace generated by CRW, the fast trace generated by FAST-CRW, the updated slow cache, the updated fast cache, and the alimitedp flag. Then, we should show the correspondence between MERGED-CRW and CRW, and between MERGED-CRW and FAST-CRW. Finally, we would carry out our proof by induction using the definition of MERGED-CRW.

But introducing MERGED-CRW seemed practically difficult. It is not entirely simple to generate MERGED-CRW from the definition of CRW. For instance, where CRW binds `a-trace` to the result of recursively rewriting `a`, we will now need to introduce two bindings, one for the fast trace and one for the slow trace. Meanwhile, writing MERGED-CRW by hand would mean it had to be updated whenever we change CRW, which we found unappealing.

By using imaging functions, we can avoid the need for MERGED-CRW altogether. The only variables in our new formulation of the theorem are the arguments to CRW, and using images we can simply compute the appropriate fast assumptions structure and fast cache from the slow ones. The proof is entirely straightforward, but there are a lot of cases to cover. When ACL2 prints just its summary of the induction scheme, 4,200 lines of output are generated. To enable ACL2 to manage such a large proof, we give its rewriter a "lean" theory where most unnecessary rules are disabled. We also provide hints regarding how to expand the definitions of the two rewriters, and instruct it to print very little since so many goals would be printed. With these optimizations, ACL2 takes about 11 minutes to finish the proof.

9.11 Rewriting Clauses

So far, we have focused on the rewriting of individual terms. We now explain how we can rewrite the literals of a goal clause, $[a_1, \ldots, a_k]$.

Recall that in Sections 7.2 and 7.3, we introduced the update clause and update clause iff rules, which effectively explain how a clause $[a_1, \ldots, a_k]$ can be simplified to $[a_1{}', \ldots, a_k{}']$ when given proofs of $a_i \equiv a_i{}'$ for each i. We cannot use these rules to carry out rewriting on the literals of a clause, because our rewriter produces conditional equivalences of the form $[\mathit{assms} \rightarrow]\ a_i \equiv a_i{}'$ rather than unconditional equivalences. Instead, we will need a new, special purpose routine, which we call CRW-CLAUSE.

Roughly speaking, for each literal in the goal clause, CRW-CLAUSE must first create the initial assumptions system to use (by assuming the negations of the other literals), and then call upon CRW to rewrite the literal under these assumptions. This process is complicated by a couple of optimizations. First, to avoid unnecessary work, CRW-CLAUSE stops early if contradictory assumptions are observed or if an obviously true literal is produced. Second as we rewrite the later literals, we would like to assume the negations of the previously simplified literals, rather than the original literals. That is, suppose we have rewritten a_1 to $a_1{}'$ and are now rewriting a_2. We would like to assume the negation of $a_1{}'$, rather than a_1, because $a_1{}'$ may be "more canonical" than a_1, and hence more useful as an assumption.

Like the revappend disjunction and aux update clause rules, we process the clause in a tail-recursive style where the literals are split up into a "todo" list, $[t_1, \ldots, t_n]$, and a "done" list, $[d_1, \ldots, d_m]$. Initially, all of the literals in the goal clause, $a_1, \ldots, a_k$, are put onto the todo list, and the done list is empty. At each step in the computation, we take t_1 from the todo list, rewrite it, and place the resulting $t_1{}'$ onto the done list. Ignoring early termination, we eventually reach an empty todo list and a done list which contains the simplified terms in reverse order, i.e., $[a_k{}', \ldots, a_1{}']$.

Our main goal for CRW-CLAUSE is to show that we may construct a proof of the original clause when we are given (1) a proof of every formula which is forcibly assumed during the course of rewriting, and (2) if CRW-CLAUSE did not terminate early, a proof of the simplified clause, i.e., the final done list, $[a_k{}', \ldots, a_1{}']$.

To carry out this proof, we work on a step-by-step basis. At every step, we say that $T_{1 \ldots n} \vee D_{1 \ldots m}$ is the step goal, where T_i is the term formula for each t_i, and D_i is the term formula for each d_i. The key part of our proof is to show that we can prove the step goal for a step when we are given (1) proofs of all the forced goals encountered during the rewriting of this step, and (2) if we do not stop early on this step, a proof of the step goal for the next step.

The initial step goal is exactly the formula for the goal clause. Meanwhile, if CRW-CLAUSE does not terminate early, our final step goal is the formula for the simplified clause. So, we can inductively compose the step proofs to arrive at the main result for CRW-CLAUSE as a whole.

How do we take and justify each step?

We begin by constructing an assumptions structure wherein $t_2, \ldots, t_n$ and $d_1, \ldots, d_m$ are assumed to be false: we begin with the empty assumptions structure, then ASSUME-LEFT the $t_2, \ldots, t_n$, and ASSUME-RIGHT the $d_1, \ldots, d_m$. Keeping the assumptions in separate lists allows us to combine the individual rewrites without carrying out excessive propositional manipulation.

Next, before we make any attempt to rewrite t_1, we ask the assumptions structure if it has observed any contradictory assumptions. If so, we stop early because we can prove the goal. In particular, recall from Section 8.3 that in such a case, we may use the equivalence trace which exhibits the contradiction to prove the hypbox formula, which is $T_{2\ldots n} \lor D_{1\ldots m}$. By trivial expansion of this proof, we can arrive at $T_{1\ldots n} \lor D_{1\ldots m}$, our step goal.

Otherwise, when no contradiction has been observed, we call CRW to rewrite t_1, under these assumptions, maintaining `iff`, with an empty cache, and using a control structure and limitations given to CRW-CLAUSE by the user. Let the rhs of the resulting trace be t_1'. By the justification of CRW, if we are given proofs of the forced goals, then we may construct a proof of $P \lor (\texttt{iff}\ \ t_1\ \ t_1') = \texttt{t}$, where P is the hypbox formula. In other words, we may establish

$$(T_{2\ldots n} \lor D_{1\ldots m}) \lor (\texttt{iff}\ \ t_1\ \ t_1') = \texttt{t}.$$

We now inspect t_1' to see if it is an obvious term. Normally, this is not the case, and we continue rewriting the other literals. The new todo list becomes $[t_2, \ldots, t_n]$ and the new done list becomes $[t_1', d_1, \ldots, d_m]$. To justify this, we need to explain how to recover a proof of the original step goal, $(t_1 \neq \texttt{nil} \lor T_{2\ldots n}) \lor D_{1\ldots m}$, when given

– a proof of this new step goal, $T_{2\ldots n} \lor (t_1' \neq \texttt{nil} \lor D_{1\ldots m})$, and

– a proof of the conclusion from CRW, $(T_{2\ldots n} \lor D_{1\ldots m}) \lor (\texttt{iff}\ \ t_1\ \ t_1') = \texttt{t}$.

We construct this proof by cases on n and m. If n and m are both zero, our goal follows by the substitute iff into literal rule. Otherwise, we make use of some auxiliary lemmas, shown below. If $n > 0$ and $m > 0$, we use lemma 1. If $n > 0$ and $m = 0$, we use lemma 2. And if $n = 0$ and $m > 0$, we use lemma 3. The name "ccstep" is short for "CRW-CLAUSE step."

Derived Rule 159. Ccstep lemma 1

$$\frac{\begin{array}{l} L \vee t_2 \neq \texttt{nil} \vee R \\ (L \vee R) \vee (\texttt{iff}\ \ t_1\ \ t_2) = \texttt{t} \end{array}}{(t_1 \neq \texttt{nil} \vee L) \vee R}$$

Derivation. (108)

$L \vee t_2 \neq \texttt{nil} \vee R$	Given
$L \vee R \vee t_2 \neq \texttt{nil}$	Dj. commute or
$(L \vee R) \vee t_2 \neq \texttt{nil}$	Associativity
$(L \vee R) \vee (\texttt{iff}\ \ t_1\ \ t_2) = \texttt{t}$	Given
$(L \vee R) \vee t_1 \neq \texttt{nil}$	Dj. sub. iff into literal
$t_1 \neq \texttt{nil} \vee L \vee R$	Commute or
$(t_1 \neq \texttt{nil} \vee L) \vee R$	Associativity □

Derived Rule 160. Ccstep lemma 2

$$\frac{\begin{array}{l} L \vee t_2 \neq \texttt{nil} \\ L \vee (\texttt{iff}\ \ t_1\ \ t_2) = \texttt{t} \end{array}}{t_1 \neq \texttt{nil} \vee L}$$

Derivation. (86)

$L \vee t_2 \neq \texttt{nil}$	Given
$L \vee (\texttt{iff}\ \ t_1\ \ t_2) = \texttt{t}$	Given
$L \vee t_1 \neq \texttt{nil}$	Dj. sub. iff into literal
$t_1 \neq \texttt{nil} \vee L$	Commute or □

Derived Rule 161. Ccstep lemma 3

$$\frac{\begin{array}{l} t_2 \neq \texttt{nil} \lor R \\ R \lor \texttt{(iff } t_1 \ t_2\texttt{)} = \texttt{t} \end{array}}{t_1 \neq \texttt{nil} \lor R}$$

Derivation. (88)

$t_2 \neq \texttt{nil} \lor R$	Given
$R \lor t_2 \neq \texttt{nil}$	Commute or
$R \lor \texttt{(iff } t_1 \ t_2\texttt{)} = \texttt{t}$	Given
$R \lor t_1 \neq \texttt{nil}$	Dj. sub. iff into literal
$t_1 \neq \texttt{nil} \lor R$	Commute or $\square$

Note that aside from the choice of lemma to use, the cost of performing this step is not dependent upon n or m. This is made possible because of our partitioning of assumptions into left and right lists, so the formulas for our rewrite traces provide easy access to the disjunctions of the T_i and D_i.

Finally, we need to address the case where $t_1{}'$ is an obvious term. Here, we stop early because we can prove the step goal given only proofs of the forced goals from the rewrite. As above, the particulars of the proof depend upon n and m. If n and m are zero, then there are no assumptions and the step goal is simply $t_1 \neq \texttt{nil}$, and this may be derived as follows:

$t_1{}' \neq \texttt{nil}$	Obvious term
$\texttt{(iff } t_1 \ t_1{}'\texttt{)} = \texttt{t}$	Conclusion from CRW
$t_1 \neq \texttt{nil}$	Substitute iff into literal

Otherwise, if n or m is nonzero, we make use of a simple lemma:

Derived Rule 162. Ccstep lemma 4

$$\frac{\begin{array}{l} t_2 \neq \texttt{nil} \\ P \lor \texttt{(iff } t_1 \ t_2\texttt{)} = \texttt{t} \end{array}}{t_1 \neq \texttt{nil} \lor P}$$

Derivation. (87)

$t_2 \neq$ `nil`	Given
$P \vee t_2 \neq$ `nil`	Expansion
$P \vee$ `(iff` t_1 t_2`)` $=$ `t`	Given
$P \vee t_1 \neq$ `nil`	Dj. sub. iff into literal
$t_1 \neq$ `nil` $\vee P$	Commute or ☐

And our derivation begins as follows:

$t_1' \neq$ `nil`	Obvious term
$(T_{2 \ldots n} \vee D_{1 \ldots m}) \vee$ `(iff` t_1 t_1'`)` $=$ `t`	Conclusion from CRW
$t_1 \neq$ `nil` $\vee (T_{2 \ldots n} \vee D_{1 \ldots m})$	Ccstep lemma 4

At this point, if m is zero, then the last line above is our step goal. Otherwise, when $m > 1$, an associativity step produces the step goal, $T_{1 \ldots n} \vee D_{1 \ldots m}$.

Chapter 10

Tactics

In the past few chapters, we have presented some tools for simplifying clauses, including a clause splitting procedure, an `if`-lifting routine, and a rewriter. Each of these techniques may be applied to a goal clause to obtain some new goals. And, given proofs of these new goals, we can recover a proof of the original goal. Because of this, we can compose these techniques to find proofs in a goal-directed manner.

Since discovering proofs usually involves many applications of rewriting, splitting, lifting, and other techniques, it would be tedious for users to explicitly manage proofs by directly working with our various clause simplifiers. Instead, we have developed a tactic system which ties these tools together.

Tactics were introduced as a way to implement goal-directed proofs in the Edinburgh LCF [30] system, and are now used in many theorem provers such as HOL [33], and HOL Light [40]. In the original Edinburgh LCF system, a tactic, t, is a function which takes a goal to prove, g, and produces (1) a new list of subgoals, $g_1, \ldots, g_n$, which, taken all together, imply the original goal, and (2) a function, v, called a validation, which, given proofs of $g_1, \ldots, g_n$, should construct a proof of g.

Since our system is first-order, we cannot use higher-order functions to implement validations. Instead, we implement each tactic as a pair of functions: one which applies the tactic (analogous to t), and one which justifies its use (analogous to v). Since we cannot dynamically construct validations, our approach is far less flexible than LCF-style systems and we face certain implementation challenges.

1. Information collected during the application of a tactic may be needed during its validation. For instance, if we apply our “limited” `if`-lifting routine to a goal, our validation function needs to know what limit we used. In an LCF-style system, this information might simply be encoded into the definition of v. In our system, we need some mechanism for storing this information in the result of the application function, and for retrieving it in the validation function.

2. We cannot dynamically compose existing validation functions to create new validations. That is, in an LCF-style system, suppose that $t_1(g_1) = \langle [g_2], v_1 \rangle$, and $t_2(g_2) = \langle [g_3], v_2 \rangle$. Now, $v_1 \circ v_2$ is a validation that establishes g_1 when given a proof of g_3, and no special infrastructure is needed for managing such compositions. In our system, we need some way to remember which tactics have been applied, so we can call their validation functions at the appropriate times.

To address these issues, we introduce *proof skeletons*. Each skeleton is a structure that keeps track of the goals at a particular point in the proof attempt, and also records how these goals were produced. Our application functions operate on proof skeletons, and produce either an extension of their input skeleton or `nil` to indicate failure. Concretely, each skeleton is an aggregate of the following components:

– *goals*, a list of clauses which still need to be proven,

– *tacname*, the name of the tactic used to produce these goals,

– *extras*, any information which the validation function will need to justify this reduction, and

– *history*, the proof skeleton to which this tactic was applied.

Special handling is required for the initial skeleton. When we would like to prove a new formula, F, we create a skeleton whose only goal is formed by compiling F into a clause (as described in Section 7.1), and whose tacname, extras, and history are all `nil`. The `nil` tacname identifies the skeleton as the starting point for our proof.

Given a skeleton, s, with goal clauses $g_1, \dots, g_n$, applying a tactic involves using some proof technique (`if`-lifting, rewriting, ...) to simplify some or all of the g_i. This process leaves us with a new list of goals, $h_1, \dots, h_m$. If all of these new goals are provable, then each g_i should also be provable. We produce a new skeleton, s', whose goals are the h_i, whose history is s, whose tacname identifies the tactic which was used (so we can later determine which validation function to call), and whose extras are any additional information which will be needed to prove the g_i when given proofs of the h_i.

Our ordinary proof process begins with the creation of an initial skeleton, s_0, from our goal formula, F. We then choose some tactic to apply to s_0, producing s_1; we then choose another tactic to apply to s_1, producing s_2, and so on. Continuing this process, if we are able to reach a skeleton, s_n, with no goals remaining, we have successfully found a proof of F.

To construct the proof, we need to call upon the validation functions for the tactics we have used. Each validation function takes as inputs (1) the skeleton, s, produced by applying the tactic to the previous skeleton, and (2) proofs of all of the goals of s. Letting h be the history for s, the validation function is responsible for constructing proofs of all of the goals of h. Notably, it can inspect the extras of s. The idea is for the application function to store information for the validation function to use (such as the limit for `if`-lifting) in the extras of the skeleton it produces.

To prove our original goal, F, we begin with our final skeleton, s_n. Since s_n

has no remaining goals, its validation function does not need any input proofs, and can construct proofs of the goals for s_{n-1}. We give these proofs to the validation function for s_{n-1}, which produces proofs of the goals for s_{n-2}, and so on. Eventually, we arrive at a proof of the goal for s_0, namely $\textsc{comp}(F) \neq$ `nil`. Then, via the compile formula rule, we may derive F.

10.1 Implementing Tactics

As examples of how tactics may be implemented, we now describe our `split-first` and `split-all` tactics. These tactics combine the `if`-lifting, clause splitting, and clause cleaning routines from Chapter 7 into a single reduction. Given a skeleton, s, with goals $[g_1, \ldots, g_n]$, the `split-first` tactic applies this reduction only to the first goal, g_1, while `split-all` applies it to all of the goals.

Early in the project, we decided to implement `-first` and `-all` versions of many of our tactics, thinking that it would be useful to either focus upon the first goal or work on all the goals together. We also imagined developing a reordering tactic which would allow the user to bring a particular goal to the front. In practice, we mainly use `-all` tactics, but `-first` tactics are also sometimes useful. We have not needed to use reordering, so although it would be straightforward to implement, we have not done so.

Before discussing our `split-first` and `split-all` tactics, we introduce our combined reduction, `clause.split`, which applies `if`-lifting, clause splitting, and clause cleaning. `Clause.split` is a function of four arguments: *liftp*, *llimit*, and *slimit*, described below, and x, a clause to operate on. It returns two values as a pair of the form `(`*progressp* `.` *subgoals*`)`, where *progressp* indicates whether any simplification has taken place, and *subgoals* are the new clauses that x has been split into. It operates in three phases:

1. If *liftp* is `t`, we try to apply `if`-lifting to the literals in the clause. If the *llimit* is zero, we apply our fixed-point function, LIFT, to lift each literal fully. Otherwise, we use our limited lifting algorithm, with *llimit* as the maximum number of tests with which to split each literal.

2. We then apply our clause-splitting routine to the resulting clause. If the *slimit* is zero, we use the unlimited routine which will split based on every top-level `if` in each literal. Otherwise, we use our limited splitting routine so that at most *slimit* splits are permitted.

3. Finally, we call our clause cleaning routine on the resulting goals, which eliminates any redundant subclauses, useless literals, and so on, that may have been introduced in the above phases.

To justify the use of `clause.split`, we introduce a typical builder function, `clause.split-bldr`, which takes the same arguments as `clause.split` but also expects to be given proofs of the resulting clauses. This function chains together the justifications of `if`-lifting, clause splitting, and clause cleaning that were presented in Chapter 7, and it is straightforward to prove it is well-typed, relevant, and faithful. In ACL2, we have the following theorems:

ACL2 Code
```
(defthm logic.appealp-of-clause.split-bldr
  (implies (and (logic.term-listp x)
                (true-listp x)
                (consp x)
                (logic.appeal-listp proofs)
                (equal (clause.clause-list-formulas
                         (cdr (clause.split liftp llimit slimit x)))
                       (logic.strip-conclusions proofs)))
           (logic.appealp
             (clause.split-bldr liftp llimit slimit x proofs))))

(defthm logic.conclusion-of-clause.split-bldr
  (implies (and (logic.term-listp x)
                (true-listp x)
                (consp x)
                (logic.appeal-listp proofs)
                (equal (clause.clause-list-formulas
                        (cdr (clause.split liftp llimit slimit x)))
                       (logic.strip-conclusions proofs)))
           (equal (logic.conclusion
                   (clause.split-bldr liftp llimit slimit x
                                      proofs))
                  (clause.clause-formula x))))

(defthm logic.proofp-of-clause.split-bldr
  (implies (and (logic.term-listp x)
                (true-listp x)
                (consp x)
                (logic.appeal-listp proofs)
                (equal (clause.clause-list-formulas
                        (cdr (clause.split liftp llimit slimit x)))
                       (logic.strip-conclusions proofs))
                (logic.term-list-atblp x atbl)
                (logic.proof-listp proofs axioms thms atbl)
                ... various arities are correct ...
                ... various formulas are thms ...
                ... various formulas are axioms ...
                )
           (logic.proofp
            (clause.split-bldr liftp llimit slimit x proofs)
            axioms thms atbl)))
```

With `clause.split` in place, we can begin implementing our `split-first` tactic. To implement a tactic, we need to provide an application function and a validation function. Our application function, `tactic.split-first-tac`, is as follows.

Definition: `tactic.split-first-tac`

```
(pequal* (tactic.split-first-tac liftp llimit slimit skelly)
         (let ((goals (tactic.skeleton->goals skelly)))
           (if (not (consp goals))
               ;; fail: no clauses to prove
               nil
             (let* ((clause1   (car goals))
                    (split     (clause.split liftp llimit slimit
                                              clause1))
                    (split-len (len (cdr split))))
               (if (not (car split))
                   ;; fail: no progress was made
                   nil
                 (tactic.extend-skeleton
                  (app (cdr split) (cdr goals))
                  'split-first
                  (list liftp llimit slimit split-len)
                  skelly))))))
```

The failure cases are uninteresting, so suppose the goals of *skelly* are $g_1, \ldots, g_n$, and that when we call `clause.split` upon g_1, progress is made and we obtain a new list of subgoals, $h_1, \ldots, h_k$. In this case, we produce a new skeleton by calling `tactic.extend-skeleton`.

– The new goals are formed by appending $[h_1, \ldots, h_k]$ to $[g_2, \ldots, g_n]$. In other words, we replace g_1 with the subgoals it has split into.

– The tacname is `split-first`.

– The extras include the liftp, llimit, and slimit, which will be needed by the validation function when it calls `clause.split-bldr`. We also record k, which will be used by the validation function to identify which of its input proofs establish $h_1, \ldots, h_k$, and which input proofs establish $g_2, \ldots, g_n$.

– The history of the new skeleton is *skelly*, the original skeleton.

Our validation function is `tactic.split-first-compile`. This function will be given the skeleton produced by `tactic.split-first-tac` and a list of proofs which establish the new goals, i.e., $h_1, \ldots, h_k, g_2, \ldots, g_n$, and must produce a list of proofs which establish $g_1, \ldots, g_n$.

Definition: `tactic.split-first-compile`
```
(pequal*
 (tactic.split-first-compile skelly proofs)
 (let* ((history       (tactic.skeleton->history skelly))
        (orig-goals    (tactic.skeleton->goals history))
        (clause1       (car orig-goals))
        (extras        (tactic.skeleton->extras skelly))
        (liftp         (first extras))
        (llimit        (second extras))
        (slimit        (third extras))
        (split-len     (fourth extras))
        (proofs1       (firstn split-len proofs))
        (other-proofs  (restn split-len proofs))
        (clause1-proof (clause.split-bldr liftp llimit slimit
                                          clause1 proofs1)))
   (cons clause1-proof other-proofs)))
```

In other words, we extract from the extras the liftp, llimit, slimit, and k, and extract from the goals of the history g_1. We partition the input proofs into two parts: *proofs1*, the proofs of $h_1, \ldots, h_k$, and *other-proofs*, the proofs of $g_2, \ldots, g_n$. We then use the `clause.split-bldr` to assemble a proof of g_1, giving it the limits, original goal g_1, and the proofs of $h_1, \ldots, h_k$ as inputs. Finally, we add this proof of g_1 to the proofs of $g_2, \ldots, g_n$, to arrive at the desired result.

We now turn our attention to the `split-all` tactic. To begin with, we implement `clause.split-list`, a new function which takes liftp, llimit, and slimit as before, and using these limits, applies `clause.split` to every clause in a list of clauses. Suppose our input clauses are $g_1, \ldots, g_k$, and let G_i be the list of clauses which results from applying `clause.split` to g_i. Then, `clause.split-list` returns the pair `(`*progressp* `.` $[G_1, \ldots, G_k]$`)`, where *progressp* is true if any call of `clause.split` made progress. It is easy to implement `clause.split-list-bldr`, which takes as arguments the limits which were used, the input clauses, and a list of proof lists that establish the G_i, and produces proofs of $g_1, \ldots, g_k$, by repeatedly calling upon `clause.split-bldr`.

Our application function is `tactic.split-all-tac`. The only complication is that the new goals which a tactic produces must be an ordinary list of clauses, whereas `clause.split-list` produces a list of clause lists. To correct for this we use a simple flattening function, and we also store the lengths of the lists in the extras so our validation function can partition its input proofs appropriately for `clause.-split-list-bldr`.

Definition: `tactic.split-all-tac`
```
(pequal* (tactic.split-all-tac liftp llimit slimit skelly)
         (let ((goals (tactic.skeleton->goals skelly)))
           (if (not (consp goals))
               ;; fail: no clauses to prove
               nil
             (let* ((split      (clause.split-list liftp llimit
                                                    slimit goals))
                    (split-lens (strip-lens (cdr split)))
                    (new-goals  (simple-flatten (cdr split))))
               (if (not (car split))
                   ;; fail: no progress was made
                   nil
                 (tactic.extend-skeleton new-goals
                                         'split-all
                                         (list liftp llimit slimit
                                               split-lens)
```

```
                                             skelly))))))
```

Our validation function, `tactic.split-all-compile`, is shown below. We simply extract the necessary information from the extras and call upon the `clause.-split-list-bldr` to construct proofs of the original goals.

Definition: `tactic.split-all-compile`

```
(pequal* (tactic.split-all-compile x proofs)
         (let* ((history     (tactic.skeleton->history x))
                (orig-goals  (tactic.skeleton->goals history))
                (extras      (tactic.skeleton->extras x))
                (liftp       (first extras))
                (llimit      (second extras))
                (slimit      (third extras))
                (lens        (fourth extras))
                (part-proofs (partition lens proofs)))
           (clause.split-list-bldr liftp llimit slimit
                                   orig-goals part-proofs)))
```

We have now discussed the implementation of two tactics. How do we know to call `tactic.split-all-compile` to validate `split-all` skeletons, and to call `tactic.split-first-compile` to validate `split-first` skeletons?

The basic story is straightforward, but we defer some complications to the next section. We begin by implementing a function which can act as a validation for any of our tactics. This is done by consulting the tactic name and calling upon the appropriate validation function. Much like `rw.compile-trace-step` from Section 9.9, this is a first-order approximation of a polymorphic call, with the obvious limitation that adding new tactics requires us to modify the function.

Definition: `tactic.compile-skeleton-step`

```
(pequal* (tactic.compile-skeleton-step x proofs)
         ;; note: simplified definition
         (let* ((tacname    (tactic.skeleton->tacname x)))
           (cond ((equal tacname 'split-first)
                  (tactic.split-first-compile x proofs))
                 ((equal tacname 'split-all)
                  (tactic.split-all-compile x proofs))
                 ... and so on for the other tactics ...
                 )))
```

Given that we can compile any skeleton step, it is straightforward to compile entire skeletons. We do this with `tactic.compile-skeleton`, which repeatedly calls our step compiler until we reach the initial skeleton.

Definition: `tactic.compile-skeleton`

```
(pequal* (tactic.compile-skeleton x proofs)
         ;; note: simplified definition
         (if (not (tactic.skeleton->tacname x))
             proofs
           (tactic.compile-skeleton
            (tactic.skeleton->history x)
            (tactic.compile-skeleton-step x proofs))))
```

10.2 Worlds

Originally, our approach to compiling skeletons was as we have just described. But to support more efficient proof checking at higher levels of our verified proof checkers, we found that a slightly more complicated approach was needed.

Consider evaluation. To justify uses of our evaluator for low-level proof checkers such as `logic.proofp`, we construct ordinary, fully expansive proofs using a builder function that follows the argument laid out in Section 6.4. Whenever the evaluator uses a function's definition, the resulting proof contains an Axiom appeal for that definition, which will need to be checked by `logic.axiom-okp`.

But eventually, in our bootstrapping process, we arrive at a higher-level proof checker which can justify evaluations in one step. To check an evaluation step which claims to show $x = c$, we would need to run our evaluator on x and ensure that it produces c. But which definitions do we give to our evaluator, and how do we know they are all axioms of the current theory?

A simple approach would be to use the *extras* field of the evaluation appeal to save the definitions to use. Then, to accept the proof step, the checking function would need to ensure these definitions are axioms. But this would not be very efficient. We typically perform evaluation using all of the definitions of the current history, and this list can grow quite long. Our complete system includes thousands of definitions, so if evaluation was used frequently during the course of a proof, we would end up repeatedly checking that these definitions are axioms. A more efficient approach would be to decide upon a fixed list of definitions to use for evaluation throughout a proof. We could then check these definitions once and for all, at the start of the proof, rather than upon every evaluation step.

A similar situation arises in rewriting. Here, the control structure to use includes definitions, and also includes rewrite rules which have been organized into a theory. Unlike definitions, we may wish to use a few different theories during the course of a proof. But we would still like to check the validity of these control structures ahead of time, rather than for each individual use of rewriting.

To accomplish this, we introduce a new structure, called the world. We can check the well-formedness of the world ahead of time, then use it to generated well-formed control structures.

A world is an aggregate with many fields. Some fields are quite simple, such as *primaryp*, *secondaryp*, *directp*, and *negativep* flags for the assumptions control system, and *forcingp*, *betamode*, *blimit*, *rlimit*, *defs*, *noexec*, and *depth* settings for the rewriter's control structure. Each world also has an *index* which can be used as an identifier when a single proof involves multiple worlds. But the most interesting fields are *theories* and *allrules*. The theories field provides a mapping from names (symbols) to theories, while allrules is simply a list of all the rewrite rules that have been introduced so far. Given a world and the name of some theory, we can construct a control structure for the rewriter to use.

We say a world is well-formed with respect to an arity table when all of its definitions and rewrite rules are well formed. Similarly, we say a world is well-formed with respect to its environment (i.e., the axioms and theorems) when all of its definitions are axioms, and all of its rules are theorems. When a world is well-formed in both senses, any control structure we construct from it is also well-formed and satisfies the faithfulness criteria for our rewriter.

A well-formed world can also be manipulated in many ways which preserve its well-formedness. Simple changes to settings like *directp*, *betamode*, and *noexec* that do not alter the theories or allrules fields are the clearest examples. But many theory changes are also acceptable, e.g., removing rules from a theory cannot compromise the well-formedness of a world. Similarly, we can modify existing rules by adding syntactic restrictions or backchain limits, since these are merely annotations which do not play a part in the rule's formula. We can even create new theories or add rules to existing theories, so long as the new rules are among the other theories or allrules, and hence have already been checked.

We begin each proof attempt with a well-formed world, say w. This world will be used to form the control structures used by our rewriting tactics until some world-changing tactic is applied, producing a new world, w'. We have six types of world-changing tactics, each of which are well-formedness preserving:

– `simple-world-change`, which can be used to make changes to the various flags and limits such as *forcingp* and *directp*,

– `update-noexec`, which can add and remove functions from the *noexec* list,

– `create-theory`, which can create a new theory,

– `e/d`, which can be used to change a theory by adding and removing rules (the name comes from ACL2, where this is called enabling and disabling),

– `restrict`, which can add new syntactic restrictions to a rule in a theory, and

– `cheapen`, which can add backchain limits to a rule in a theory.

After a world changing tactic is applied, the new world, w', will be used for subsequent applications of our rewriting tactics, until another world-changing tactic is used.

World-changing tactics are like other tactics in that they produce a new proof skeleton. However, none of our world-changing tactics have any effect upon the goals of the skeleton they are extending, so their validation functions are the identity. Instead, the main purpose of world-changing tactics is to explain which world to use at each point in the proof.

For each world-changing tactic, we introduce a function called a *world compiler* which, given the skeleton produced by the application function, and the current world, produces the updated world. For instance, consider the `create-theory` tactic. We begin by writing a function, `(tactic.create-theory name world)`, which produces a new world by (1) incrementing the index, and (2) either adding a new, empty theory of the given name, or leaving the theories unchanged when this name is already in use as a theory. Our application function, then, is `tactic.create-theory-tac`.

Definition: `tactic.create-theory-tac`

```
(pequal* (tactic.create-theory-tac name skelly)
         (tactic.extend-skeleton (tactic.skeleton->goals skelly)
                                 'create-theory
                                 name
                                 skelly))
```

Since the goals of the skeleton are unchanged, the validation function is simply the identity on its input proofs. Meanwhile, the corresponding world compiler function is `tactic.create-theory-compile-world`.

Definition: `tactic.create-theory-compile-world`

```
(pequal* (tactic.create-theory-compile-world skelly world)
         (let ((name (tactic.skeleton->extras skelly)))
           (tactic.create-theory name world)))
```

After introducing similar world compilers for our other world-changing tactics, we can create a world compiler for an arbitrary proof step. Given a skeleton and the previous world, `tactic.compile-worlds-step` produces the updated world after applying this tactic.

Definition: `tactic.compile-worlds-step`

```
(pequal*
 (tactic.compile-worlds-step x world)
 (let ((tacname (tactic.skeleton->tacname x)))
   (cond ((equal tacname 'simple-change-world)
          (tactic.simple-change-world-compile-world x world))
         ((equal tacname 'update-noexec)
          (tactic.update-noexec-compile-world x world))
         ((equal tacname 'create-theory)
          (tactic.create-theory-compile-world x world))
         ((equal tacname 'e/d)
          (tactic.e/d-compile-world x world))
         ((equal tacname 'restrict)
          (tactic.restrict-compile-world x world))
         ((equal tacname 'cheapen)
          (tactic.cheapen-compile-world x world))
         (t
          ;; Other tactics do not change the world
          world))))
```

Accordingly, given the initial world for a proof skeleton, it is straightforward to construct a list of all the worlds used throughout the proof, using the function `tactic.compile-worlds`.

Definition: `tactic.compile-worlds`

```
(pequal* (tactic.compile-worlds x initial-world)
         (if (not (tactic.skeleton->tacname x))
             (list initial-world)
           (if (tactic.world-stepp x)
               (let* ((prev-worlds (tactic.compile-worlds
                                    (tactic.skeleton->history x)
                                    initial-world))
                      (prev-world  (car prev-worlds)))
                 (cons (tactic.compile-worlds-step x prev-world)
                       prev-worlds))
             (tactic.compile-worlds (tactic.skeleton->history x)
                                    initial-world))))
```

Since each world-changing tactic is well-formedness preserving, the list of worlds produced by `tactic.compile-worlds` are all well-formed, so long as the initial world is well-formed.

Previously, we have suggested that the inputs to validation functions are (1) the skeleton produced by the application function, and (2) a list of proofs of the goals for that skeleton. Many of our validation functions still have this format. But for other tactics which make use of rewriting or evaluation, the validation function may also take the list of worlds as a parameter. Our step compiler, then, is actually defined as follows:

Definition: `tactic.compile-skeleton-step`

```
(pequal* (tactic.compile-skeleton-step x worlds proofs)
         (let* ((tacname    (tactic.skeleton->tacname x)))
           (cond ((equal tacname 'split-first)
                  (tactic.split-first-compile x proofs))

                 ((equal tacname 'split-all)
                  (tactic.split-all-compile x proofs))

                 ((equal tacname 'crewrite-all)
                  (tactic.crewrite-all-compile x worlds proofs))

                 ... and so on for the other tactics ...
                 )))
```

And, for our whole-skeleton compiler, we have:

Definition: `tactic.compile-skeleton`

```
(pequal* (tactic.compile-skeleton x worlds proofs)
         (if (not (tactic.skeleton->tacname x))
             proofs
           (tactic.compile-skeleton
            (tactic.skeleton->history x)
            worlds
            (tactic.compile-skeleton-step x worlds proofs))))
```

How does `tactic.crewrite-all-compile` know which world to use? Each world-changing tactic increments the world's index, and the application function for `crewrite-all` records the index of the current world as an extra in the skeleton; the validation function, `tactic.crewrite-all-compile` can then use this index to find the proper world.

10.3 Tactic Library

We have many tactics besides `split-first` and `split-all`, which we now cover in alphabetical order.

Cleanup

The `cleanup` tactic takes no parameters besides the skeleton to operate on. It runs the clause cleaning algorithm from Section 7.4 to simplify all of the outstanding goals, and fails unless progress is made.

Cleaning was not always done by our splitting tactics, but since it is now built into `clause.split`, having a separate `cleanup` tactic is mostly redundant. One slight advantage to using `cleanup` is that more subsumed clauses may be eliminated. That is, suppose our outstanding goal clauses are $g_1, \ldots, g_n$ and we run `split-all`. When we use `split-all`, each g_i splits into a list of subgoals, say $G_i = [g_{i,1}, \ldots, g_{i,k_i}]$. Our clause cleaning routine is then run on each of these lists separately. So, $g_{1,1}$ will be eliminated if it is subsumed by $g_{1,2}$, but not if it is subsumed by $g_{2,1}$. By running the cleaning routine on all of the outstanding goals, we may be able to eliminate additional subsumed clauses.

Conditional Eqsubst

The `conditional-eqsubst-first` and `conditional-eqsubst-all` tactics allow us to simplify a goal by using a conditional equality. Both the `-first` and `-all` forms take three arguments besides the skeleton to operate on, called *hyp*, *lhs*, and *rhs*, each of which should be terms. For the tactic to produce a sensible result, when the *hyp* holds, the *lhs* and *rhs* should be provably equal.

Let $g = [t_1, \ldots, t_n]$ be a goal clause, and let $t_i{}' = \textsc{repl}(t_i, \mathit{lhs}, \mathit{rhs})$ for all i. Then, conditional equality substitution splits g into three subgoals:

1. The correctness of the replacement, [`(not` *hyp*`)`, `(equal` *lhs rhs*`)`], which shows that indeed the *hyp* implies that *lhs* and *rhs* are equal;

2. The applicability of replacement, $[hyp, t_1, \ldots, t_n]$, which shows that if the hyp is false, the goal clause holds for some other reason;

3. The post-replacement goal, $[t_1{}', \ldots, t_n{}']$, formed by replacing *lhs* by *rhs* everywhere throughout the goal clause.

To justify this reduction, we need to be able to derive the original goal clause when given proofs of the formulas for these subgoals. We make use of a couple of auxiliary rules.

Derived Rule 163. Disjoined = nil from negative lit

$$\frac{P \lor \texttt{(not}\ \ a\texttt{)} \neq \texttt{nil}}{P \lor a = \texttt{nil}}$$

Derivation. (19)

$\texttt{x} = \texttt{nil} \lor \texttt{(not x)} = \texttt{nil}$	Th. not when nnil
$\texttt{(not x)} = \texttt{nil} \lor \texttt{x} = \texttt{nil}$	Commute or
$\texttt{(not}\ \ a\texttt{)} = \texttt{nil} \lor a = \texttt{nil}$	Instantiation
$P \lor \texttt{(not}\ \ a\texttt{)} = \texttt{nil} \lor a = \texttt{nil}$	Expansion
$P \lor a = \texttt{nil}$	Given
$P \lor a = \texttt{nil}$	Dj. mp2

□

Derived Rule 164. Conditional eqsubst lemma1

$$\frac{\texttt{(not}\ \ hyp\texttt{)} \neq \texttt{nil} \lor \texttt{(equal}\ \ a\ \ b\texttt{)} \neq \texttt{nil}}{hyp = \texttt{nil} \lor a = b}$$

Derivation. (76)

$\texttt{(not}\ \ hyp\texttt{)} \neq \texttt{nil} \lor \texttt{(equal}\ \ a\ \ b\texttt{)} \neq \texttt{nil}$	Given
$\texttt{(not}\ \ hyp\texttt{)} \neq \texttt{nil} \lor \texttt{(equal}\ \ a\ \ b\texttt{)} = \texttt{t}$	Dj. eq. t fr. nnil
$\texttt{(not}\ \ hyp\texttt{)} \neq \texttt{nil} \lor a = b$	Dj. = from eq.

$a = b \lor$ `(not` hyp`)` $\neq$ `nil`	Commute or
$a = b \lor hyp =$ `nil`	Dj. $=$ nil fr. neg. lit
$hyp =$ `nil` $\lor a = b$	Commute or $\square$

Now, to validate the use of the conditional eqsubst tactic, we can use the following derivation.

`(not` hyp`)` $\neq$ `nil` $\lor$ `(equal` lhs rhs`)` $\neq$ `nil`	Given 1
$hyp =$ `nil` $\lor lhs = rhs$	Cnd. eqsub. lm. 1
$hyp =$ `nil` $\lor t_i = t_i'$	Dj. repl. subterm (*a)
$t_1' \neq$ `nil` $\lor \cdots \lor t_n' \neq$ `nil`	Given 3
$hyp =$ `nil` $\lor t_1' \neq$ `nil` $\lor \cdots \lor t_n' \neq$ `nil`	Expansion
$hyp =$ `nil` $\lor t_1 \neq$ `nil` $\lor \cdots \lor t_n \neq$ `nil`	Dj. upd. clause *a
$hyp \neq$ `nil` $\lor t_1 \neq$ `nil` $\lor \cdots \lor t_n \neq$ `nil`	Given 2
$(t_1 \neq$ `nil` $\lor \cdots \lor t_n \neq$ `nil`$) \lor (t_1 \neq$ `nil` $\lor \cdots \lor t_n \neq$ `nil`$)$	Cut
$t_1 \neq$ `nil` $\lor \cdots \lor t_n \neq$ `nil`	Contraction

Early in the project, we relied upon the conditional eqsubst tactics, along with generalization, to carry out destructor elimination [18]. That is, `cons` is called a constructor, while `car` and `cdr` are said to be destructors. In this case, destructor elimination involves replacing the expressions `(car` x`)` and `(cdr` x`)` with some new, fresh variables.

As a concrete example, suppose we want to apply destructor elimination to the goal clause

[`(not (consp x))`, `(not (foo (car x)))`, `(not (bar (cdr x)))`, `(baz x y)`],

which may be more easily read as an ACL2-style implication,

```
(implies (and (consp x)
              (foo (car x))
              (bar (cdr x)))
         (baz x y)).
```

We can accomplish this in two phases. First, we use conditional eqsubst, letting the variables be as follows:

hyp = `(consp x)`,

lhs = `x`, and

rhs = `(cons (car x) (cdr x))`.

This generates three subgoals. First, to establish the correctness of the replacement, we must show

```
(implies (consp x)
         (equal x (cons (car x) (cdr x)))),
```

which follows easily from the axiom cons of car and cdr. Next, we must show the applicability of the replacement,

```
(implies (and (not (consp x))
              (consp x)
              (foo (car x))
              (bar (cdr x)))
         (baz x y)),
```

which is trivial since it contains complementary literals. Finally, we have the post-replacement goal, where all occurrences of `x` have been replaced by `(cons (car x) (cdr x))`,

```
(implies (and (consp (cons (car x) (cdr x)))
              (foo (car (cons (car x) (cdr x))))
              (bar (cdr (cons (car x) (cdr x)))))
         (baz (cons (car x) (cdr x)) y)).
```

At this point, we would call upon our generalization tactic to replace `(car x)` and `(cdr x)` with fresh variables, say `x1` and `x2`. This leaves us with

```
(implies (and (consp (cons x1 x2))
              (foo (car (cons x1 x2)))
              (bar (cdr (cons x1 x2))))
         (baz (cons x1 x2) y)),
```

which after trivial rewriting can be simplified to

```
(implies (and (foo x1)
              (bar x2)
         (baz (cons x1 x2) y)).
```

This reduced goal may be easier to prove than the original, particularly if we have a rewrite rule about `(bar (cons a b) c)`.

We now have a more automatic elimination tactic for `car` and `cdr`, and because of this we no longer make much use of conditional eqsubst. But it may still be a useful tactic for performing other kinds of destructor elimination.

Crewrite

The `crewrite-first` and `crewrite-all` tactics allow us to perform conditional rewriting. Both of these tactics take three arguments besides the skeleton to operate on: the name of the theory to use, the world, and a "fast" flag that determines whether CRW or FAST-CRW should be used.

Why would we ever use CRW instead of FAST-CRW? In our bootstrapping process, before FAST-CRW is verified, to justify any uses of FAST-CRW we will need to call upon CRW, anyway. Because of this, it can be more efficient to just use CRW from the beginning and save the traces it produces.

Suppose the goals of the skeleton are $[g_1, \ldots, g_n]$. In the case of `crewrite--first`, the application function begins by constructing a control structure from the world, and then calls either CRW-CLAUSE or FAST-CRW-CLAUSE upon the first goal,

g_1. If no progress is made, we fail. Otherwise, rewriting produces (1) a possibly empty list of formulas, $f_1, \ldots, f_m$, which were forced during the rewrite, and (2) if g_1 was not proven, a new subgoal, ${g_1}'$.

As a useful optimization, we remove any duplicate forced formulas, so let $h_1, \ldots, h_k$ be the unique forced formulas. The subgoals for the new skeleton include (1) the compilation of each h_i into a clause, (2) ${g_1}'$, if necessary, and (3) $g_2, \ldots, g_n$. Meanwhile, the extras include information needed by the validation function, such as the name of the theory, the (fast-)traces recorded from the calls to (FAST-)CRW, the simplified goal ${g_1}'$, and the list of unique, forced formulas, $h_1, \ldots, h_k$.

To reverse this simplification, the validation function begins by recovering this information from the extras. Using the provided proof of the compiled clause for each h_i, it constructs a proof of the formula h_i using the compile formula rule. If FAST-CRW was used, we then construct the analogous slow rewrite traces by using CRW to redo the rewrite; otherwise these traces are already available in the extras. We give the trace compiler the proofs of the h_i, the proof for ${g_1}'$, and the traces to compile, to obtain a proof of g_1. Along with the provided proofs for $g_2, \ldots, g_n$, we now have a proof for each g_i.

The situation for `crewrite-all` is quite similar, except that every goal is rewritten, and the removal of duplicate forced formulas can be done even for formulas which were forced in different goal clauses.

Distribute

The `distribute-all` tactic can be used to simplify the goals by removing certain variables. We have not implemented an equivalent `-first` version. Suppose we have a hypothesis of the form `(equal` v x`)` or `(equal` x v`)`, where v is a variable and x is a term which does not mention v. Then, we typically would like to eliminate

v from the clause by replacing its every occurrence with x. We call this distribution. In ACL2, distribution is implemented with the function `remove-trivial-equivalences`, which is slightly more complex than our tactic due to handling equivalence relations other than `equal`.

The `distribute-all` tactic is automatic and takes no arguments besides the skeleton to operate on. It scans each clause for a term matching `(not (equal` v x`))` or `(not (equal` x v`))`, where v is a variable that is not in FREEVARS(x). If such a literal is found, all occurrences of v throughout the clause are replaced with x.

Distribution can be viewed as a special, more automatic case of our fertilization tactic, so we will not separately address its justification.

Elim

The `elim-first` and `elim-all` tactics are somewhat similar to the conditional eqsubst tactic, but allow us to carry out destructor elimination [18] for `car` and `cdr` more automatically.

Our tactic is more primitive than ACL2's destructor elimination procedure. In particular, ACL2 allows the user to introduce `:elim` rules which permit destructor elimination to be applied to user-defined functions and under equivalence relations besides `equal`, whereas our tactic only supports `car` and `cdr` elimination under `equal`. ACL2 also supports `:generalize` rules [12] which allow additional hypotheses to be added about the new variables as the elimination occurs, but we have no such mechanism. Additionally, ACL2 may perform many eliminations simultaneously, while we perform at most one elimination per clause.

Our elimination tactic attempts to identify a variable which is suitable for elimination. Early on, made this decision by searching for the first occurrence of `(car` v`)` or `(cdr` v`)`, for any variable v. But this approach failed to trigger elimination on goals where no destructor occurred, such as `(implies (consp x) (foo x))`.

To correct for this, we tried expanding our heuristic to search for occurrences of `(consp` v`)`. But this sometimes led us to eliminate "bad" variables. For instance, on a goal such as

```
(implies (and (not (consp y))
              (bar y)
              (consp x))
         (foo (car x) (cdr x))),
```

our tactic would choose to eliminate `y` instead of `x`, which is not useful. To correct for this, we now only consider literals whose form is precisely `(not (consp` v`))`, which we think of as hypotheses of the form `(consp` v`)`. This way, above, we would only choose `x` and not `y`.

In the end, to choose a "good" variable, our approach is to first scan the goal for terms of the form `(car` v`)` or `(cdr` v`)` and accumulate, with duplication, such v into a list. If there are any such variables, we choose to eliminate one with maximal duplicity. Otherwise, as a backup plan, we search for the first literal of the form `(not (consp` v`))`, and choose to eliminate v. Otherwise, we fail. In practice, this heuristic seems to reliably choose good variables to eliminate.

After we have chosen the variable to eliminate, say v, we would like to replace each occurrence of v with `(cons` v_1 v_2`)`, where v_1 and v_2 are new, fresh variables that do not occur anywhere else in the clause. This poses a practical problem, because variables in our logic are represented as symbols, and we have no mechanism for programmatically generating symbols.

One solution would be to add symbol generation primitives to our logic. In ACL2, this involves introducing character and string types and an `intern` function that can create symbols from strings. To keep our logic simpler, we have decided against doing this.

Another approach would be to change our term representation so that variables could be indexed. For instance, perhaps we could treat as variables tuples of the form `(var` s n`)` where s is a symbol and n is an index, so that fresh variables could be generated by simply changing the index. We have decided against doing this since it would complicate our connection with Common Lisp, where variables are ordinary symbols.

Instead, our approach is to have the user supply our elimination tactic with symbols to use. When our elimination tactic needs to choose v_1 and v_2, it simply searches the supplied symbols for two variables that are not found in the goal, and fails if fresh variables are not available. This is not a very satisfying solution, but it allows us to implement elimination without adding primitives to the logic or changing our term representation.

Fertilize

The `fertilize` tactic can be used to eliminate an arbitrary equality hypothesis from a clause. That is, suppose that the literal `(not (equal` x y`))` or `(not (equal` y x`))` occurs in the clause. In this case, we think of `(equal` x y`)` or `(equal` y x`)` as a hypothesis. The fertilization tactic can be used to replace every instance of x with y throughout the clause.

Cross-fertilization is tried automatically [18] in Boyer-Moore provers. But it seems difficult to automatically infer when it is desirable to eliminate an equality hypothesis. In practice, automatic fertilization can be frustrating. The prover often chooses to fertilize an equality in the wrong direction, or to fertilize equalities that should be left alone. Because of this, in our ACL2 proof sketch, we have explicitly disabled automatic fertilization except in a few special cases.

Our `fertilize` tactic is entirely manual. It operates only on the first clause, and we have no `-all` version. In addition to the skeleton to operate on, the user must explicitly provide the tactic with the x and y terms to use. The tactic fails unless the clause contains the literal (not (equal x y)) or (not (equal y x)). On success, fertilization produces a new subgoal where every occurrence of x has been replaced with y.

To justify the `fertilize` tactic, we make use of a lemma.

Formal Theorem 48. Fertilize lemma1 helper

$$\texttt{(not (equal x y))} \neq \texttt{nil} \vee \texttt{x} = \texttt{y}$$

Proof.

$\texttt{x} \neq \texttt{y} \vee \texttt{x} = \texttt{y}$	Prop. schema
$\texttt{x} = \texttt{y} \vee \texttt{x} \neq \texttt{y}$	Commute or
$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Dj. not eq. fr. $\neq$
$\texttt{x} = \texttt{y} \vee \texttt{(not (equal x y))} \neq \texttt{nil}$	Dj. neg. lit fr. = nil
$\texttt{(not (equal x y))} \neq \texttt{nil} \vee \texttt{x} = \texttt{y}$	Commute or $\square$

Derived Rule 165. Fertilize lemma 1

$$\frac{}{(t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}) \vee x = y}, \text{ where } t_i \text{ is } \texttt{(not (equal } x\ y\texttt{))}$$

Derivation.

$\texttt{(not (equal x y))} \neq \texttt{nil} \vee \texttt{x} = \texttt{y}$	Fertilize lemma 1
$\texttt{(not (equal } x\ y\texttt{))} \neq \texttt{nil} \vee x = y$	Instantiation
$(t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}) \vee x = y$	Multi-assoc expansion $\square$

We now explain how `fertilize` may be justified. Suppose our original goal clause is $[t_1, \ldots, t_n]$. Let $t_i{}' = \textsc{repl}(t_i, x, y)$ for each i, so the result of fertilization is

$[{t_1}', \ldots, {t_n}']$. We may assume we are given a proof of this resulting clause. Our first step is to establish $x \neq y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$, as follows.

$x \neq y \vee x = y$	Propositional schema		
$x \neq y \vee t_i = {t_i}'$	Disjoined replace subterm		(*a)
${t_1}' \neq \texttt{nil} \vee \cdots \vee {t_n}' \neq \texttt{nil}$	Given		
$x \neq y \vee {t_1}' \neq \texttt{nil} \vee \cdots \vee {t_n}' \neq \texttt{nil}$	Expansion		
$x \neq y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Disjoined update clause	*a	(*b)

Next, we will establish $x = y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$. There are two cases. If some t_i is `(not (equal` x y`))`, then we can obtain our goal as follows.

$(t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}) \vee x = y$	Fertilize lemma 1	
$x = y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Commute or	(*c)

Otherwise, some t_i is `(not (equal` y x`))`, and we only need to commute the equality after using our lemma.

$(t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}) \vee y = x$	Fertilize lemma 1	
$(t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}) \vee x = y$	Disjoined commute =	
$x = y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Commute or	(*c)

Finally, we combine *b and *c to obtain a proof of our original goal clause.

$x = y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	*c
$x \neq y \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	*b
$(t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}) \vee (t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil})$	Cut
$t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Contraction

Generalize

The `generalize-first` and `generalize-all` tactics can be used to replace an arbitrary term with a new, fresh variable, either throughout the first clause or throughout all of the clauses. Suppose we replace some term, t, with a new variable v. Then, given a proof of the reduced clause, it is trivial to prove the original clause via instantiation, using $\sigma = [v \leftarrow t]$.

Like cross-fertilization, generalization is tried automatically [18] in Boyer-Moore provers. Also like cross-fertilization, this automation can be frustrating. It often picks terms for which generalization is not useful, and can leave the user with strange goals that are not provable, even when the original goal is a theorem. Because of this, we almost always disable generalization when working with ACL2.

In comparison, our generalization tactics are manual. The user must explicitly say which term to replace, and provide a new variable as a replacement. Generalization fails if the variable is not fresh or if the term does not occur.

Induct

The `induct` tactic can be used to begin an inductive proof attempt. It applies only to the first goal; we usually apply it at the beginning of a proof when there is only one goal.

In Boyer-Moore provers, considerable automation exists to automatically determine which induction schemes might apply to a conjecture [18], and one can also explicitly instruct the system to induct as suggested by the recursive definition of a function. We have not implemented this automation, but it should be straightforward to layer it atop our more explicit tactic.

In addition to the skeleton to operate on, our `induct` tactic takes as argument m, a measure term, $qs = [q_1, \ldots, q_k]$, a list of terms which determine the induction steps, and *allsigmas*, a list of lists of substitution lists. That is, $allsigmas = [\Sigma_1, \ldots, \Sigma_k]$, where each Σ_i is a list of substitution lists, $\Sigma_i = [\sigma_{\langle i,1\rangle}, \ldots, \sigma_{\langle i,h_i\rangle}]$.

These arguments follow from our description of the induction rule, presented in Section 2.8, except that the *qs* given to our tactic are terms instead of formulas.

We can view each q_i as a formula using the usual interpretation, i.e., $q_i \neq \texttt{nil}$. When the `induct` tactic is successful, the first goal is split into many subgoals, namely the compiled basis step, inductive steps, ordinal steps, and measure steps for this choice of m, qs, and $allsigmas$.

Urewrite

The `urewrite-first` and `urewrite-all` tactics allow us to perform rewriting using URW, another rewriter which we have not yet described.

URW is a simple, light-weight rewriter that makes no assumptions and only applies unconditional rules. Because of this, no backchaining is required, and only two modes of operation (`term` and `list`) are needed. It uses the same rewrite trace system and trace compiler as CRW, and we have a fast version, FAST-URW, in the same spirit as FAST-CRW. A minor advantage of URW is that since no assumptions are made, the proofs generated by compiling its traces typically use the non-disjoined versions of rules. Because of this, URW can result in shorter fully expansive proofs than CRW.

Use

The `use` tactic allows us to add an explicit instance of another theorem as a hypothesis into our clause, and is our analogue of ACL2's `:use` hints.

Suppose our goal clause is $[t_1, \ldots, t_n]$, and we have previously proven $a \neq \texttt{nil}$. The `use` tactic allows us to reduce our goal to $[\texttt{(not } a\texttt{)}, t_1, \ldots, t_n]$. Given a proof of the reduced goal, we can derive the original goal for the `use` tactic as follows.

$a \neq \texttt{nil}$	Previous proof
$\texttt{(not } a\texttt{)} = \texttt{nil}$	Negative lit from $\neq$ nil
$\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Given
$t_1 \neq \texttt{nil} \vee \cdots \vee t_n \neq \texttt{nil}$	Modus ponens

The reduced goal may be easier to prove for a variety of reasons. For instance, our free-variable matching routine may fail to identify that a certain binding is needed. In such cases, the `use` tactic may allow us to explicitly say which binding to use. Alternately, a hypothesis may be difficult to relieve via rewriting, but if we explicitly add it to the clause, we may be able to apply techniques besides rewriting to see that it holds.

The `use` tactic applies only to the first goal, and we do not have a `-all` version. Commonly, if the `use` tactic is necessary, we call upon it early in the proof, before splitting has occurred.

Waterfall

Our most sophisticated tactic is `waterfall`.

In most of our proofs, the real engines of progress are the `split-all` and `crewrite-all` tactics. By alternating the application of these tactics, we effectively carry out the proof in a breadth-first manner. That is, we rewrite every goal, then split every resulting subgoal, then rewrite every resulting subgoal from that, etc. We follow this approach when carrying out most of our proofs.

This strategy makes it easy to carry out staged simplification, wherein the early parts of a proof are carried out in a limited theory consisting mainly of cheap rules, and only later are more expensive rules (such as definitions that introduce many cases) allowed to be used.

But the breadth-first approach can be wasteful when goals are asymmetrically hard to prove. For instance, in an inductive proof, there are often goals like the ordinal and measure steps which are easy, and other goals like the main inductive cases which are much harder. In a cheap theory, we may need only three applications of rewriting and splitting to reduce each easy goal, but ten applications of rewriting and splitting to reduce the hard goals. In this case, the breadth-first approach will require us to try to repeatedly rewrite and split the easy goals after they have already been maximally reduced. If there are many easy goals, this can waste a considerable amount of time.

The `waterfall` tactic avoids this problem by rewriting and splitting each goal in a depth-first manner, until either some limit has been reached or the goals have become maximally stable. As arguments besides the skeleton to simplify, it takes the current world, the name of the theory to use, a strategy to apply (described below), and a maximum number of steps (to ensure termination).

To justify its work, the waterfall builds a tree of waterfall-step structures that record what has been done to each goal. Each waterfall step is an aggregate of the following components:

– *method*, the kind of step this is,

– *clause*, the clause being proven,

– *extras*, any additional information needed to justify this step, and

– *substeps*, any subsidiary waterfall steps which are needed to justify this clause.

We have only implemented four kinds of waterfall steps, but our step structures are flexible enough that new kinds of steps could be added easily. For each kind of step, we must be able to prove the clause when given proofs of the clauses for the substeps. We currently implement the following steps.

– `Stop` steps are atomic and are used when either (1) we cannot make any more progress using this theory, or (2) we are forced to stop because we have taken the maximum number of steps permitted.

– `Urewrite` steps are added when we use our unconditional rewriter. The extras include the theory name being used, the traces generated, etc. A `urewrite` step always has a single substep for the reduced clause.

– `Crewrite` steps are constructed when we call upon the conditional rewriter. The extras include the theory name, traces, etc., and the substeps include the reduced clause and any forced goals, as in the `crewrite-first` tactic.

– `Split` steps are introduced when we use `clause.split` to simplify a clause.

To justify uses of the waterfall tactic, we introduce a compiler which, given proofs of every Stop step, can transform these trees into proofs.

The order in which steps are tried is determined by the strategy, which is a list that names the techniques to apply. For instance, we might use the strategy [`crewrite`, `split`]. For each clause encountered during the waterfall, we try each technique in the strategy in order until one makes progress. We then restart from the beginning of the strategy on each resulting subgoal.

10.4 Verifying Tactics

A basic expectation of any tactic is that every successful application of the tactic can be justified. That is, suppose the application function was given some goals, $g_1, \ldots, g_n$, and produced new goals, $h_1, \ldots, h_m$. Then, given proofs of the h_i, along with whatever additional information was saved in the skeleton, the validation function should be able to construct proofs of each g_i. In this LCF system [30], this property was called *validity*.

It is not difficult to prove each of our tactics is valid. As a representative example, we now cover our ACL2 proof sketch of the validity of the `split-first` tactic. Recall that `tactic.split-first-tac` takes as arguments various settings such as *liftp*, *llimit*, *slimit*, and also takes the input skeleton, x. If no progress is made, it returns `nil` to indicate failure; otherwise it produces a new output skeleton, say x', whose tacname is `split-first`, and which includes the limits to use as extras. Meanwhile, recall that the validation function, `tactic.split-first-compile`, expects to be given the output skeleton, x', along with proofs of the goals of x'. From these inputs, it is intended to produce proofs of the goals of x.

We verify the `split-first` tactic in a slightly indirect way. First, we introduce a new function, `tactic.split-first-okp`, which recognizes when a skeleton is a valid use of the `split-first` tactic. Then, we show that:

1. the application function, `tactic.split-first-tac`, always produces a skeleton which is accepted by `tactic.split-first-okp`, and

2. the validation function, `tactic.split-first-compile`, can be used to validate any skeleton satisfying `tactic.split-first-okp`.

Together, these lemmas establish that every use of `tactic.split-first-tac` can be validated by `tactic.split-first-compile`. As we will see shortly, this indirection provides a useful benefit: the `tactic.split-first-okp` function can be combined with similar recognizers for our other tactics to arrive at a notion of whole-skeleton validity.

The definition of `tactic.split-first-okp` is given below. It simply ensures that the skeleton has the proper tacname and extras, and that the new goals of the skeleton are properly related to the previous goals.

Definition: `tactic.split-first-okp`

```
(pequal*
(tactic.split-first-okp x)
(let ((goals   (tactic.skeleton->goals x))
      (tacname (tactic.skeleton->tacname x))
      (extras  (tactic.skeleton->extras x))
      (history (tactic.skeleton->history x)))
  (and (equal tacname 'split-first)
       (tuplep 4 extras)
       (let ((old-goals  (tactic.skeleton->goals history))
             (liftp      (first extras))
             (liftlimit  (second extras))
             (splitlimit (third extras))
             (split-len  (fourth extras)))
         (and (consp old-goals)
              (booleanp liftp)
              (natp liftlimit)
              (natp splitlimit)
              (let* ((clause1       (list-fix (car old-goals)))
                     (clause1-split (clause.split liftp liftlimit
                                                  splitlimit
                                                  clause1)))
                (and (car clause1-split)
                     (equal split-len (len (cdr clause1-split)))
                     (equal (firstn split-len goals)
                            (cdr clause1-split))
                     (equal (restn split-len goals)
                            (cdr old-goals))))))))
```

When tactic.split-first-tac is given sensible inputs and succeeds in producing a new skeleton, x', it is trivial to see that x' satisfies tactic.split-first-okp by examining the definitions of the two functions. In our ACL2 proof sketch, we have the following theorem.

ACL2 Code

```
(defthm tactic.split-first-okp-of-tactic.split-first-tac
  (implies
   (and (tactic.split-first-tac liftp liftlimit splitlimit x)
        (booleanp liftp)
        (natp liftlimit)
        (natp splitlimit)
        (tactic.skeletonp x))
   (tactic.split-first-okp
    (tactic.split-first-tac liftp liftlimit splitlimit x))))
```

Next, recall that `tactic.split-first-compile` simply extracts the limits to use from the extras of its skeleton, and calls upon `clause.split-bldr` to build the necessary proofs. Since we have already proven `clause.split-bldr` is well-typed, relevant, and faithful, it is easy to arrive at similar theorems for `tactic.split-first-compile`.

ACL2 Code

```
(defthm logic.appeal-listp-of-tactic.split-first-compile
   (implies (and (tactic.skeletonp x)
                 (tactic.split-first-okp x)
                 (logic.appeal-listp proofs)
                 (equal (clause.clause-list-formulas
                         (tactic.skeleton->goals x))
                        (logic.strip-conclusions proofs)))
            (logic.appeal-listp
             (tactic.split-first-compile x proofs))))

(defthm logic.strip-conclusions-of-tactic.split-first-compile
  (implies (and (tactic.skeletonp x)
                (tactic.split-first-okp x)
                (logic.appeal-listp proofs)
                (equal (clause.clause-list-formulas
                        (tactic.skeleton->goals x))
                       (logic.strip-conclusions proofs))))
           (equal (logic.strip-conclusions
                   (tactic.split-first-compile x proofs))
                  (clause.clause-list-formulas
                   (tactic.skeleton->goals
                    (tactic.skeleton->history x))))))

(defthm logic.proof-listp-of-tactic.split-first-compile
   (implies (and (tactic.skeletonp x)
                 (tactic.split-first-okp x)
                 (logic.appeal-listp proofs)
                 (equal (clause.clause-list-formulas
                         (tactic.skeleton->goals x))
                        (logic.strip-conclusions proofs))
                 (tactic.skeleton-atblp x atbl)
                 (logic.proof-listp proofs axioms thms atbl)
                 ... various arities are correct ...
                 ... various formulas are thms ...
                 ... various formulas are axioms ...
                 )
            (logic.proof-listp
             (tactic.split-first-compile x proofs)
             axioms thms atbl)))
```

For each of our other tactics, we introduce similar -okp functions and carry out these validity proofs. We can then combine the -okp functions for the separate tactics into a unified check, which is like another polymorphic call.

Definition: tactic.skeleton-step-okp

```
(pequal* (tactic.skeleton-step-okp x worlds)
         (let ((tacname (tactic.skeleton->tacname x)))
           (cond ((not tacname)
                  t)
                 ((equal tacname 'cleanup)
                  (tactic.cleanup-okp x))
                 ((equal tacname 'conditional-eqsubst)
                  (tactic.conditional-eqsubst-okp x))
                 ... and so on ...
                 ((equal tacname 'split-first)
                  (tactic.split-first-okp x))
                 ... and so on ...
                 (t
              nil))))
```

To establish the validity of certain tactics, such as `use` and `conditional-eqsubst`, we must also ensure that certain newly introduced terms and formulas are well-formed with respect to the arity table, or are among the current axioms and theorems. For these tactics, we also have a second notion of step-validity, `tactic.skeleton-step-env-okp`.

Definition: `tactic.skeleton-step-env-okp`

```
(pequal*
 (tactic.skeleton-step-env-okp x worlds axioms thms atbl)
 (let ((tacname (tactic.skeleton->tacname x)))
   (cond ((equal tacname 'conditional-eqsubst)
          (tactic.conditional-eqsubst-env-okp x atbl))
         ((equal tacname 'conditional-eqsubst-all)
          (tactic.conditional-eqsubst-all-env-okp x atbl))
         ... and so on ...
         ((equal tacname 'use)
          (tactic.use-env-okp x axioms thms atbl))
         (t
          ;; other tactics have no such requirements
          t))))
```

Finally, by combining the proofs for each compiler, we arrive at the three theorems for `tactic.compile-skeleton-step`. Note that the `tactic.skeleton-step-env-okp` is only needed for faithfulness.

ACL2 Code

```
(defthm logic.appeal-listp-of-tactic.compile-skeleton-step
  (implies (and (tactic.skeletonp x)
                (tactic.world-listp worlds)
                (tactic.skeleton-step-okp x worlds)
                (logic.appeal-listp proofs)
                (equal (clause.clause-list-formulas
                        (tactic.skeleton->goals x))
                       (logic.strip-conclusions proofs))))
  (equal (logic.appeal-listp
          (tactic.compile-skeleton-step x worlds proofs))
         t))

(defthm logic.strip-conclusions-of-tactic.compile-skeleton-step
   (implies (and (tactic.skeletonp x)
                 (tactic.world-listp worlds)
                 (tactic.skeleton-step-okp x worlds)
                 (logic.appeal-listp proofs)
                 (equal (clause.clause-list-formulas
                         (tactic.skeleton->goals x))
                        (logic.strip-conclusions proofs)))
            (equal (logic.strip-conclusions
                    (tactic.compile-skeleton-step x worlds proofs))
                   (if (tactic.skeleton->tacname x)
                       (clause.clause-list-formulas
                        (tactic.skeleton->goals
                         (tactic.skeleton->history x)))
                     (clause.clause-list-formulas
                      (tactic.skeleton->goals x))))))

(defthm logic.proof-listp-of-tactic.compile-skeleton-step
  (implies (and (tactic.skeletonp x)
                (tactic.world-listp worlds)
                (tactic.skeleton-step-okp x worlds)
                (logic.appeal-listp proofs)
                (equal (clause.clause-list-formulas
                        (tactic.skeleton->goals x))
                       (logic.strip-conclusions proofs))
                (tactic.skeleton-step-env-okp x worlds axioms
                                              thms atbl)
                (tactic.skeleton-atblp x atbl)
                (logic.proof-listp proofs axioms thms atbl)
                (tactic.world-list-atblp worlds atbl)
                (tactic.world-list-env-okp worlds axioms thms)
            ... various arities are correct ...
            ... various formulas are thms ...
            ... various formulas are axioms ...
           )
       (equal (logic.proof-listp
               (tactic.compile-skeleton-step x worlds proofs)
               axioms thms atbl)
              t)))
```

Finally, we extend the tactic.skeleton-step-okp and tactic.skeleton-step-env-okp across the entire skeleton, to ensure that every step in the skeleton is valid according to one of our tactics.

Definition: tactic.skeleton-okp

```
(pequal* (tactic.skeleton-okp x worlds)
         (if (tactic.skeleton->tacname x)
             (and (tactic.skeleton-step-okp x worlds)
                  (tactic.skeleton-okp (tactic.skeleton->history x)
                                       worlds))
           t))
```

Definition: tactic.skeleton-env-okp

```
(pequal* (tactic.skeleton-env-okp x worlds axioms thms atbl)
         (if (tactic.skeleton->tacname x)
             (and (tactic.skeleton-step-env-okp x worlds axioms thms
                                                 atbl)
                  (tactic.skeleton-env-okp
                   (tactic.skeleton->history x)
                   worlds axioms thms atbl))
           t))
```

An easy proof by induction then establishes that any valid skeleton can be compiled to produce a proof of its original goals. We think of this result as the fidelity of Milawa.

Part IV

Self-Verification

Chapter 11

User Interface

With our theorem prover implemented and the ACL2 proof of its fidelity completed, our attention now turns to using Milawa to (1) rediscover the fidelity proof, and (2) emit this proof in a format that `logic.proofp` can check. An important tool for carrying out this work is a user interface for interacting with Milawa. Our interface provides three main features.

– *Proof management.* The interface provides an environment for applying tactics, controlling theories, and otherwise carrying out proofs using Milawa. It also includes features for debugging proofs, and for rebuilding proofs on multiple machines, in parallel.

– *ACL2 connection.* The interface allows us to read in definitions and theorems from our ACL2 proof sketch. This allows us to avoid duplicating each definition and goal formula, and helps to keep the Milawa proof in sync with the ACL2 sketch.

– *Proof-checking support.* The interface can save fully expansive versions of the proofs it has found, and can also write command files for our proof checker to process.

Our interface is implemented as a collection of ACL2 macros which issue `table` and `make-event` commands. ACL2 tables act like global variables, and `make-event` allows us to inspect the values of these variables and also other parts of the ACL2 state, such as the definitions and theorems that have been accepted by the ACL2 system.

Programming in this style was quite awkward at first, but the resulting interface seems to be reasonable. Without any effort on our part, every Milawa command can be undone. We can also use ACL2's notion of `local` events to limit the scope of commands that manipulate theories and change other parameters.

11.1 Proof Management

A good part of our interface has nothing to do with ACL2, but only provides an environment for applying tactics and carrying out Milawa proofs. This part of our interface is somewhat similar to the Subgoal package in Cambridge LCF [72] or HOL [33], and as such it is not much like ACL2's usual interface. However, recent versions of ACL2 include a new feature called `gag-mode`, which is somewhat closer to this style of interface.

At any point during a proof attempt, the user is shown a (possibly truncated) list of the currently outstanding goals. He inspects these goals and then chooses to apply some tactic. He is then shown a list of the new goals which result from the application of that tactic. This process continues until all the goals are proven.

As an example, given two association lists, x and y, (`submapp` x y) determines whether every key in x is bound to the same value in y as it is in x. Below, we annotate a transcript for the Milawa proof which shows this function is transitive. Note that we have made some minor formatting changes so the transcript will fit into the margins.

We begin with the initial goal. This goal, and all of the other goals shown below, are clauses, but our interface displays them as more familiar ACL2-style implications.

```
One goal remains.

  1. (IMPLIES (AND (SUBMAPP X Y) (SUBMAPP Y Z))
              (EQUAL (SUBMAPP X Z) 'T))
```

At this point, we instruct the system to apply the `use` tactic to add an instance of a previously proven theorem to the goal. Below, `MILAWA !>` is a prompt which is printed by ACL2 when it is ready for input. The user's input is shown in bold. All of the Milawa user-interface commands are prefixed with the **%** character. The interface responds to the command by printing the new goal which is generated by applying the tactic.

```
MILAWA !>(%use (%instance (%thm submapp-badguy-membership-property)
                          (x x)
                          (y z)))
One goal remains.

 1. (IMPLIES
     (AND
      (IF
       (EQUAL
        (EQUAL
         (IMPLIES
          (SUBMAPP-BADGUY X Z)
          (IF (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
              (NOT (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                          (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z)))
              'NIL))
         'T)
        'NIL)
       'NIL
       'T)
      (SUBMAPP X Y)
      (SUBMAPP Y Z))
     (EQUAL (SUBMAPP X Z) 'T))
```

We now instruct the system to apply the waterfall tactic to simplify this goal by case-splitting and rewriting. Our interface expects to be told which theory to use, and we tell it to use the `default` theory. We ask it to run the waterfall for at most 40 steps, which is far more than necessary. Because the waterfall tactic is often expensive, the interface prints some performance notes before printing the reduced goal.

```
MILAWA !>(%waterfall default 40)
;; Waterfall: clause #1 took 1.400088 seconds, producing 1 subgoals
(RW.WATERFALL-LIST-WRAPPER ...) took 1,414,173 microseconds
  (1.414173 seconds) to run with 8 available CPU cores.
  During that period,
   1,368,086 microseconds (1.368086 seconds) were spent in user mode
   32,002 microseconds (0.032002 seconds) were spent in system mode
   23,284,864 bytes of memory allocated.
   5,718 minor page faults, 1 major page faults, 0 swaps.
; Applied waterfall to 1 clauses; 1 remain
One goal remains.

 1. (IMPLIES
     (AND (NOT (SUBMAPP X Z))
          (SUBMAPP Y Z)
          (SUBMAPP X Y)
          (NOT (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                      (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))))
     (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)))
```

We now make a change to the default theory, removing one of its rules which we are about to use. This trick is common in ACL2 proofs. If we explicitly use an instance of an enabled rule, then the rule itself may counterproductively rewrite the instance to `t`, leaving us where we started. Disabling the rule before using it is a simple way to prevent this.

```
MILAWA !>(%disable default equal-of-lookups-when-submapp)
Removing 1 rules from DEFAULT.
```

We now use some instances of the rule. After each application of the use tactic, the interface prints the new goals which are generated.

```
MILAWA !>(%use (%instance (%thm equal-of-lookups-when-submapp)
                          (a (cdr (submapp-badguy x z)))
                          (x x)
                          (y y)))
One goal remains.

 1. (IMPLIES
     (AND
      (IF
       (IF (EQUAL (NOT (SUBMAPP X Y)) 'NIL) 'NIL 'T)
       'T
       (IF (IF (EQUAL (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X))
                      'NIL)
               'NIL
               'T)
           'T
           (IF (EQUAL (EQUAL (EQUAL
                               (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                               (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y))
                             'T)
                      'NIL)
               'NIL
               'T)))
      (NOT (SUBMAPP X Z))
      (SUBMAPP Y Z)
      (SUBMAPP X Y)
      (NOT (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                  (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))))
     (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)))


MILAWA !>(%use (%instance (%thm equal-of-lookups-when-submapp)
                          (a (cdr (submapp-badguy x z)))
                          (x y)
                          (y z)))
One goal remains.
```

```
1. (IMPLIES
    (AND
     (IF
      (IF (EQUAL (NOT (SUBMAPP Y Z)) 'NIL) 'NIL 'T)
      'T
      (IF (IF (EQUAL (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y))
                     'NIL)
              'NIL
              'T)
          'T
          (IF (EQUAL (EQUAL (EQUAL
                              (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y)
                              (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))
                             'T)
                     'NIL)
              'NIL
              'T)))
     (IF
      (IF (EQUAL (NOT (SUBMAPP X Y)) 'NIL) 'NIL 'T)
      'T
      (IF (IF (EQUAL (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X))
                     'NIL)
              'NIL
              'T)
          'T
          (IF (EQUAL (EQUAL (EQUAL
                              (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                              (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y))
                             'T)
                     'NIL)
              'NIL
              'T)))
     (NOT (SUBMAPP X Z))
     (SUBMAPP Y Z)
     (SUBMAPP X Y)
     (NOT (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                 (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))))
    (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)))
```

At this point, we might call upon the waterfall to finish the proof, but instead, for exposition, we will use the split tactic to clean up the messy if-expressions and show a split into two subgoals.

```
MILAWA !>(%split)
; Splitting clause 1.
Two goals remain.
 1. (IMPLIES
     (AND (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y))
          (EQUAL (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                        (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y))
                 'T)
          (NOT (SUBMAPP X Z))
          (SUBMAPP Y Z)
          (SUBMAPP X Y)
          (NOT (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                      (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))))
     (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)))

 2. (IMPLIES
     (AND (EQUAL (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y)
                        (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))
                 'T)
          (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y)
          (EQUAL (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                        (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Y))
                 'T)
          (NOT (SUBMAPP X Z))
          (SUBMAPP Y Z)
          (SUBMAPP X Y)
          (NOT (EQUAL (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)
                      (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) Z))))
     (NOT (LOOKUP (CDR (SUBMAPP-BADGUY X Z)) X)))
```

The proof may now be easily finished by rewriting. We could do this using the waterfall, but to show a different tactic we instead use the rewriter. We do this via the `%crewrite` command, which we give the name of the theory to use.

```
MILAWA !>(%crewrite default)
; Rewrote clause #2 in 0.00 seconds (proved), 0.00% assm 0.00% rw
; Rewrote clause #1 in 0.71 seconds (proved), 0.00% assm 100.00% rw
(RW.CREWRITE-CLAUSE-LIST ...) took 721,220 microseconds
  (0.721220 seconds) to run with 8 available CPU cores.
  During that period,
    708,045 microseconds (0.708045 seconds) were spent in user mode
    12,000 microseconds (0.012000 seconds) were spent in system mode
    12,064,560 bytes of memory allocated.
    2,947 minor page faults, 0 major page faults, 0 swaps.
; Rewrote 2 clauses; 0 remain (0 forced)
All goals have been proven.
```

Some amount of work is necessary to support the features shown in the above transcript. For instance, global variables (in the form of ACL2 `tables`) are used to store default parameters for tactics such as `split-all`, and a world structure for the rewriting tactics. To manage the proof itself, another variable stores the evolving proof skeleton. Commands like `%use` are implemented as macros which apply the desired tactic to this skeleton.

11.2 ACL2 Connection

The above discussion has not covered how we arrived at the initial goal, nor what we do after the proof has been completed.

Using the `make-event` facility of ACL2, we can inspect the ACL2 state, which includes the lemmas and definitions which have been accepted by ACL2. We take advantage of this capability to avoid duplicating the definitions and lemmas in our Milawa proof scripts.

For function definitions, we provide a user-interface command called `%auto-admit`. When the user types `(%autoadmit` *fn*`)`, where *fn* is the name of an ACL2 function, we look up the ACL2 definition of *fn* and also the measure which was used to admit *fn*. If *fn* is recursive, we compute its termination obligations and put them into the skeleton for the user to prove. After the user has completed the proofs, we can add the definition of *fn* to a table that records the definitions that have been accepted by Milawa.

In certain cases, the ACL2 definition of a function does not precisely match its Milawa definition. One example of this is in the function `<<`. In ACL2, there are additional data types (such as characters and strings) which are not found in Milawa, and there is no closed universe axiom. Because of this, certain theorems of Milawa, such as the transitivity and trichotomy rules for `<<`, would not hold for all ACL2 objects. To address this difference, our ACL2 definition of `<<` includes a special case for non-Milawa objects.

To prevent such a difference from causing problems for `%autoadmit`, we allow our ACL2 definitions to be annotated with an alternate form for use in Milawa, via a new "xarg" called `:export`. The ACL2 definition of `<<` is shown below.

ACL2 Code
```
(defun << (x y)
  (declare (xargs :guard t
                  :export
                  ;; Definition for Milawa.
                  (cond ((natp x)
                         (if (natp y) (< x y) t))
                        ((natp y)
                         nil)
                        ((symbolp x)
                         (if (symbolp y) (symbol-< x y) t))
                        ((symbolp y)
                         nil)
                        (t
                         (if (equal (car x) (car y))
                             (<< (cdr x) (cdr y))
                           (<< (car x) (car y)))))))
;; Definition for ACL2.
  (cond ((natp x)
         (if (natp y) (< x y) t))
        ((natp y)
         nil)
        ((symbolp x)
         (if (symbolp y) (symbol-< x y) t))
        ((symbolp y)
         nil)
        ;; Special case for ACL2 compatibility
        ((or (not (consp x))
             (not (consp y)))
         (if (consp x)
             nil
           (if (consp y)
               t
             ;; ACL2's usual total order
             (and (ACL2::lexorder x y)
                  (not (equal x y))))))
        (t
         (if (equal (car x) (car y))
             (<< (cdr x) (cdr y))
           (<< (car x) (car y))))))
```

ACL2 also includes a variety of functions which cause side-effects that are invisible from a logical perspective. For instance, the ACL2 function `fmt-to-comment-window` may be used to print a message to standard output, but an ACL2 axiom is that `(fmt-to-comment-window ...)` is equal to `nil`. Other notable side-effecting functions include `time$`, which may be used to report timing information, `ec-call`, which can be used to suppress ACL2's guard-verification mechanism, `er`, which can be used to generate run-time errors, and `prog2$`, which is used purely to cause side-effects, e.g., via `fmt-to-comment-window` or `er`.

We sometimes make use of these functions, particularly `fmt-to-comment-window` and `time$`, to provide commentary about how many goals remain or how long various operations are taking to execute. We also use them to print warnings when, e.g., the stack depth is exhausted in our evaluator, or when the rlimit is exhausted during rewriting. These side-effecting functions are eliminated by `%autoadmit`, and are not part of our Milawa definitions.

To load ACL2 theorems, we implement a command called `%autoprove`. When the user types (`%autoprove` *thm*), where *thm* is the name of an ACL2 theorem, we look up the formula associated with the theorem, and load it into the skeleton as a goal for the user to prove. Once the user has completed the proof, we can create a Milawa rewrite rule from the theorem, and add it to our theory.

There are many subtleties to importing theorems from ACL2. First, there are a few simple matters of syntax. ACL2 embeds forcing annotations and syntactic restrictions directly into the formula, whereas we use annotations on our hypothesis structures. ACL2 stores the backchain limits for its hypotheses in a separate list, while we store them in each hypothesis structure. Finally, ACL2 prevents loops in rules such as the commutativity of `+` by annotating its rules with a separate `:loop-stopper` field, whereas we handle this using syntactic restrictions involving `logic.term-<`. It is not difficult to write the appropriate transformations to deal with these differences.

All rewrite rules in ACL2 are inside-out, but Milawa also supports outside-in rewriting. So, when `%autoprove` is used to import a rule from ACL2, we examine the rule to see if it would make a good outside-in rule. Inside-out rules are created in all cases, but we also create an outside-in rule when the right-hand side never increases the duplicity of a variable, there are no hypotheses, and there are no syntactic restrictions.

What motivates these criteria? The main advantage of outside-in rewriting is that we can sometimes avoid rewriting subterms. For instance, we can rewrite `(car (cons x y))` to x without ever examining y. On the other hand, a rule which duplicates a variable could lead us to examine a large term twice. For hypotheses, we do not want to examine unsimplified terms repeatedly during backchaining. For syntactic restrictions, our main motivation is for the user to be able to assume that the terms they are examining with syntactic restrictions have already been simplified. For instance, if we rewrite x to x' and y to y', we may often find that x' and y' are not in the same term order as x and y.

At any point in an ACL2 session, there is an implicit, default theory. Each rewrite rule (and definition) is either "enabled" (part of the theory) or "disabled" (not part of the theory), and the rewriter is only allowed to use enabled rules. Our user interface manages a similar theory named `default`. When definitions and rules are added with `%autoadmit` and `%autoprove`, we add them to the default theory only when they are enabled in ACL2. We adopt the good practice of avoiding non-local theory changes in our ACL2 proof scripts, and similarly we avoid non-local theory changes to the `default` theory in our Milawa proofs. Together, this discipline keeps our `default` theory in sync with ACL2's theory automatically.

The use of `%autoadmit` and `%autoprove` means that definitions and theorems in our Milawa proofs are automatically updated as we make changes to their ACL2 counterparts. But we might also add a theorem to our ACL2 proof script and forget to add the corresponding theorem to Milawa. Matt Kaufmann was able to develop a tool, now distributed with ACL2 as `misc/book-thms`, which produces the names of all the theorems introduced in an ACL2 book, even when they are generated by macros or `make-event` commands. We use this tool to ensure our Milawa files are complete with respect to the ACL2 files they are reimplementing.

An interesting consequence of writing our user interface within ACL2 is that when we get stuck in a Milawa proof, we can ask ACL2 to prove the same goal to see whether it is successful and what it tries to do. This was frequently useful when we were having difficulty translating a proof, especially to ensure no inappropriate forcing had led us to goals that ACL2 could not see how to prove. Our interface automates this process with the `(%check)` command, which asks ACL2 to prove each outstanding goal and displays ACL2's attempt to the user.

11.3 Proof-Checking Support

The tables used by our interface uses to store proof skeletons, rules, definitions, etc., are not protected by ACL2 in any way, so the user may freely and unsafely redefine functions, add rules, and so on. We make no claims that the interface obeys the rules of our logic, and one should only trust the proofs it finds after they have been checked by the system developed in Chapter 4.

To facilitate this checking, our interface can be used to build fully expansive versions of the proofs it finds by simply compiling the skeletons obtained at the completion of each proof attempt. These full proofs can be saved into files using the compacting printer introduced by Boyer and Hunt [16], which writes objects using the `#1=...`-style abbreviations supported by our file reader. The use of abbreviations helps to reduce disk space needs, and also reduces the amount of memory needed by the proof checker to read the proof.

In addition to saving proofs for later checking, our interface can run `logic.proofp` directly on proofs as they are constructed. To support this, we "emulate" the behavior of the proof checker by keeping a list of axioms, a list of theorems, and an arity table as global variables and by extending these lists as new definitions and theorems are processed. We normally do not use check proofs as we build them since doing so adds a significant amount of time to our Milawa scripts, but the ability to do so is sometimes useful as a sanity check after making changes to our interface code.

Our interface also keeps track of the Milawa definitions and theorems that have been submitted. This history can be conveniently written into a file of `DEFINE`, `VERIFY`, `SKOLEM`, and `FINISH` commands for the core proof checker to process.

11.4 Rewriter Debugging

Our user interface also includes two important debugging features. The first of these is a mechanism for profiling the operation of the rewriter, and is styled after ACL2's `accumulated-persistence` feature.

We implement our profiler by using the `advise` feature of Clozure Common Lisp, which allows us to add functionality around CRW without needing to explicitly redefine it. Similar functionality is available in many other Lisp systems, but is not part of the Common Lisp standard. The goal is to associate, with each rule we backchain through,

– *frames*, the number of stack frames generated because of this rule,

– *tries*, the number of times the rule was explicitly tried, and

– *successes*, the number of times the rule was applied successfully.

A naive implementation of profiling would be as follows: each time a rule is tried, increment the frames for every active rule in the call stack, and increment the tries for this particular rule; then, if the rule applies, increment its successes.

To make this more efficient, we use two data structures. First, we have a *stored costs table*, which is a hash table mapping rule names to tuples of the three counts. Next, we have an *active rules stack*, which is a list of the form

$$[(\mathit{frames}_1\ .\ \mathit{name}_1), \ldots, (\mathit{frames}_n\ .\ \mathit{name}_n)],$$

The idea is that each *frames*$_i$ is some number of frames which *name*$_i$ should be blamed for, but which have not yet been added to the stored costs table. As the rewriter operates, we update these structures as follows. Every time we begin working to apply a new rule, *name*, we push `(1 .` *name*`)` onto the active rule stack. Now, suppose we are about to return from the `rule` mode of CRW, and that our active rules stack is [`(`*frames*$_1$ `.` *name*$_1$`)`, . . . , `(`*frames*$_n$ `.` *name*$_n$`)`]. Here,

– we increment the tries associated with *name₁* in the stored costs table, to record this attempt to use *name₁*,

– we increment the frames associated with *name*$_1$ in the stored costs table by *frames*$_1$,

– if $n \geq 2$, we increment *frames*$_2$ by *frames*$_1$, so that later, when its frame is popped, *name*$_2$ will also be blamed for the frames we have just assigned to *name*$_1$, and

– we pop `(`*frames*$_1$ `.` *name*$_1$`)` from the active rules stack.

This approach allows us to avoid traversing the list of active rules to count the frames for each rule, considerably reducing the overhead of profiling.

The profiler may be enabled and disabled using the commands `(%profile)` and `(%profile.stop)`. After profiling has been enabled, the rewriter is used as normal, e.g., the user might run `(%crewrite ...)` or `(%waterfall ...)`, or may even run whole files of proofs. Finally, to see the results of profiling, the user writes `(%profile.report)`. These results are cumulative until `(%profile.clear)` is run or profiling is stopped, so incremental reports can be viewed during interrupts.

An example of a profiling report, generated for our proof of the theorem `nth-of-first-index-of-domain-and-range`, is shown below. (We have compressed the report so it will fit within the margins.) When reading these reports, one is typically looking for rules which are rarely or never successful but which are responsible for a high number of frames. These rules are slowing down the rewriter without contributing to any progress.

```
MILAWA !>(%profile.report)
The following statistics were gathered since the last (%profile) or
(%profile.clear) was issued.

Rewrite Rule Report

Cache hit rate: 22% (1,199 hits in 5,216 tries)

In the following table,
  - "Success" counts how many times all the hyps were relieved.
  - "Frames" counts how many rules were tried due to this rule
    backchaining.
  - "Tries" counts how many times this rule itself was tried.
  - "Ratio" is the average number of frames per try.
  - A star indicates this rule can cause case splits.

Success   Tries  Frames    Ratio  Rule
      1      57     795    13.94  NTH-WHEN-ZP
      0      98     589      6.1  NOT-EQUAL-WHEN-LESS
      0      98     552     5.63  NOT-EQUAL-WHEN-LESS-TWO
      0      98     547     5.58  SAME-LENGTH-PREFIXES-EQUAL-CHEAP
      0      54     487      9.1  CONSP-OF-CDR-WHEN-LEN-TWO-CHEAP
      0      57     432     7.57  NTH-WHEN-INDEX-TOO-LARGE
      0      52     411     7.90  TRICHOTOMY-OF-<
      . . . and so on . . .
      1       1       1      1.0  REFLEXIVITY-OF-EQUAL
      1       1       1      1.0  CONS-UNDER-IFF
      1       1       1      1.0  IFF-OF-T-LEFT
      1       1       1      1.0  [OUTSIDE]EQUAL-OF-ZERO-AND-NFIX

Useless, Expensive Rules

The following rules were never successful and each took over 100
frames.  To speed up your rewriting, you may wish to consider
disabling them:

(NOT-EQUAL-WHEN-LESS NOT-EQUAL-WHEN-LESS-TWO
                     SAME-LENGTH-PREFIXES-EQUAL-CHEAP
```

```
CONSP-OF-CDR-WHEN-LEN-TWO-CHEAP
```
... and so on ...
```
CONSP-OF-CDR-WHEN-TUPLEP-3-CHEAP
CONSP-OF-CDR-WHEN-TUPLEP-2-CHEAP).
```

Another useful tool is a rewrite-loop debugger, similar to the :cw-gstack command in ACL2, which we implement in our interface by redefining functions. The rewriter is modified so that when the rewrite limit is reached, a warning is printed and a global variable, `*rw.rlimit-was-reached*`, is set to `t`. Our rule-trace constructor is then modified so that whenever this flag is `t`, as a side-effect it prints the name of the rule it is using and the result of applying the rule. Finally, in the rewriter, as the backchaining unwinds, we set `*rw.rlimit-was-reached*` back to `nil` when the rlimit reaches 5. The net effect is that rules used "near the rlimit" are printed. Usually these are the rules responsible for the loop.

All of this happens automatically, and the user only needs to intervene if he interrupts the rewriter before the `*rw.rlimit-was-reached*` flag is set back to `nil`. To show the loop-debugger in use, it is easy to set up rules which loop with one another.

Rule:
```
(equal (app (app x y) z)
       (app x (app y z)))
```

Rule:
```
(equal (app x (app y z))
       (app (app x y) z))
```

If these rules are both part of the theory, then goals involving multiple `app` terms will cause loops. To see the loop debugger work, we set up the following goal. When we rewrite it, the problematic rules become obvious. In this case, the loop does not prevent us from proving the goal, but even so, such loops may dramatically slow the rewriter.

```
One goal remains.

 1. (EQUAL (APP A (APP B (APP C D)))
           (APP A (APP (APP B C) D)))


MILAWA !>(%crewrite default)
WARNING: rlimit exhausted - the rewriter may be looping!
Be sure to run (rw.stop-loop-debugging) if you interrupt!
APP-OF-APP-ALT: (APP (APP A (APP B C)) D)
[OUTSIDE]APP-OF-APP: (APP A (APP (APP B C) D))
[OUTSIDE]APP-OF-APP: (APP B (APP C D))
APP-OF-APP-ALT: (APP (APP B C) D)
[OUTSIDE]APP-OF-APP: (APP B (APP C D))
APP-OF-APP-ALT: (APP (APP B C) D)
[OUTSIDE]APP-OF-APP: (APP B (APP C D))
APP-OF-APP-ALT: (APP (APP B C) D)
[OUTSIDE]APP-OF-APP: (APP B (APP C D))
APP-OF-APP-ALT: (APP (APP B C) D)
[OUTSIDE]APP-OF-APP: (APP B (APP C D))
APP-OF-APP-ALT: (APP (APP B C) D)
[OUTSIDE]APP-OF-APP: (APP B (APP C D))
... and so on ...
; Rewrote clause #1 in 0.46 seconds (proved), 0.00% assm 100.00% rw
(RW.CREWRITE-CLAUSE-LIST ...) took 473,028 microseconds
  (0.473028 seconds) to run with 8 available CPU cores.
  During that period,
    444,028 microseconds (0.444028 seconds) were spent in user mode
    20,002 microseconds (0.020002 seconds) were spent in system mode
    7,513,792 bytes of memory allocated.
    4,312 minor page faults, 0 major page faults, 0 swaps.
; Rewrote 1 clauses; 0 remain (0 forced)
All goals have been proven.
```

11.5 Parallelism

Although there is experimental support for some parallelism in ACL2 [73], these features cannot be combined with the hash-consing extension [16] which we use for our proof printing and rewriter caching. As a result, our user interface is a single-threaded program which cannot take advantage of multiple processors when searching for a proof.

Despite this, we can still achieve parallelism when rebuilding our proofs by splitting up our work into many files, and invoking separate instances of our user interface to process each file. In ACL2, such files are called *books*, and the act of processing a book is called *certification*. Our files of interface instructions are also valid ACL2 books, and can be processed with ACL2's normal certification mechanisms.

But ACL2's approach to certification imposes significant limits on parallelism, because books must be certified in the order of their dependencies. To avoid this limitation, we have developed an alternate approach, called "provisional certification," which allows our books to be processed in any order. When a book that has not yet been certified is included, we simply add its definitions and theorems without checking their proofs.

Our build system takes advantage of this, splitting up the work of rebuilding our proofs across eight machines, each of which has eight processors. The ability to rebuild proofs more quickly has been quite valuable, particularly when we made updates to our ACL2 scripts.

Chapter 12

Bootstrapping

With our tactics written and our user-interface in place, we are ready to begin the work of translating ACL2 proofs into Milawa proofs. This process is made more difficult by our desire to emit proofs that can be checked in a reasonable amount of time by the core proof checker.

What is reasonable? Until we started generating proofs, we had no intuition for how large the translated proofs would be or how long they would take to check. Our proof checker, `logic.proofp`, is implemented quite simply and has a number of obvious inefficiencies. For instance, to check axiom and theorem steps, we perform a linear search of the current `*axioms*` and `*theorems*` for the formula being used. Arity checking is also particularly expensive, and requires a linear search through an association list for every function named throughout a term or formula.

On the other hand, there are also some good aspects of the proof checker's performance. Excessive consing is often a cause of slowness in Lisp programs, but most of our proof-step checking functions do little consing. Another important note is that, in practice, all of the proofs we will check are written to disk using Boyer and Hunt's compacting printer [16], which uses `#1=...`-style abbreviations to refer to structures which are repeated many times, such as formulas. This has some obvious benefits for disk space, memory usage, and memory locality, but it also means that many equality checks, such as those used to compare parts of formulas, may be settled by pointer comparison rather than by a deep, structural equality check.

As a simple way to measure the size of the proofs being generated, we began to use the `rank` function to count the number of conses. We adopt the ordinary SI prefixes, so a *kilocons* (KC) means a thousand conses, a *megacons* (MC) means a million conses, and a *gigacons* (GC) means a billion conses.

During our proof development, the computer we have mainly used is named Lhug-3. This computer has four 2.2 GHz AMD Opteron 850 processors, 32 GB of memory, and runs 64-bit Linux. We typically use Clozure Common Lisp (CCL). Early in the project, as a rule of thumb we estimated that checking a gigacons worth of proof steps on this machine would take about forty minutes. Since that time, we have made some efficiency improvements to avoid some unnecessary arity checking, and CCL has presumably improved. Today, a revised estimate is that `logic.proofp` can check about a gigacons of proof every 7 minutes on the same machine.

At any rate, we set a rather arbitrary goal that no individual proof should be larger than 500 megaconses. Toward this goal, we implemented a proof-size check in our user interface so that, upon building each proof, a size-check is performed. If the proof is larger than our threshold of 500 megaconses, an error is caused. In such cases, the user can either attempt to make the proof smaller (e.g., by using a different proof strategy, by improving the efficiency of the proof-building functions being used, or by introducing intermediate proof checkers that can check smaller proofs), or can choose to accept the large proof by increasing the threshold with a `%max-proof-size` command.

12.1 Level 2 – Propositional Rules

As we began to build proofs, we were able to optimize many of our proof-building functions to emit smaller proofs. Many aspects of this can be seen throughout the previous chapters. For instance, we favor "custom" derivations for propositional manipulations rather than using our tautology rule, even optimizing the generic subset rule for special, common cases; we introduce and instantiate formal theorems to reduce the number of steps needed in derivations, as first described in Section 6.1; we develop the update clause and CS-AUX routines in tail-recursive style so that the proof grows only linearly in the size of the clause; and we keep the hypotheses for our assumptions structure partitioned so that we may perform only minimal propositional manipulation while rewriting clauses.

Even so, as we began translating more difficult proofs, it became apparent that fully expansive proofs for our more sophisticated algorithms would not be practical. Instead, we set upon the goal of verifying a series of increasingly powerful proof checkers, which could accept smaller proofs. Although the proof checking program we developed in Chapter 4 initially requires us to prove each new theorem we introduce using `logic.proofp`, it also allows us to begin using a new proof checker, via the `SWITCH` command, after we have shown the fidelity of the new proof checker.

The first of our new proof checkers is named `level2.proofp`. What kinds of proofs should `level2.proofp` accept? As a modest goal, we decided to try to implement all of the rules accepted by `logic.proofp`, and also all of the simple, non-recursive propositional rules which were introduced in Section 5.3, such as modus ponens and disjoined associativity. Later, to make `level2.proofp` slightly more capable, we decided to also add a few more rules of this variety, such as the aux split twiddle and aux split twiddle 2 rules. The hope was that these rules would be simple enough that a fully expansive proof of the fidelity of `level2.proofp` would be practical.

How is `level2.proofp` defined? In Section 3.5, we introduced step-checking functions for the basic rules of inference that are accepted by `logic.proofp`. Our first step is to write similar functions for the new kinds of proof steps we wish to support.

We begin with the commute or rule, which allows us to derive $B \vee A$ when given a proof of $A \vee B$. The function build.commute-or-okp determines if an appeal is a valid use of this rule of inference. To be valid, we say the appeal's method must be build.commute-or (named after the proof-building function for the commute or rule), it may have no extras, and it must have precisely one subproof with a conclusion of the appropriate form. Following our previous convention for arity checking, we assume the subproofs are well-formed with respect to the arity table, so in this case no additional arity checking is needed.

Definition: build.commute-or-okp
```
(pequal* (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)))))))
```

As another example, the right expansion rule allows us to derive $A \vee B$ when given a proof of A. The function build.right-expansion-okp determines if an appeal is a valid use of this rule of inference: the method must be build.right-expansion, there should be no extras, and there should be a subproof whose conclusion is appropriate. Since the B portion of the formula is new, we must check to ensure that it is well-formed with respect to the arity table.

Definition: `build.right-expansion-okp`

```
(pequal*
 (build.right-expansion-okp x atbl)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'build.right-expansion)
        (equal extras nil)
        (equal (len subproofs) 1)
        (equal (logic.fmtype conclusion) 'por*)
        (equal (logic.vlhs conclusion)
               (logic.conclusion (car subproofs)))
        (logic.formula-atblp (logic.vrhs conclusion) atbl))))
```

This continues through the other rules. As one more example, the disjoined associativity rule allows us to conclude $P \vee (A \vee (B \vee C))$ when given a proof of $P \vee ((A \vee B) \vee C)$. Here, checking whether the conclusions line up appropriately becomes somewhat elaborate. As a programming aide, we generate these step-checking functions using an ACL2 macro which allows us to describe the formulas involved using a simple pattern-matching style.

Definition: `build.disjoined-associativity-okp`

```
(pequal*
 (build.disjoined-associativity-okp x)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
      (and
       (equal method 'build.disjoined-associativity)
```

```
        (equal extras nil)
        (equal (len subproofs) 1)
        (let ((subconc (logic.conclusion (car subproofs))))
          (and
           (equal (logic.fmtype conclusion) 'por*)
           (equal (logic.fmtype (logic.vrhs conclusion)) 'por*)
           (equal (logic.fmtype (logic.vlhs (logic.vrhs conclusion)))
                  'por*)
           (equal (logic.fmtype subconc) 'por*)
           (equal (logic.fmtype (logic.vrhs subconc)) 'por*)
           (equal (logic.fmtype (logic.vrhs (logic.vrhs subconc)))
                  'por*)
           (equal (logic.vlhs conclusion) (logic.vlhs subconc))
           (equal (logic.vlhs (logic.vlhs (logic.vrhs conclusion)))
                  (logic.vlhs (logic.vrhs subconc)))
           (equal (logic.vrhs (logic.vlhs (logic.vrhs conclusion)))
                  (logic.vlhs (logic.vrhs (logic.vrhs subconc))))
           (equal (logic.vrhs (logic.vrhs conclusion))
                  (logic.vrhs (logic.vrhs (logic.vrhs subconc))))
           )))))
```

These new functions are similar in complexity and execution time to our primitive proof checking functions for steps like Cut and Associativity. But a single disjoined associativity step for `level2.proofp` can do the work of 126 primitive steps for `logic.proofp`. The savings for the other propositional rules are more modest, but are realized every time one of these steps is used. As a result, proofs written for `level2.proofp` can be much shorter than proofs for `logic.proofp`. (Some comparisons are presented in Section 12.11.)

After we have introduced a checking function for each new kind of proof step, we implement a new analogue of `logic.appeal-step-okp` which can accept any Level 2 proof step. Note that after checking for each new kind of step, we fall back to logic.appeal-step-okp, so any primitive step is also accepted as a Level 2 step. Throughout our progression of proof checkers, each new level is "purely additive" and will accept proof steps from any of the previous levels.

Definition: level2.step-okp
```
(pequal* (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))
                 ((equal method 'build.modus-ponens)
                  (build.modus-ponens-okp x))
                 ((equal method 'build.modus-ponens-2)
                  (build.modus-ponens-2-okp x))
                 ((equal method 'build.right-associativity)
                  (build.right-associativity-okp x))
                 ... and so on ...
                 (t
                  (logic.appeal-step-okp x axioms thms atbl)))))
```

The definition of level2.proofp follows very closely with logic.proofp, simply substituting level2.step-okp for logic.appeal-step-okp.

Definition: level2.flag-proofp
```
(pequal* (level2.flag-proofp flag x axioms thms atbl)
         (if (equal flag 'proof)
             (and (level2.step-okp x axioms thms atbl)
                  (level2.flag-proofp 'list (logic.subproofs x)
                                      axioms thms atbl))
           (if (consp x)
               (and (level2.flag-proofp 'proof (car x) axioms thms
                                        atbl)
                    (level2.flag-proofp 'list (cdr x) axioms thms
                                        atbl))
             t)))
```

Definition: `level2.proofp`

```
(pequal* (level2.proofp x axioms thms atbl)
         (level2.flag-proofp 'proof x axioms thms atbl))
```

Before we can use the SWITCH command to begin using `level2.proofp` in our proof checking system, we must show that it only accepts provable formulas. The fidelity claim for `level2.proofp` is the following formula.

```
(por* (pequal* (logic.appealp x) nil)
      (por* (pequal* (level2.proofp x axioms thms atbl) nil)
            (pnot* (pequal* (logic.provablep (logic.conclusion x)
                                             axioms thms atbl)
                            nil))))
```

With our previous work in place, this proof is easy. To begin with, for each individual step-checking function we prove that if (1) the appeal is accepted by this step checker, and (2) all of the formulas for the subproofs are provable, then the conclusion of the appeal is also provable. For instance, for `build.commute-or-okp`, this lemma is:

ACL2 Code

```
(defthm fidelity-of-build.commute-or-okp
  (implies (and (logic.appealp x)
                (build.commute-or-okp x)
                (logic.provable-listp (logic.strip-conclusions
                                        (logic.subproofs x))
                                       axioms thms atbl)))
           (logic.provablep (logic.conclusion x) axioms thms atbl)))
```

These lemmas are easy corollaries of the three theorems for each builder function. That is, since the subproofs are provable, let p be a proof the first subproof. By the definition of `build.commute-or-okp`, we can see that p has a conclusion of the form $A \vee B$, while the conclusion of x is $B \vee A$. Let q be the result of (build.commute-or p). By the relevance theorem for build.commute-or, we can see that the conclusion of q is $B \vee A$. Meanwhile, by the faithfulness theorem for build.commute-or, we can see that q is accepted by logic.proofp. Hence, q is a proof of the conclusion of x, which is what we wanted to show.

We carry out a similar argument for each of our new step checkers, then compose the results to obtain a fidelity lemma for level2.step-okp:

ACL2 Code
```
(defthm fidelity-of-level2.step-okp
  (implies (and (logic.appealp x)
                (level2.step-okp x axioms thms atbl)
                (logic.provable-listp (logic.strip-conclusions
                                       (logic.subproofs x))
                                      axioms thms atbl))
           (logic.provablep (logic.conclusion x) axioms thms atbl)))
```

The fidelity of level2.proofp then follows by simple induction. In our ACL2 proof sketch, the theorem is expressed as follows.

ACL2 Code
```
(defthm logic.provablep-when-level2.proofp
  (implies (and (logic.appealp x)
                (level2.proofp x axioms thms atbl))
           (logic.provablep (logic.conclusion x) axioms thms atbl)))
```

We regard our translated proof of this theorem to be 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.

How large is the proof? In our ACL2 proof sketch, we have a directory named utilities which contains definitions and theorems for simple functions about arithmetic, lists, etc., and another directory named logic which includes our definitions of logical concepts such as terms, formulas, and provability. Many of these definitions and theorems are probably not necessary in order to justify `level2.proofp`, but rather than try to determine exactly which lemmas are needed, we carry out all of the proofs in these directories at the primitive level. We then admit the definitions and proofs for the builder functions and step checkers leading up through `level2.proofp`. In total, there are 421 definitions and 4,188 theorems. All together, the proofs come to 18.4 gigaconses, with the largest individual proof at 353.8 megaconses. On disk, this comes to 1.7 GB using the compacting printer (without using an external compression program).

12.2 Level 3 – Basic Functions

With the Level 2 proof checker now verified, we can use the `SWITCH` command so that our proof checking program will use `level2.proofp` to check proofs instead of `logic.proofp`.

We would now like to return to our ACL2 proof plan. But now, instead of writing fully expansive proofs, we would like to retarget our proof-building functions so that they produce Level 2 proofs. For instance, everywhere we have previously called upon `build.commute-or` to construct a two-step, fully expansive proof that can be accepted by `logic.proofp`, we would now like to call upon a new function, say `build.commute-or-high`, which builds a higher-level, one-step proof that `level2.proofp` can accept.

Definition: `build.commute-or-high`
```
(pequal* (build.commute-or-high x)
         (logic.appeal 'build.commute-or
                       (logic.por (logic.vrhs (logic.conclusion x))
                                  (logic.vlhs (logic.conclusion x)))
                       (list x)
```

```
                              nil))
```

In our user interface, a simple way to accomplish this is simply to redefine `build.commute-or` as an alias to `build.commute-or-high`. Then, automatically, the builder functions for algorithms such as our clause splitter, rewriter, and so on, will begin to emit Level 2 proofs. From the perspective of ACL2, in which we have written our interface, this sort of redefinition is inconsistent: after we redefine `build.commute-or`, its "logical definition" will no longer agree with its "executable definition," and we could exploit this to "prove `nil`" in ACL2. But at this point in the project, we have already completed our ACL2 proofs, and the correctness of ACL2 is no longer of any consequence. At any rate, we are still using Milawa's algorithms to find the proofs of these theorems—we are simply using redefinition as a convenient trick to emit higher-level proofs.

What kinds of proof steps should the Level 3 proof checker accept? The idea is to add rules that will allow Level 3 proofs to skip many proof steps, and yet which are not too hard to verify with the Level 2 proof checker.

To help identify such rules, we first wanted to identify which tactics would be most important to optimize. We instrumented our user interface so that the skeleton compiler prints messages explaining the incremental cost of compiling each step. For instance, some representative output is:

```
All goals have been proven.
Compiling skeleton for LOGIC.FORMULA-LIST-ATBLP-OF-LIST-FIX.
; SPLIT-ALL.  Incremental Cost: 174,086.  Total cost: 174,086
; CLEANUP.  Incremental Cost: 69,844.  Total cost: 243,930
; UREWRITE-ALL.  Incremental Cost: 397,124.  Total cost: 641,054
; CLEANUP.  Incremental Cost: 8,184.  Total cost: 649,238
; ELIM-ALL.  Incremental Cost: 828,960.  Total cost: 1,478,198
; SPLIT-ALL.  Incremental Cost: 213,780.  Total cost: 1,691,978
```
. . . and so on . . .

After examining this output for many proofs, it became clear that the `cleanup`, `split`, and rewriting tactics were the largest contributors to proof size. To decide which rules to add to Level 3, we outlined the control flow for the builder functions associated with these tactics. For instance, within `rw.compile-trace`, the builder function for our rewriters, we wrote down the following call tree.

```
- rw.compile-trace
  - rw.compile-trace-step
    - rw.compile-fail-trace
      - build.iff-reflexivity
      - build.equal-reflexivity
    - rw.compile-transitivity-trace
      - build.transitivity-of-iff
      - build.transitivity-of-equal
      - build.disjoined-transitivity-of-iff
      - build.disjoined-transitivity-of-equal
    - rw.compile-equiv-by-args-trace
      - build.iff-from-equal
      - build.disjoined-iff-from-equal
      - build.equal-by-args
      - build.disjoined-equal-by-args
    - rw.compile-lambda-equiv-by-args-trace
      - build.iff-from-equal
      - build.disjoined-iff-from-equal
      - build.lambda-equal-by-args
      - build.disjoined-lambda-equal-by-args
```

. . . and so on . . .

The leaves of these trees became candidates for inclusion in Level 3. In the end, we implement the recursive propositional rules (e.g., modus ponens list, generic subset, etc.) and also most of the non-recursive rules about =, `equal`, `if`, `iff`, and `not` which are covered in Sections 6.1, 6.2, and 7.3.

Unlike the propositional rules which we added in Level 2, these rules refer to particular functions like `equal` which are expected to have certain arities, and make use of axioms and theorems. For instance, to make build.equal-reflexivity produce smaller fully expansive proofs, we first introduced a theorem to capture the reflexivity of equal, and this theorem is instantiated by build.equal-reflexivity. Because of this, proofs constructed by build.equal-reflexivity are only valid in histories where this theorem has been established. In our ACL2 proof sketch, we can see this as a hypothesis in the faithfulness theorem:

ACL2 Code
```
(defthm logic.proofp-of-build.equal-reflexivity
  (implies (and (logic.termp a)
                (logic.term-atblp a atbl)
                (equal (cdr (lookup 'equal atbl)) 2)
                (memberp (theorem-reflexivity-of-equal) thms))
           (logic.proofp (build.equal-reflexivity a)
                         axioms thms atbl)))
```

When it comes time to check high-level build.equal-reflexivity proof steps, we have some options for addressing this. A simple approach would be to have build.equal-reflexivity-okp ensure that that (theorem-reflexivity-of--equal) is a member of the current theorems every time it is used. But this would impose a linear search of the theorems during every use of the rule.

A more efficient alternative, which is also easy to implement, is to move the check into level3.proofp, itself. That is, before the actual steps of the proof are checked, we can ensure that all of the necessary axioms and theorems are part of the history. Then, within build.equal-reflexivity-okp, we may simply assume that the theorem is available. This way, the cost is incurred only once per proof, no matter how many times the rule is used. Following this approach, we implement our level3.proofp wrapper as follows:

Definition: `level3.proofp`
```
(pequal* (level3.proofp x axioms thms atbl)
(and (memberp (axiom-equal-when-diff) axioms)
     (memberp (axiom-equal-when-same) axioms)
     (memberp (axiom-if-when-not-nil) axioms)
     (memberp (axiom-if-when-nil) axioms)
     (memberp (axiom-reflexivity) axioms)
     (memberp (theorem-commutativity-of-pequal) thms)
     (memberp (theorem-substitute-into-not-pequal) thms)
     ... and so on ...
     (memberp (theorem-not-when-not-nil) thms)
     (memberp (theorem-iff-when-not-nil) thms)
     (equal (cdr (lookup 'not atbl)) 1)
     (equal (cdr (lookup 'equal atbl)) 2)
     (equal (cdr (lookup 'iff atbl)) 2)
     (equal (cdr (lookup 'if atbl)) 3)
     (level3.flag-proofp 'proof x axioms thms atbl)))
```

We call these one-time checks *static*, and say checking the main part of the proof is *dynamic*. Because of these static checks, in addition to having fewer steps to check than equivalent Level 2 proofs, Level 3 proofs may require considerably fewer lookups for these highly-used theorems and axioms.

A further optimization would be to eliminate the static theorem checks altogether by showing that each necessary theorem is provable from the axioms. The main obstacle to doing this is expanding away all of the uses of theorems from each proof, which could require some large evaluations to prove correct. We have not pursued this approach since the theorem checks are so inexpensive.

In total, there are 230 definitions and 815 theorems in Level 3. Together, the proofs come to 8.0 GC, or, when printed to disk, 201.0 MB. The largest individual proof, the faithfulness of the disjoined transitivity of equal rule, is 614.3 MC, and is the only proof which exceeds our goal of 500 megaconses.

12.3 Level 4 – Miscellaneous Groundwork

In Level 4, we add a hodgepodge of new proof steps, again motivated by inspecting the call trees for our major tactics. To improve clause updating, we add the substitute iff into literal rules and the aux update clause lemmas for `equal` and `iff`. To improve clause cleaning, we add the standardize negative and double negative terms, and the obvious term rule. To improve clause splitting, we add most of the supporting lemmas for CS-AUX, such as aux split positive, aux split negative, etc. To improve `if`-lifting, we add the factor and cases lemmas. Finally, to improve rewriting, we add the = by args rules (for functions), the ccstep lemmas, and the compile formula rule.

Taken all together, we introduce 168 definitions and 991 theorems. The combined proofs weigh in at 19.1 GC and take 288.1 MB when printed to disk. Eight proofs exceed our goal threshold of 500 megaconses. The largest proofs are of the faithfulness of the CS-AUX lemmas, and the largest individual proof is 1.2 GC.

12.4 Level 5 – Equivalence Traces, Updating Clauses

In Level 5, most of our improvements are aimed at improving rewriting. Many of the new rules we add are only small improvements for trace compilation, e.g., we add the rules used in our step compilers for our If False, If True, If General, and If Same traces. We also add the dual substitution and lambda = by args rules, which results in an important improvement for rewriting terms with lambda abbreviations.

Another improvement in Level 5 is the addition of our equivalence trace compiler from Section 8.3. We show the step-checking function for equivalence trace appeals, below. The equivalence trace to check is stored in the extras of the appeal. We ensure that the trace is structurally well-formed with `rw.eqtracep`, that it is a semantically valid trace (i.e., comprised entirely of legitimate Primary, Secondary, Direct iff, Negative Iff, Trans1, Trans2, Trans3, and Weakening steps) using rw.eqtrace-okp, and that its conclusion is correct.

Definition: rw.eqtrace-bldr-okp

```
(pequal* (rw.eqtrace-bldr-okp x atbl)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'rw.eqtrace-bldr)
                (tuplep 2 extras)
                (let ((trace (first extras))
                      (box   (second extras)))
                  (and (rw.hypboxp box)
                       (rw.hypbox-atblp box atbl)
                       (rw.eqtracep trace)
                       (rw.eqtrace-okp trace box)
                       (equal conclusion
                              (rw.eqtrace-formula trace box))
                       (not subproofs))))))
```

We think of rw.eqtrace-bldr steps as adapters that allow equivalence traces to be plugged into our appeal system. That is, in all of our previous step-checking functions, the conclusion is justified mainly by inspecting the validity of some subproofs, which are themselves appeals, with whatever appeal-checking function is appropriate for this level. But in an rw.eqtrace-bldr step, the "core" of the proof is instead stored in the extras, has a custom format which is not based on appeals, and is checked with specialized functions.

Aside from rewriting improvements, we also add new proof steps to perform update clause and update clause iff rules as single proof steps. This improves the sizes of proofs generated by clause cleaning and if-lifting. We also improve clause splitting by adding the remaining lemmas, aux split negative 1 and aux split negative 2, whose proofs were large enough that we left them out of Level 4.

Together, Level 5 adds 192 definitions and 1,071 theorems, totaling 9.3 GC (or 185.0 MB on disk). All of the proofs are under our 500 megacons goal.

12.5 Level 6 – Factoring and Splitting

In Level 6, an easy addition is the factor rule, which shortens the proofs created by `if`-lifting. But the major accomplishment at this level is the addition of our clause splitting algorithm, CS-AUX.

This was particularly challenging. When we presented CS-AUX in Section 7.5, we explained that the goal at each step is to prove the formula $(T_1 \vee \cdots \vee T_n) \vee (D_1 \vee \cdots \vee D_m)$, where the T_i are the term formulas for the todo literals, and the D_i are the term formulas for the done literals. But this notation hides the case split which is necessary to handle situations where one of n or m is 0. A precise description of our goal at each step is

```
(cond ((and (consp todo) (consp done))
       (logic.por (clause.clause-formula todo)
                  (clause.clause-formula done)))
      ((consp todo)
       (clause.clause-formula todo))
      (t
       (clause.clause-formula done))).
```

This case split is infectious. Each time we make a recursive call of CS-AUX to build a piece of the proof, we must consider the shape of the proof which has been produced in order to extend it properly. Because of this, our attempts to simply "push the proof through" were unable to produce a proof under 10 GC. In one particularly misguided effort, the proof reached 78 GC.

In retrospect, the solution was fairly obvious: contain the case split by wrapping it in a function that could be disabled. We introduced a new function, clause.-aux-split-goal, which produces the formula for the step goal when given the list of todo and done literals, and restated the theorems for each auxiliary rule in terms of this new function. This abstraction was quite effective: the ACL2 proof decreased from 80 seconds to 9 seconds, and our new Milawa proof is now at 759.0 MC.

We only add 55 definitions and 402 theorems at this level. The proofs total 3.3 GC (116.5 MB on disk). Only the faithfulness proof for CS-AUX exceeds our 500 MC goal.

12.6 Level 7 – Split Tactics

Level 7 adds a single and powerful rule: the clause.split routine introduced in Section 10.1, used by our split-first and split-all tactics. This requires us to also translate the proofs for our if-lifting and clause cleaning routines. We do not add a new kind of proof step for if-lifting, since we only make use of it through our splitting tactics. Also, we do not add a proof step for our clause cleaning routine: its processing of a list of clauses, rather than a single clause, cannot be easily accommodated by our appeal structures, which can have only a single conclusion. The new step-checking function for clause.split is shown below.

Definition: clause.split-bldr-okp
```
(pequal*
 (clause.split-bldr-okp x atbl)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'clause.split-bldr)
        (tuplep 4 extras)
        (let ((liftp      (first extras))
              (liftlimit  (second extras))
              (splitlimit (third extras))
              (clause     (fourth extras)))
          (and (natp liftlimit)
               (natp splitlimit)
               (logic.term-listp clause)
               (logic.term-list-atblp clause atbl)
               (true-listp clause)
               (consp clause)
               (equal conclusion (clause.clause-formula clause))
               (equal (clause.clause-list-formulas
                       (cdr (clause.split liftp liftlimit splitlimit
                                          clause)))
                      (logic.strip-conclusions subproofs))))))
```

This level adds 83 definitions and 749 theorems, totaling 949.5 MC and taking 44.0 MB on disk. None of the proofs exceed our goal of 500 megaconses.

12.7 Level 8 – Rewrite Traces

In Level 8, we add two rules. The first of these allows a rewrite trace to be used as a proof. The next allows CRW-CLAUSE to be used as a single step. Later, when we build Level 8 proofs using our interface, the CRW-CLAUSE rule is used to justify uses of CRW during our `crewrite-first`, `crewrite-all`, and `waterfall` tactics, and the rewrite trace rule is only used for proofs generated by our unconditional rewriter, URW.

Adding these rules requires a considerable amount of supporting work. In particular, we need to translate the ACL2 proofs that justify `rw.compile-trace` (Section 9.9), which in turn requires us to carry out the proofs for each kind of rewrite trace. Most of these proofs, for instance those for Failure and Transitivity traces, are quite straightforward. As a notable exception, to justify Evaluation traces we additionally need to carry out the proofs for our evaluator from Section 6.4.

Our rewrite trace rule is much like the equivalence trace rule in Level 5: the trace itself is stored in the extras of the appeal, and we check it for validity using optimized versions of our various well-formedness checks. (These optimizations greatly improve the speed of proof-checking for some particularly hard proofs in Level 10, and we discuss them in Section 12.9.) Our step-checking function is shown below.

Definition: `rw.compile-trace-okp`
```
(pequal* (rw.compile-trace-okp x defs thms atbl)
         (let ((method     (logic.method x))
               (conclusion (logic.conclusion x))
               (subproofs  (logic.subproofs x))
               (extras     (logic.extras x)))
           (and (equal method 'rw.compile-trace)
                ;; extras is a valid trace
                (rw.fast-tracep extras)
                (rw.fast-trace-atblp extras atbl)
                (rw.trace-okp extras defs)
                (rw.trace-env-okp extras defs thms atbl)
                ;; the trace has the right conclusion
                (equal conclusion (rw.trace-formula extras))
                ;; subproofs established any forced goals
                (equal (remove-duplicates
                        (rw.collect-forced-goals extras))
                       (logic.strip-conclusions subproofs)))))
```

The use of `defs` above requires some special attention. Since rewrite traces can make use of evaluation, a basic faithfulness requirement is that definitions used by the evaluator are axioms in the current history. As discussed in Section 10.2, one way to ensure this would be to store the definitions alongside the trace in the extras of the appeal, and check them in `rw.compile-trace-okp`. Since, in practice, the definitions we use throughout the proof are fixed, this would inefficiently lead us to check the definitions over and over again. To avoid this inefficiency, our approach is to check the definitions ahead of time, and then pass them in to the step checking function above.

In the Level 3 proof checker (and levels since then), a similar situation was encountered: we wanted to ensure, statically, that certain axioms and theorems were available, so that each use of a rule such as the transitivity of = would not require us to search the theorems. To accomplish this, the `level3.proofp` function looks for the necessary axioms and theorems before calling upon `level3.flag-proofp` to do the proof checking. This is easy to implement because only a fixed set of axioms and theorems are needed by each builder.

But the definitions used during evaluation may change from proof to proof as new definitions are added. This leads to a practical problem: where can the list of definitions to check be stored?

We approach this problem by introducing a special adapter appeal. Each Level 8 proof must begin with a special appeal that stores (1) the definitions that will be used throughout the proof, and (2) the adapter-free, "core" of the proof, as extras. Our `level8.proofp` function performs the static checks of the definitions before calling upon an auxiliary function to do the dynamic checks on the core of the proof.

Definition: `level8.proofp`
```
(pequal*
 (level8.proofp x axioms thms atbl)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'level8.adapter)
        (not subproofs)
        (tuplep 2 extras)
        (let ((defs (first extras))
              (core (second extras)))
          (and
           ;; Static checks as in Levels 3, 4, ...
           (memberp (axiom-equal-when-diff) axioms)
           (memberp (axiom-equal-when-same) axioms)
           ... and so on ...
           (equal (cdr (lookup 'not atbl)) 1)
           (equal (cdr (lookup 'equal atbl)) 2)
           (equal (cdr (lookup 'iff atbl)) 2)
           (equal (cdr (lookup 'if atbl)) 3)
           ;; Static checks of the definitions
           (definition-listp defs)
           (logic.formula-list-atblp defs atbl)
           (subsetp defs axioms)
           ;; Actual proof checking, with defs pre-checked
           (logic.appealp core)
           (equal conclusion (logic.conclusion core))
           (level8.proofp-aux core defs axioms thms atbl)))))
```

The auxiliary function, `level8.proofp-aux`, is like previous proof checkers, but takes the definitions as an extra argument. That is, `level8.proofp-aux` is a thin wrapper for the following flag function, in `proof` mode:

Definition: `level8.flag-proofp-aux`

```
(pequal* (level8.flag-proofp-aux flag x defs axioms thms atbl)
         (if (equal flag 'proof)
             (and (level8.step-okp x defs axioms thms atbl)
                  (level8.flag-proofp-aux 'list (logic.subproofs x)
                                          defs axioms thms atbl))
           (if (consp x)
               (and (level8.flag-proofp-aux 'proof (car x)
                                            defs axioms thms atbl)
```

```
                    (level8.flag-proofp-aux 'list (cdr x)
                                                    defs axioms thms atbl))
             t)))
```

Finally, `level8.step-okp` is just like the step-checking functions for previous levels, except that the definitions are also available as an additional argument.

Definition: `level8.step-okp`

```
(pequal* (level8.step-okp x defs axioms thms atbl)
         (let ((method (logic.method x)))
           (cond
            ((equal method 'rw.ccstep-list-bldr)
             (rw.ccstep-list-bldr-okp x defs thms atbl))
            ((equal method 'rw.compile-trace)
             (rw.compile-trace-okp x defs thms atbl))
            (t
             (level7.step-okp x axioms thms atbl)))))
```

Different adapters are used in subsequent levels. This approach is quite flexible, and effectively allows us to introduce and statically check any additional structures we would like to use before checking the core of the proof.

Altogether, we add 184 definitions and 1,059 theorems. The proofs total 6.8 GC and take 114.0 MB on disk. The largest proofs are the faithfulness theorems for the `if`-trace compilers and the evaluation builder. In total, four proofs exceed our 500 MC goal, with the largest taking 707.8 MC.

12.8 Level 9 – Unconditional Rewriting

In Level 9, we add a rule which allows us to rewrite a clause in a single step with our fast, unconditional rewriter. This involves translating the proofs that show URW produces a valid trace, and showing that FAST-URW produces the same result as URW.

Our step-checking function directly calls FAST-URW, so no rewrite trace ever needs to be constructed or checked. For faithfulness, we need to ensure that the control structure being used by FAST-URW is valid with respect to the current history. This can be an expensive check since the control structure includes all of the function definitions and rewrite rules that might be used by the rewriter. To avoid checking the control structures at every rewriting step, we introduced worlds in Section 10.2; the idea is to statically establish, in the adapter trace for Level 9, that the list of worlds to be used throughout the proof is valid. That is, all of the terms in the world should be well-formed with respect to the arity table, all of the definitions should be axioms, and the formulas for all of the rewrite rules should be theorems.

Since the worlds constructed by our user interface include all of the definitions and rewrite rules we have introduced, they can be quite large. We initially planned to carry out the static validity check by walking through each formula in the world and checking its arities via `lookup`, checking that the formula for every rule was a theorem using `memberp`, and checking that every definition was an axiom using `memberp`. This check was unacceptably slow, and resulted in delays of over five minutes before the dynamic checks began.

To address this, we implemented faster checks. For arity-checking, we first collect a list of obligations—pairs which associate function names with the number of arguments provided—for the terms and formulas throughout the worlds. We then use a simple mergesort to order the obligations and remove duplicates; we sort the arity table and run a linear ordered-subset check to ensure each obligation is met. To ensure the validity of each definition, we can similarly gather the definitions, sort them, sort the axioms, and use our ordered-subset check. And we take the same approach for rewrite rules. With these improvements, our static checks take about two seconds at the beginning of each proof.

Our step-checking function can assume the worlds it is given are valid, and we present its definition below. It looks up the world to use, and then calls upon FAST-URW (via rw.fast-world-urewrite-list) to ensure that the rewrite is justified. Because FAST-URW does not force any goals and simply rewrites each literal of the clause, it always produces exactly one resulting clause, and we expect a proof of the formula for this clause to be provided as a subproof.

Definition: rw.world-urewrite-list-bldr-okp

```
(pequal*
 (rw.world-urewrite-list-bldr-okp x worlds atbl)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'rw.world-urewrite-list-bldr)
        (tuplep 1 subproofs)
        (tuplep 4 extras)
        (let* ((orig-clause   (first extras))
               (result-clause (second extras))
               (theoryname    (third extras))
               (windex        (fourth extras))
               (world         (tactic.find-world windex worlds))
               (subconc       (logic.conclusion (first subproofs))))
          (and world
               (consp orig-clause)
               (consp result-clause)
               (logic.term-listp orig-clause)
               (logic.term-list-atblp orig-clause atbl)
               (equal (rw.fast-world-urewrite-list orig-clause
                                                   theoryname world)
                      result-clause)
               (equal (clause.clause-formula result-clause) subconc)
               (equal (clause.clause-formula orig-clause)
                      conclusion))))))
```

Level 9 is relatively large, and introduces 427 definitions and 2,475 theorems. Many of these proofs are quite trivial, and together the proofs are only 1.5 GC, or 903.4 MB when printed. The largest proof is under 200 MC.

12.9 Level 10 – Conditional Rewriting

In Level 10, we add conditional rewriting, via FAST-CRW, as a new kind of proof step. This requires us to translate the ACL2 proofs which establish that CRW produces a valid trace, and that FAST-CRW produces the same result as CRW. In this effort, we can make use of fairly powerful rules such as the one-step unconditional rewrites from Level 9, the one-step `clause.split` rule from Level 7, and can also use rewrite traces as proofs using the rules from Level 8.

Even so, these are difficult theorems, and and many of our translated proofs vastly exceed our goal of 500 MC. The worst offender is the main lemma for the fast rewriter (see Appendix C), which takes over 19 GC.

Why is this proof so large? With so many conjuncts and cases, the hypboxes used throughout each rewrite trace become quite large. These hypboxes are repeated again and again in each trace step, leading the number of conses measured by `rank` to reach this large number. Because of the large amount of structure sharing, the proof only requires 110 MB of disk space when printed. Also, we were also able to optimize our Level 8 step-checking functions for rewrite traces in order to avoid repeatedly checking these large hypboxes. As a result, we can check the proof in under 20 minutes on Lhug-3. (This 19 GC proof improves upon a previous, successful proof that took 241 GC. Even this larger proof was only 146 MB on disk, and with our optimizations we could check it successfully in just over an hour on Lhug-3.)

These optimizations are not difficult. Our usual recognizer for well-formed traces, rw.tracep, is a thin wrapper for the following flag function. This function performs poorly since it checks that every hypbox throughout the trace satisfies rw.hypboxp, and these checks are often redundant.

Definition: rw.flag-tracep

```
(pequal* (rw.flag-tracep flag x)
         (if (equal flag 'term)
             (let ((method    (car (car x)))
                   (rhs       (cdr (car x)))
                   (lhs       (car (car (cdr x))))
                   (iffp      (cdr (car (cdr x))))
                   (hypbox    (car (cdr (cdr x))))
                   ;; extras are (car (cdr (cdr (cdr x))))
                   (subtraces (cdr (cdr (cdr (cdr x))))))
               (and (symbolp method)
                    (rw.hypboxp hypbox)
                    (logic.termp lhs)
                    (logic.termp rhs)
                    (booleanp iffp)
                    (rw.flag-tracep 'list subtraces)))
           (if (consp x)
               (and (rw.flag-tracep 'term (car x))
                    (rw.flag-tracep 'list (cdr x)))
             t)))
```

Our optimized implementation takes an extra parameter, ext-hypbox, which we think of as an "external hypbox." For our function to be correct, the caller must separately check that ext-hypbox satisfies rw.hypboxp. If the hypbox of x happens to equal this external hypbox, we may conclude that it must be a valid rw.hypboxp, so we do not bother to check it. Once we have established that the hypbox of x is a valid (either because it is equal to the separately checked, external hypbox, or by running rw.hypboxp on it), we can begin using it as the ext-hypbox as we check its subtraces. Our optimized flag function is shown below.

Definition: `rw.fast-flag-tracep`

```
(pequal* (rw.fast-flag-tracep flag x ext-hypbox)
         (if (equal flag 'term)
             (let* ((method    (car (car x)))
                    (rhs       (cdr (car x)))
                    (lhs       (car (car (cdr x))))
                    (iffp      (cdr (car (cdr x))))
                    (hypbox    (car (cdr (cdr x))))
                    (subtraces (cdr (cdr (cdr (cdr x))))))
               (and (symbolp method)
                    (or (equal hypbox ext-hypbox)
                        (rw.hypboxp hypbox))
                    (logic.termp lhs)
                    (logic.termp rhs)
                    (booleanp iffp)
                    (rw.fast-flag-tracep 'list subtraces hypbox)))
           (if (consp x)
               (and (rw.fast-flag-tracep 'term (car x) ext-hypbox)
                    (rw.fast-flag-tracep 'list (cdr x) ext-hypbox))
             t)))
```

By an ordinary flag-function induction, we can prove the correctness of our optimized function. In our ACL2 proof sketch, the theorem is as follows.

ACL2 Code

```
(defthm rw.fast-tracep-removal
  (implies (rw.hypboxp ext-hypbox)
           (equal (rw.fast-tracep x ext-hypbox)
                  (rw.tracep x))))
```

Why is this fast? Recall from Section 9.1 that in most valid rewrite traces, the hypbox for the trace must agree with the hypboxes in the subtraces. In fact, the only exceptions are If General and If Same traces, where additional assumptions are made while rewriting the true and false branches. As a result, in practice the `(equal` `hypbox ext-hypbox)` check is almost always true, allowing us to avoid checking the hypbox.

We also optimize the arity-checking of traces in two ways. First, rather than repeatedly calling `lookup`, we collect the obligations to be checked and mergesort them to remove duplicates, then use an ordered-subset check. This reduces the computational complexity of the arity checking the trace from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, where n is the number of function symbols in the trace. Second, we implement the same `ext-hypbox` optimization as in `rw.fast-tracep`, so that we typically do not need to redundantly gather the obligations from these repeated hypboxes.

There are two other well-formedness checks for traces, but these do not inspect the hypboxes in any deep way, so we do not need to take any special measures to optimize them.

In all, Level 10 involves 82 definitions and 616 theorems. The proofs come to 60.6 GC. The proof of the main lemma for the fast rewriter is the largest proof, at 19.6 GC. Despite these large proof sizes, on disk the proofs come to only 2.1 GB, due to high amount of structure sharing and our use of the compacting printer.

12.10 Level 11 – Tactics

Level 11, our final proof checker, adds a single proof step that allows us to use a proof skeleton produced by our tactic system as a proof. In support of this, we translated the ACL2 proofs that justify all of the tactics described in Section 10.3. Since we can use FAST-CRW to rewrite whole clauses in a single step, proof sizes become a non-issue and it was easy to translate the ACL2 proofs.

The step-checking function for our highest-level proof steps is shown below. The proof skeleton is stored in the extras of the appeal, and is checked for validity using tactic.skeletonp, tactic.skeleton-okp, etc. Recall that the function tactic.skeleton-okp calls upon functions like tactic.split-first-okp and tactic.crewrite-all-okp, which in turn invoke functions like clause.split and FAST-CRW to ensure that the reduction is justified. Much like our arity checking for rewrite traces in Level 8, our fast arity-checking function for skeletons simply collects all of the function symbols, mergesorts them to remove duplicates, and carries out a linear, ordered subset check.

Definition: tactic.compile-skeleton-okp
```
(pequal*
 (tactic.compile-skeleton-okp x worlds axioms thms atbl)
 (let ((method     (logic.method x))
       (conclusion (logic.conclusion x))
       (subproofs  (logic.subproofs x))
       (extras     (logic.extras x)))
   (and (equal method 'tactic.compile-skeleton)
        (tactic.skeletonp extras)
        (tactic.skeleton-okp extras worlds)
        (tactic.fast-skeleton-atblp extras atbl)
        (tactic.skeleton-env-okp extras worlds axioms thms atbl)
        (memberp conclusion (clause.clause-list-formulas
                             (tactic.original-conclusions extras)))
        (equal (logic.strip-conclusions subproofs)
               (clause.clause-list-formulas
                (tactic.skeleton->goals extras))))))
```

A single tactic.compile-skeleton step can justify the use of any number of any of our tactics. Because of this, the typical Level 11 proof is comprised of two appeals: a level11.adapter that runs the static checks on the world (like the adapter traces for Levels 8-10), and a single tactic.compile-skeleton appeal that immediately justifies the proof using the skeleton produced by our tactic system.

All together, Level 11 involves 238 definitions and 1,169 theorems. The largest proof is only 20.3 MC. Together, the proofs take 2.7 GC, or 2.6 GB when printed on disk.

12.11 Comparing Proof Checkers

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 for a couple of example lemmas. We instruct our interface to construct the proof at each level. We can then compare the sizes of these proofs, as well as the amount of time needed to build and construct them. We carried out these comparisons on our ordinary development platform, and checked the proofs using our interface's embedded proof checker. Because of this, the times reported do not include any saving or reading of files. We took no efforts to ensure the machine was unused by others, so the times below may have some variance.

The "search" times reported below indicate how long it took our tactics to run to construct the proof skeleton, while the "build" time is the time it takes for the skeleton to be compiled into a Level n proof. Why does the search time change? In the lower levels, we use the slow versions of CRW and URW during the proof search, and save the resulting rewrite traces in the proof skeleton. Although this means the proof search takes longer, it reduces the overall time needed to find and build the proof: if we had instead used FAST-CRW and FAST-URW during the search, then building the proof would require us to redo each rewrite with CRW or URW to obtain the traces to compile.

As a first example, we consider the faithfulness of the disjoined transitivity of iff rule. In our ordinary bootstrapping process, this is a relatively large (441 MC) Level 2 proof. Below, all sizes are reported in megaconses, and all times are reported in seconds.

Level	1	2	3	4	5	6	7	8	9	10	11
Search	39.4	39.4	39.4	39.4	39.4	39.4	39.4	39.4	39.0	12.3	12.3
Build	406	226	117	106	102	101	101	0.8	0.5	.04	.008
Size	3,681	441	234	62	53	38	36	76	76	.8	.8
Check	11,440	2,968	914	433	408	342	332	50	50	12.8	12.6

As another example which gives a better picture of the higher levels, we consider the faithfulness of our evaluation rule. In our ordinary bootstrapping process, this is a medium size (222 MC) Level 7 proof, which makes comparably heavier use of case splitting and only light use of rewriting. Our attempt to construct a Level 1 version of this proof failed, exhausting the 32 GB of memory available on the machine (the proof had grown to over 25 GC before the failure was encountered). For comparison, our ACL2 proof of this lemma takes 82 seconds.

Lev.	1	2	3	4	5	6	7	8	9	10	11
Sch.	354	354	354	354	354	354	354	354	354	346	346
Bld.	∅	6,238	2,879	2,279	2,157	1,482	768	691	167	65	8
Size	∅	8,289	4,310	1,117	1,049	426	222	171	129	58	27
Chk.	∅	31,451	5,323	2,816	3,120	2,737	1,874	1,430	440	457	163

12.12 Final Checks

Once the bootstrapping is completed, we are left with many directories of proof files. We would now like to check these proofs using the program we developed in Chapter 4.

We summarize these directories below. The "utilities" directory contains definitions and theorems for simple arithmetic and list functions, and the "logic" directory includes our definitions and theorems about terms, formulas, and provability. The "level$n$" directories contain the definitions and proofs introduced at each level. The "user" directory is not part of any of our proof-checkers, and only contains a few trivial proofs about multiplication which use the Level 11 proof checker. (We imagine that an end-user of Milawa would begin working here.)

Directory	**Defs**	**Thms**	**Largest Proof**	**All Proofs**	**Disk Size**
			(megaconses)	(megaconses)	(megabytes)
Utilities	133	1,659	112.9	2,998.7	586.0
Logic	201	2,015	353.7	6,426.1	959.6
Level2	87	514	345.5	9,038.8	205.1
Level3	230	815	614.2	7,990.3	201.0
Level4	168	991	1,151.7	19,112.9	288.0
Level5	192	1,071	383.8	9,328.1	184.9
Level6	55	402	758.9	3,279.7	116.5
Level7	83	749	445.6	949.5	43.9
Level8	184	1,059	707.8	6,810.0	114.0
Level9	427	2,475	193.7	1,547.6	903.4
Level10	82	616	19,559.1	60,576.3	2,115.9
Level11	238	1,169	20.3	2,717.8	2,635.1
User	1	28	4.7	84.2	77.9
Totals	2,081	13,563		130,860.0	8,431.0

Note that all of the proofs carried out in the utilities, logic, and level2 directories are Level 1 proofs which will be checked by `logic.proofp`. The proofs in the level3 directory are Level 2 proofs which will be checked by `level2.proofp`, and so on. The user directory contains Level 11 proofs. Of course, since we can use the `SWITCH` command to change which proof checker is used by our core program, we can treat all of these directories as one long list of events to be checked.

Because of the large amount of computation required to check all of these proofs, we checkpoint our progress after processing each directory. That is, we process all of the events from our `utilities` directory, then use the `FINISH` command to save a new Milawa image where the utilities directory has been pre-loaded. We then use this image to process the `logic` directory, and so on.

When we use computers to check proofs, we always run the risk that the computing platform may make a mistake. This is partly unavoidable since computers are physical devices, but we are also relying upon a collection of software, including a Lisp environment and operating system, which are unverified and probably have bugs. These bugs, or an error in the underlying hardware, could perhaps lead our program to accept an invalid "proof" as legitimate.

To guard against this, we would like to check our proofs using a diverse collection of computers, operating systems, and Lisp environments. Accordingly, we have assembled and made available a compressed archive of our generated proofs which may be downloaded and checked independently. We also set out to check the proofs ourselves, using the following hardware and operating systems.

- *Lhug-3* is an HP Proliant DL585 server, and is part of the Mastodon cluster at the University of Texas at Austin. It has four 2.2 GHz AMD Opteron 850 processors, 32 GB of memory, and runs 64-bit Linux.
- *Nemesis* is a Dell Precision 390 workstation with a single 2.13 GHz Intel Core 2 Duo processor and 4 GB of memory, running 32-bit Linux.
- *Cele* is an Apple MacBook laptop with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of memory, running Darwin 9.7.0.
- *Jordan* is a home-assembled workstation with a 3.0 GHz Intel Core 2 Duo processor, and 4 GB of memory, running 64-bit Linux.

We had also checked previous versions of the proofs on *Shadowfax*, a Sun Fire 480W server with four 1.2 GHz UltraSparc III+ processors and 16 GB of memory, running SunOS 5.9, but we no longer have access to this or any other reasonably powerful Solaris machine.

A variety of Lisp implementations can be run on these computers. We considered using four Open Source implementations:

– Clozure Common Lisp (CCL) version 1.3 from June 2009,

– CLISP version 2.47 from October 2008,

– CMU Common Lisp (CMUCL) version 19f from April 2009, and

– Steel Bank Common Lisp (SBCL) version 1.0.29 from June 2009.

We have also tried using two commercial implementations,

– Allegro Common Lisp (ACL) version 8.0 from September 2006, and

– Scieneer Common Lisp (SCL) version 1.3.9 from November 2008.

The size of our proofs seems to stress many of these systems. In every case, measures were needed to increase various default limitations on resources such as the amount of heap space available and the depth of the call stack. We also made adjustments to improve the performance of garbage collection, e.g., by increasing the amount to allocate between collections.

Unfortunately, even with some effort, certain Lisps cannot process all of the proofs on these computers. For instance, despite reconfiguring the image to allow for a larger Lisp heap, Allegro runs out of memory on Nemesis while checking proofs in the level10 directory. Similarly, even when we increase the stack limit to the (low) maximum allowed by the Darwin operating system, CLISP runs out of stack space while processing the utilities directory on Cele.

Below, we summarize the user time (in minutes) taken to check each directory below for the systems we tried. The times we report are only suggestive: we took no efforts to ensure the machines were not in use by others, and on multi-core machines we typically ran more than one Lisp at a time. We write "err" to indicate Lisp bugs such as segmentation faults, hangs, or other failures, and "mem" to indicate that the Lisp ran out of stack or heap space. In no case was a proof ever rejected as invalid.

Platform	**Util**	**Logic**	**Lev2**	**Lev3**	**Lev4**	**Lev5**	**Lev6**
Lhug-3/CCL	19.0	47.5	77.4	218.5	249.0	276.0	101.7
Lhug-3/CLISP	168.8	511.4	1074.0	1703.4	3162.3	2868.4	1169.8
Lhug-3/CMUCL	21.5	63.9	111.7	204.5	324.6	306.8	116.6
Lhug-3/SBCL	32.9	88.4	163.7	283.4	519.7	436.4	193.1
Lhug-3/SCL	err	-	-	-	-	-	-
Nemesis/ACL	18.0	53.0	98.9	153.3	281.1	242.9	102.6
Nemesis/CCL	err	-	-	-	-	-	-
Nemesis/CLISP	78.2	230.8	456.5	728.9	err	-	-
Nemesis/CMUCL	16.9	50.8	87.8	159.4	253.5	227.6	88.6
Nemesis/SBCL	17.4	52.8	86.0	145.4	235.9	220.4	82.2
Cele/CCL	13.5	33.9	50.2	103.7	172.6	173.5	67.5
Cele/CLISP	mem	-	-	-	-	-	-
Cele/CMUCL	15.9	45.3	81.1	141.4	231.5	209.3	82.1
Cele/SBCL	18.2	46.3	84.0	144.4	242.1	220.5	83.9
Jordan/CCL	9.9	24.7	37.7	71.5	128.4	err	-
Jordan/CLISP	69.3	194.2	380.0	606.3	1156.6	1014.3	408.5
Jordan/CMUCL	13.2	36.7	65.6	114.0	184.7	167.7	67.7
Jordan/SBCL	17.8	52.7	98.4	166.6	295.3	264.9	104.3

Platform	**Lev7**	**Lev8**	**Lev9**	**Lev10**	**Lev11**	**User**	**Total**
Lhug-3/CCL	47.0	317.4	128.0	162.4	102.0	2.8	29.1 hrs
Lhug-3/CLISP	480.4	3497.2	1162.0	1706.3	1199.2	30.4	13.0 days
Lhug-3/CMUCL	51.2	357.3	131.5	228.0	185.3	4.8	35.2 hrs
Lhug-3/SBCL	71.5	517.8	224.9	285.5	210.9	5.4	50.6 hrs
Lhug-3/SCL	-	-	-	-	-	-	-
Nemesis/ACL	44.2	256.0	83.7	mem	-	-	-
Nemesis/CCL	-	-	-	-	-	-	-
Nemesis/CLISP	-	-	-	-	-	-	-
Nemesis/CMUCL	38.1	267.7	94.6	165.2	129.1	3.3	26.4 hrs
Nemesis/SBCL	36.3	248.1	91.2	158.3	119.7	3.0	24.9 hrs
Cele/CCL	30.7	207.8	80.9	120.5	82.7	2.4	19.0 hrs
Cele/CLISP	-	-	-	-	-	-	-
Cele/CMUCL	34.9	247.7	89.7	err	-	-	-
Cele/SBCL	36.0	256.0	91.7	249.1	111.3	2.7	26.4 hrs
Jordan/CCL	-	-	-	-	-	-	-
Jordan/CLISP	164.7	1214.9	390.8	658.2	478.8	12.3	4.7 days
Jordan/CMUCL	28.5	198.0	71.0	122.0	97.7	2.5	19.5 hrs
Jordan/SBCL	47.7	313.2	109.2	165.3	121.4	3.5	29.3 hrs

Chapter 13

Conclusion

In this dissertation, we have presented an approach for implementing a theorem prover that can be trusted. Rather than carry out proofs in a fully expansive manner, our approach is to show, ahead of time, that the theorem prover obeys the rules of its logic.

We begin by introducing a simple logic, and a social proof which argues that the rules of this logic are sound. Importantly, our logic allows us to introduce recursive functions, and we can implement these functions on a Common Lisp system. To have confidence that the Lisp system can run our functions correctly, we intentionally keep this connection quite simple.

We then implement a proof checker system for our logic, which (1) serves as a formalization for provability, and (2) can be run as a Common Lisp program to check actual proofs. So that one may have confidence that the proof checker only accepts theorems, we implement the program quite plainly and only allow it to accept proof steps corresponding to the rules of inference of our logic.

We develop a small program around the proof checker which allows us to introduce new definitions and prove theorems. Before accepting a formula as a theorem, this system requires the user to provide a proof of the formula which is accepted by the proof checker. These proofs are so large that the basic system would be impractical to use in formal verification. To counter this, we allow for the verification of new proof checkers. These higher-level proof checkers may accept proofs that make use of new kinds of proof steps that are not permitted by our basic system. By taking advantage of these new proof steps, high-level proofs can be written more concisely and checked more efficiently.

We write a theorem prover which is styled after ACL2, and which can carry out a backward proof search involving induction, case splitting, assumptions, calculation, rewriting with lemmas, destructor elimination, and other more manual techniques such as generalization and the use of equalities. Our high-level approach to verifying each of these techniques is to (1) introduce a fully expansive version of the technique, then (2) show that it can be used to produce a proof of any claim being made.

To follow this approach, we begin by verifying each algorithm with ACL2. Then, following the ACL2 proof as a sketch, we use our new theorem prover to discover proofs which justify the use of each of its own algorithms. In this way, our program is "self-verifying."

Through a bootstrapping process involving several intermediate proof checkers, we then mechanically translate these proofs into a format that can be checked by our small program. This process culminates in the verification of a very high-level proof checker which can employ all of our theorem prover's techniques in a single step.

Finally, using a number of different computers and Lisp implementations to minimize the chance of computer error, we check these proofs using our simple proof-checking system.

13.1 Relation to Other Work

There are several general-purpose theorem provers in widespread use. Our program is most closely related to ACL2 [52], but there are also several theorem proving systems based on higher-order logic, including HOL [34], HOL Light [40], Isabelle/HOL [67], and PVS [71]. Some other popular theorem provers, such as Coq [7] and Nuprl [88], employ constructive type theory.

Proof Representation

A major difference among these systems is the representation of proofs. In the ACL2 system, proofs are "whatever the `defthm` command accepts," and formal proofs are not generated. The 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.

Proof attempts in ACL2 produce human-readable logs which explain generally what the theorem prover is doing. But these logs contain English text and are not suitable for checking by other programs, so there is no readily available mechanism for gaining additional confidence in an ACL2 proof. Moreover, ACL2 itself has not been subjected to any mechanized formal analysis, and many reasoning errors have been discovered [51] in official releases, as summarized below.

ACL2 Release	**Reasoning Errors Fixed**
October, 1998 (Version 2.3)	Subversive recursions
August, 1999 (2.4)	Immediate force mode
June, 2000 (2.5)	Metafunctions with hypotheses
November, 2001 (2.6)	Linear arithmetic
	Evaluation in proofs
November, 2002 (2.7)	Functional instantiation
	BDDs
	Guards
March, 2004 (2.8)	Tautology checking
	ACL2 arrays
	Proof checker commands
	Defining packages
	Tracking axioms
	Type prescriptions in equivalences
	Redundancy and single-threaded objects

October, 2004 (2.9)	Package names
	Tracking program mode
	Macro expansion
	Linear arithmetic
August, 2005 (2.9.3)	Program mode in defconst
February, 2006 (2.9.4)	Meta rules with local events
June, 2006 (3.0)	Program mode in local
August, 2006 (3.0.1)	Local table events
December, 2006 (3.1)	Package witnesses
	Forcing in linear arithmetic
	Redundancy and measures
April, 2007 (3.2)	Unknown/hidden packages
	Meta rules
	Redefinition and program mode
	Raw lisp code in tracing
August, 2008 (3.4)	MBE in encapsulates
	State (value triples)
	Redundancy and built-in functions
May, 2009 (3.5)	Flet handling
	Inferred type-prescriptions in encapsulate
	ACL2 arrays
	Type-reasoning in lambdas
	Termination
August, 2009 (3.6)	Redundancy and ruler extenders
	Subversive recursions for constraints
September, 2009 (3.6.1)	Ruler extenders

In contrast, systems like HOL and HOL Light follow the fully expansive LCF [30] approach. Here, theorems are objects of type `thm` and represent proofs of sequents, $\Gamma \vdash c$, where Γ is a set of assumptions and c is a conclusion. The `thm` type is *abstract*, so the only way to construct a `thm` is to use a built-in constructor.

In a "pure" implementation of an LCF-style system (e.g., HOL Light), these constructors correspond to the rules of inference for the logic. For example, the reflexivity rule in higher-order logic is:

$$\frac{}{\emptyset \vdash t = t}$$

The corresponding constructor, `REFL`, takes a `term`-typed argument t as input and produces a `thm` with no assumptions and with the conclusion $t = t$. As another example, the rule of inference for discharging assumptions is:

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \rightarrow t_2}$$

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

If the type system is implemented correctly, the only way to create a `thm` object is to invoke constructors like `REFL` and `DISCH`. As a result, in a pure system, any `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 LCF-style systems [93, 4, 68], either as a way to export proofs into other theorem provers or to facilitate double-checking by external proof checkers.

It is also possible to "impurely" adopt the LCF approach. Here, an abstract `thm`-type is still used, but the proof constructors may be more powerful than the primitive rules of inference. An example is the PVS [70] system, which includes powerful primitives such as rewriting with lemmas. This approach is almost ACL2-like in that the correctness of the system's reasoning depends upon a relatively large amount of code, and "proofs" of non-theorems [38] sometimes result.

Coq and Nuprl have another, well-defined notion of proof. Certain types are called *propositions*. Whenever the type of an object x is a proposition, we say x itself is a *proof* of that proposition. No abstract type is used; instead 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., $P \to Q$.

Like the `thm` approach, the correctness of these systems depends on a relatively small kernel. The (relatively complex) 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.

Milawa's notion of proof differs from all of these approaches.

At lower levels, our proofs are somewhat LCF-like in that the proofs are carried out in a fully expansive manner. On the other hand, an important part of the LCF approach is that the type system ensures that every `thm` has been constructed with a built-in constructor, so that the intermediate steps of every `thm` may be garbage collected. Since no type system prevents us from constructing invalid appeals, low-level Milawa proofs must be stored "in full."

At higher levels, our proofs are more ACL2- or PVS-like, in that they make use of assumptions, evaluation, rewriting, and other algorithms which can emit no justification of their work. On the other hand, our algorithms have been verified, and unlike ACL2 we still have a well-defined notion of proof, viz. those appeals accepted by `level11.proofp`.

Finding Proofs

Another way in which theorem provers are distinguished is in the style of interaction used to find proofs. In each prover besides ACL2, proofs are constructed with tactics written in a general purpose programming language such as ML. The flexible nature of these programming language allows for tactics to be easily composed, and the user can make use of *strategies* or *tacticals* which can try to apply tactics in a variety of ways, e.g., “try these tactics and use the first one that succeeds.”

If the validation produced by the tactic attempts to construct an invalid proof, an error will be caused by the `thm` constructor and the proof attempt will fail. Hence, a tactic may have bugs like any ordinary program, but the proofs it constructs can be trusted even without needing to verify the tactic, itself.

In contrast, the ACL2 system does not have an explicit notion of proof, and its closest approximation of tactics and tacticals, called proof checker macros, are rarely used. Instead, following The Method, most work is accomplished by proving lemmas that add rewrite rules to influence the ACL2 rewriter. When the rewriter is unable to find the proof using the lemmas available, the user may provide extra hints, either manually or automatically through a hint-computation mechanism.

Nothing prevents a tactic-based system from following the heuristic rewriting approach. For instance, Boulton [9] 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 HOL4, and the `simp` tactic in Isabelle/HOL. Our tactics are in this same spirit, and were intended to allow us to carry out ACL2-like steps.

Because our system is first-order, the tactic systems of other theorem provers are considerably more flexible than ours. We cannot dynamically construct or compose validations, and to add a new tactic we must modify our tactic compiler. The “list of goals” approach in our skeleton also limits the flexibility of tacticals, e.g., considerable infrastructure must be developed to implement our waterfall, whereas in a higher-order tactic system one would implement this as a relatively simple tactical.

Formal Analysis of Proof Checkers

A key construction in Gödel's [28] proof was the introduction, in his logic, of a proof checking program for his logic. The logics used by theorem provers are expressive enough to introduce proof checkers, as we have done with `logic.proofp`, and such embeddings have been carried out in many projects.

Shankar [82] carried out a proof of the incompleteness theorem in NQTHM by first defining, as an NQTHM function, a proof checker for Shoenfield's first-order logic with Cohen's Z2 axioms. In this effort, NQTHM functions were also implemented to implement rules such as tautology checking and equivalence checking, and shown with NQTHM to be correct with respect to the proof checker. We have reimplemented many of these functions in our system (see Chapter 5), and this use of NQTHM as a metalogic is quite similar to our use of ACL2 to sketch out our proofs.

In a similar effort, incompleteness was studied by O'Connor [69] in the Coq theorem prover. In this work, proofs are represented using dependent types, so that any object of type `Prf` represents a valid proof.

Such embeddings have also been used to study properties of proof-checking programs. For instance, von Wright [90, 91] 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 [75] wrote a first-order theorem prover as definitions in Isabelle/HOL and, using Isabelle/HOL, proved the program to be sound and complete.

Harrison [41] 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 implementation is consistent. Without the axiom, he can show the encoded implementation, minus the axiom of infinity, is consistent. These results indicate "something close to the actual *implementation* of HOL" is sound.

Barras [2] has used Coq to prove the strong normalization and decidability of type inference for the Calculus of Constructions. This work may be an important first step toward the verification of the kernel of Coq (which implements the Calculus of Inductive Constructions, rather than the Calculus of Constructions).

Independently Checking Proofs

There have also been some projects where one system is used to check the work of another. This approach may also be useful in separating proof search from proof construction, or may simply be used to import or double-check proofs from one system with another.

McCune and Shumsky [64] have written ACL2 functions to check proof objects emitted by the resolution prover Otter (and its successor, Prover9) for validity. The proof search is carried out by Otter, and the ACL2 program only checks that Otter did not make a mistake. No attempt is made to verify Otter itself (which is an optimized C program). Instead, an ACL2 function is introduced to check the Otter proof transcript, and ACL2 is used to prove that the checking function is sound with respect to a simple notion of interpreting formulas.

Obua and Skalberg [68] have extended HOL Light with a proof recorder that tracks calls to the proof constructors. A 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.

Caldwell and Cowles [22] 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 involves a 60,000-line Lisp core and a 40,000-line ML interface, with 167 rules of inference that are sometimes complicated, e.g., the `arith` rule. The project is seems to be in the early stages.

The type inferencing algorithm verified by Barras [2] for the Calculus of Constructions has been combined with a parser and pretty-printer to obtain a stand-alone proof checking program for the Calculus of Constructions, and this program can be used to independently check (some) proofs from Coq.

Meta Reasoning

Our system uses `logic.proofp` in two ways. As our lowest-level proof checker, it is executed by our Common Lisp program to check proofs during the initial stages of bootstrapping. But it also serves as a formalization of provability in our logic, which allows us to reason about the fidelity of our higher-level proof checkers.

Even without such a proof checker, many other theorem provers have some support for meta reasoning (reasoning about provability). Most of this work follows, with minor differences, the *metafunction* approach [13], which involves five steps:

1. An encoding for the relevant terms is introduced,

2. A semantic function, $\mathit{meaning}(\mathit{term}, \mathit{env})$, is introduced to evaluate an encoded term w.r.t. an environment that provides values for variables,

3. A "metafunction", $\mathit{fn}(\mathit{term})$, is introduced to simplify encoded terms,

4. The user proves $\mathit{meaning}(\mathit{fn}(\mathit{term}), \mathit{env}) = \mathit{meaning}(\mathit{term}, \mathit{env})$, 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* functions for a fixed set of concepts can be introduced using the `defevaluator` facility. A metafunction, *fn*, is a regular ACL2 program, written as a recursive function that manipulates encoded terms. A built-in mechanism allows the system to begin using a metafunction after the *meaning* theorem has been proven.

Metafunctions can be a useful tool [85] 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. In recent versions of ACL2, these issues are largely solved by a new feature called clause processors [55], which are essentially metafunctions that operate on clauses instead of terms.

Unfortunately, since the ACL2 simplifier is not a function in the ACL2 logic, metafunctions can only call upon it heuristically [46]. That is, even if ACL2 can rewrite *term* to term', we cannot assume $\mathit{term} = \mathit{term}'$ when we try to prove that the meaning of terms is preserved by our metafunction.

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 metafunctions. Many features, such as linear arithmetic and type reasoning, are deeply integrated into the rewriter [11], involve keeping track of state, and generally do not fit well into the metafunction or clause processing paradigms. 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 this metafunction preserves the meaning of terms 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." [39]

Slind [84] proposed a scheme for allowing `mk_thm`, an "arbitrary" `thm` constructor that does not correspond to any rule of inference, to be used under restricted circumstances. First, the semantics of ML would be formalized in HOL, as would the HOL implementation. Then, `mk_thm t` is to be permitted only if we can prove there is some function `f` that produces a usual, fully expansive HOL proof of `t`. But this idea has not been implemented.

More recently Chaieb and Nipkow [23] 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 *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 [5]. 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 reportedly 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 `thm` objects remains separated from the logic, and as a result we still cannot reason about `thm` construction and the provability of formulas.

Metafunctions are also supported in Coq. Grégoire and Mahboubi [35] have introduced a procedure for reasoning about equality between polynomials in commutative rings. They define a new type to represent encoded polynomials over a ring and provide a *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 formulas in general.

Knoblock and Constable [56] 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. But these ideas have not been implemented.

13.2 Future Directions

No matter what logic and basic architecture is used, a large undertaking is required to develop a theorem prover and the tactic or lemma libraries to make it useful. Since mechanically verifying the theorem prover's algorithms certainly adds to this work, some important considerations are how much effort is required and whether it is worthwhile.

This project began four years ago, in the fall of 2005. The main programming and proving effort taking the author about two and a half years. The ACL2 proof scripts, which contain all of the function definitions and lemmas for our theorem prover, and which also implement the user interface, come to around 110,000 lines with about 30% comments or blank lines. The bootstrapping code, which drives the interface to carry out the self-verification process, takes another 60,000 lines of code, about 40% of which is comments or blank lines. In comparison with other popular theorem provers, Milawa is admittedly modest in its capabilities, yet this amount of effort does not seem unreasonable.

If we were to develop a successor to Milawa, there are a couple of things we would do differently from the beginning. The syntax of formulas in our logic and many of our rules of inference have been directly adopted from Shoenfield's [83] 1967 text. This decision was made early in the project, and was motivated by the desire to implement something very close to the ACL2 logic. (Descriptions of the ACL2 logic, such as *Computer Aided Reasoning: An Approach* [50], *A Precise Description of the ACL2 Logic* [53], and *Structured Theory Development for a Mechanized Logic* [54] typically adopt Shoenfield's presentation, with the qualification that any classical first-order logic with equality would be acceptable.) In retrospect, implementing a Hilbert-style system seems like a mistake. Working with formulas rather than sequents or clauses is needlessly difficult because one must always be concerned with the shape of the formulas. For instance, this leads to disjoined and non-disjoined versions of many rules.

Our proof representation could also likely be improved by allowing each proof to have a list of conclusions, rather than a single conclusion. In particular, it is not possible to develop a high-level step-checker for our clause cleaning routine, since it applies to a list of clauses rather than to a single clause. We can work around this in some ways, e.g., by integrating cleaning into our splitting algorithm. But it would be better to be able to support clause-list to clause-list reductions directly.

But to really develop an "industrial strength" version of Milawa, we think the two main areas to improve upon are the efficiency and expressivity of our programming language.

Theorem provers are often used as tools for modeling other systems and proving properties about those models. The ability to animate these models efficiently is often considered to be unimportant, and specifications are often written using constructs that are difficult to execute, e.g., quantifiers. Some notable exceptions include models of processors [92], floating point circuits [78, 49], etc., where co-simulation is used to gain confidence in the validity of the model. Here, execution efficiency directly impacts the amount of co-simulation which is feasible.

Efficient animation may have greater importance in systems which, like ours, involve developing theorem proving algorithms in the logic. In our system, our primary theorem proving algorithms are written in our logic. In other systems with reflective capabilities, the efficiency of metafunctions, etc., would also seem to be important. Because of this, many theorem proving systems now also include features to facilitate efficient execution, for instance the Coq system allows its specifications to be run as OCaml programs [7], and Isabelle/HOL specifications can now be translated into ML programs [5].

In ACL2, functions in the logic can be executed as Common Lisp programs in a fairly direct way. This connection is quite sophisticated, e.g., specifications can be annotated with type declarations and the ACL2 system can verify that these annotations are justified; the compiler can then use these annotations to use native machine arithmetic, etc. Historically, execution efficiency has been regarded as one of ACL2's strengths in comparison to other theorem provers; for instance, Gordon, et. al, [32] have developed a mechanism to use ACL2 as an execution engine for HOL models, and report that using ACL2 execution is 300 times faster than using HOL's EVAL facility when animating a specification of the floating-point unit of an ARM co-processor.

Our system is like ACL2's in that we can run functions in our logic using a Common Lisp system. Although we can write programs somewhat efficiently by inlining function calls and using tail recursion, our connection to Common Lisp is much less sophisticated than ACL2's. To name a few inefficiencies with our approach:

– We do not have a mechanism like ACL2's guards, so we cannot safely annotate our functions with type declarations. As a result, all arithmetic in Milawa must be performed on arbitrary-precision integers, and primitives like `+` and `car` always involve runtime type checking.

– We lack any kind of function object or function pointer, so case statements must be used to emulate polymorphic calls.

– We do not have any mutable structures such as arrays and hash tables. Instead, we must rely upon trees of conses to implement records, lists, search trees, and so on. This leads many computations to perform consing, causing more work for the garbage collector.

– We have not implemented any parallelism capabilities, which is a particularly significant limitation given current trends toward multi-core processors.

Another minor note is that because our arithmetic needs have been so light, we have not implemented primitive functions for multiplication, division, remainder, and other bitwise operations. This would be very easy to remedy by adding new primitives—we only need to extend the initial arity table and the base evaluator.

At any rate, a particularly interesting line of future research would be to identify a small set of primitives whose behavior could be cleanly described through axioms, yet which would allow for the implementation of high-performance, parallel algorithms.

A good example of this is the fast association list system implemented by Boyer and Hunt [16] as an experimental extension to ACL2. Here, the special function `(hons-acons key val alist)` is logically equal to `(cons (cons key val)` alist), and (hons-get key alist) is akin to our lookup function. When hons-acons is used to construct an association list, a corresponding, "hidden" hash table is extended by binding key to val. When hons-get is used, the value can be read from the hash table rather than the alist. For this optimization to take place, the alist must be used in a single-threaded manner. That is, if we extend the alist x to x' by calling (hons-acons k v x), then subsequent calls of hons-acons and hons-get should only be applied to x', and not to x. If this discipline is not followed, the hash table is not used, warning messages about slow performance are generated, and ordinary association list operations are carried out.

Another example is Rager's [73] experimental introduction of parallelism primitives into ACL2. A plet construct is like let, but allows for parallel execution of the computations being bound. That is, in the following example, the $v_1, \ldots, v_n$ might be run in parallel:

```
(plet ((x_1 v_1)
       ...
       (x_n v_n))
  ...)
```

Another new primitive, pand, can compute (if (and v_1 ... v_n) t nil) by perhaps executing each v_i in parallel. This evaluation can also short-circuit, e.g., once any v_i evaluates to nil, any threads computing the other values can be aborted.

In Milawa there is very little "system-level" code that makes direct use of Common Lisp primitives. Instead, almost all of our functions are defined atop our logical primitives. Because of this, it would be relatively easy to change our system by using new definitions for primitives like car and +, and this may open many possibilities for extensions of the varieties just mentioned.

For instance, in fast association lists, the actual alist must also be constructed, so that if it is passed as an argument to functions like car, cdr, and equal, the correct result can be determined. In Milawa, it might be possible to avoid this overhead, by having primitives like `car` and `cdr` implement special cases for hash tables, e.g., they could cause an error, or could perhaps print a performance warning before constructing an alist from the hash table.

Similarly, Rager's parallelism primitives do not permit producer/consumer style parallelism. It seems difficult to develop a logical story to explain the behavior of a shared queue if `car` or `cdr` might be called on the queue while values are still being produced. Queue-aware versions of the primitives, which could block until the queue has additional data, might neatly solve this problem.

We now consider the expressivity of our programming language. With the exception of ACL2, most general-purpose theorem provers use logics which are considerably more expressive than ours. Notably, our logic lacks static typing, quantifiers, and higher-order functions.

In *Computer Aided Reasoning: An Approach*, Kaufmann, Manolios, and Moore assert that these limitations "can be overcome without undue violence to the intuitions you are trying to capture," [50] and indeed the ACL2 and NQTHM systems have been successfully used in many wide-ranging hardware and software verification projects, as mentioned at the beginning of Chapter 2. It may also be that the relative simplicity of our formulas has played a role in our success in reasoning about proofs.

And yet, doing without higher-order functions has not been particularly easy. Our tactic system is quite convoluted and is much less flexible than that of a true LCF-style system. This is largely due to our inability to dynamically produce validation functions. A great number of our theorems and functions—for introducing new "types," for recognizing the validity of steps in traces and proofs, and so on—have a boilerplate feel. It seems like much of this could be made easier using higher-order functions and a typed logic.

Above, we noted that although efficient animation is often not important when writing models of other systems, it may be much more important if our theorem proving algorithms are written in the logic. Perhaps, similarly, while the limitations of first-order systems are not too damaging when modeling hardware, virtual machines, and so on, the greater flexibility of higher-order systems is desirable when writing theorem provers.

Appendices

Appendix A

Derivations for Iff

In this appendix, we present the details behind the formal theorems and derived rules from Section 7.3 about the function `iff`.

Derived Rule A-1. If of t

$$\frac{}{\texttt{(if t } b\ c\texttt{)} = b}$$

Derivation. (8)

$\texttt{t} \neq \texttt{nil}$	Axiom t not nil
$\texttt{(if t } b\ c\texttt{)} = b$	If when not nil ☐

Derived Rule A-2. If of nil

$$\frac{}{\texttt{(if nil } b\ c\texttt{)} = c}$$

Derivation. (9)

$\texttt{nil} = \texttt{nil}$	Reflexivity
$\texttt{(if nil } b\ c\texttt{)} = c$	If when nil ☐

Formal Theorem A-1. Iff lhs false

$$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$$

Proof.

$\texttt{(iff x y)}$	Definition of iff	
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$		
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x y)}$	Expansion	(*1)
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$		
$\texttt{x} \neq \texttt{nil} \lor \texttt{x} = \texttt{nil}$	Prop. schema	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x (if y t nil) (if y nil t))}$	Dj. if when nil	
$\quad = \texttt{(if y nil t)}$		
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y nil t)}$	Dj. trans. = *1	□

Formal Theorem A-2. Iff lhs true

$$\texttt{x} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y t nil)}$$

Proof.

$\texttt{(iff x y)}$	Definition of iff	
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$		
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x y)}$	Expansion	(*1)
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$		
$\texttt{x} \neq \texttt{nil} \lor \texttt{x} = \texttt{nil}$	Prop. schema	
$\texttt{x} = \texttt{nil} \lor \texttt{x} \neq \texttt{nil}$	Commute or	
$\texttt{x} = \texttt{nil} \lor \texttt{(if x (if y t nil) (if y nil t))}$	Dj. if when nnil	
$\quad = \texttt{(if y t nil)}$		
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y t nil)}$	Dj. trans. = *1	□

Formal Theorem A-3. Iff rhs false

$$\texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if x nil t)}$$

Proof.

$\texttt{(iff x y)}$	Definition of iff	
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$		
$\texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)}$	Expansion	(*1)
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$		
$\texttt{x} = \texttt{x}$	Reflexivity	
$\texttt{y} \neq \texttt{nil} \lor \texttt{x} = \texttt{x}$	Expansion	(*2a)
$\texttt{y} \neq \texttt{nil} \lor \texttt{y} = \texttt{nil}$	Prop. schema	
$\texttt{y} \neq \texttt{nil} \lor \texttt{(if y t nil)} = \texttt{nil}$	Dj. if when nil	(*2b)

$\texttt{y} \neq \texttt{nil} \lor \texttt{(if y nil t)} = \texttt{t}$	Dj. if when nil	(*2c)	
$\texttt{y} \neq \texttt{nil} \lor \texttt{(if x (if y t nil) (if y nil t))}$	Dj. = args *2abc		
$\quad = \texttt{(if x nil t)}$			
$\texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if x nil t)}$	Dj. trans. = *1		□

Formal Theorem A-4. Iff rhs true

$$\texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if x t nil)}$$

Proof.

$\texttt{(iff x y)}$	Definition of iff		
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$			
$\texttt{y} = \texttt{nil} \lor \texttt{(iff x y)}$	Expansion	(*1)	
$\quad = \texttt{(if x (if y t nil) (if y nil t))}$			
$\texttt{x} = \texttt{x}$	Reflexivity		
$\texttt{y} = \texttt{nil} \lor \texttt{x} = \texttt{x}$	Expansion	(*2a)	
$\texttt{y} \neq \texttt{nil} \lor \texttt{y} = \texttt{nil}$	Prop. schema		
$\texttt{y} = \texttt{nil} \lor \texttt{y} \neq \texttt{nil}$	Commute or		
$\texttt{y} = \texttt{nil} \lor \texttt{(if y t nil)} = \texttt{t}$	Dj. if when nnil	(*2b)	
$\texttt{y} = \texttt{nil} \lor \texttt{(if y nil t)} = \texttt{nil}$	Dj. if when nnil	(*2c)	
$\texttt{y} = \texttt{nil} \lor \texttt{(if x (if y t nil) (if y nil t))}$	Dj. = args *2abc		
$\quad = \texttt{(if x t nil)}$			
$\texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if x t nil)}$	Dj. trans. = *1		□

Formal Theorem A-5. Iff both true

$$\texttt{x} = \texttt{nil} \lor \texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y t nil)}$	Th. iff lhs true		
$(\texttt{x} = \texttt{nil} \lor \texttt{y} = \texttt{nil}) \lor \texttt{(iff x y)} = \texttt{(if y t nil)}$	Multi assoc exp.	(*1)	
$\texttt{x} = \texttt{nil} \lor \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil		
$\texttt{y} = \texttt{nil} \lor \texttt{(if y t nil)} = \texttt{t}$	Instantiation		
$(\texttt{x} = \texttt{nil} \lor \texttt{y} = \texttt{nil}) \lor \texttt{(if y t nil)} = \texttt{t}$	Multi assoc exp.		
$(\texttt{x} = \texttt{nil} \lor \texttt{y} = \texttt{nil}) \lor \texttt{(iff x y)} = \texttt{t}$	Dj. trans. = *1		
$\texttt{x} = \texttt{nil} \lor \texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Right assoc.		□

Formal Theorem A-6. Iff both false

$$\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$$

Proof.

$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$	Th. iff lhs false	
$(\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$	Multi assoc exp.	(*1)
$\texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$\texttt{y} \neq \texttt{nil} \vee \texttt{(if y nil t)} = \texttt{t}$	Instantiation	
$(\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(if y nil t)} = \texttt{t}$	Multi assoc exp.	
$(\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(iff x y)} = \texttt{t}$	Dj. trans. = *1	
$\texttt{x} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$	Right assoc.	□

Formal Theorem A-7. Iff true false

$$\texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{nil}$$

Proof.

$\texttt{x} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y t nil)}$	Th. iff lhs true	
$(\texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(iff x y)} = \texttt{(if y t nil)}$	Multi assoc exp.	(*1)
$\texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$\texttt{y} \neq \texttt{nil} \vee \texttt{(if y t nil)} = \texttt{nil}$	Instantiation	
$(\texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(if y t nil)} = \texttt{nil}$	Multi assoc exp.	
$(\texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(iff x y)} = \texttt{nil}$	Dj. trans. = *1	
$\texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{nil}$	Right assoc.	□

Formal Theorem A-8. Iff false true

$$\texttt{x} \neq \texttt{nil} \vee \texttt{y} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{nil}$$

Proof.

$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$	Th. iff lhs false	
$(\texttt{x} \neq \texttt{nil} \vee \texttt{y} = \texttt{nil}) \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$	Multi assoc exp.	(*1)
$\texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$\texttt{y} = \texttt{nil} \vee \texttt{(if y nil t)} = \texttt{nil}$	Instantiation	

$(\texttt{x} \neq \texttt{nil} \lor \texttt{y} = \texttt{nil}) \lor \texttt{(if y nil t)} = \texttt{nil}$	Multi assoc exp.	
$(\texttt{x} \neq \texttt{nil} \lor \texttt{y} = \texttt{nil}) \lor \texttt{(iff x y)} = \texttt{nil}$	Dj. trans. = *1	
$\texttt{x} \neq \texttt{nil} \lor \texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil}$	Right assoc.	□

Formal Theorem A-9. Iff t when not nil

$$\texttt{x} = \texttt{nil} \lor \texttt{(iff x t)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y t nil)}$	Th. iff lhs true	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x t)} = \texttt{(if t t nil)}$	Instantiation	(*1)
$\texttt{(if t t nil)} = \texttt{t}$	If of t	
$\texttt{x} = \texttt{nil} \lor \texttt{(if t t nil)} = \texttt{t}$	Expansion	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x t)} = \texttt{t}$	Dj. trans. = *1	□

Derived Rule A-3. Iff t from ≠ nil

$$\frac{a \neq \texttt{nil}}{\texttt{(iff } a \texttt{ t)} = \texttt{t}}$$

Derivation. (7)

$\texttt{x} = \texttt{nil} \lor \texttt{(iff x t)} = \texttt{t}$	Th. iff t, nnil	
$a = \texttt{nil} \lor \texttt{(iff } a \texttt{ t)} = \texttt{t}$	Instantiation	
$a \neq \texttt{nil}$	Given	
$\texttt{(iff } a \texttt{ t)} = \texttt{t}$	Modus ponens 2	□

Derived Rule A-4. Disjoined iff t from ≠ nil

$$\frac{P \lor a \neq \texttt{nil}}{P \lor \texttt{(iff } a \texttt{ t)} = \texttt{t}}$$

Derivation. (17)

$\texttt{x} = \texttt{nil} \lor \texttt{(iff x t)} = \texttt{t}$	Th. iff t, nnil
$a = \texttt{nil} \lor \texttt{(iff } a \texttt{ t)} = \texttt{t}$	Instantiation
$P \lor a = \texttt{nil} \lor \texttt{(iff } a \texttt{ t)} = \texttt{t}$	Expansion

$P \vee a \neq$ `nil`	Given	
$P \vee$ `(iff` a `t)` $=$ `t`	Dj. mp2	$\square$

Formal Theorem A-10. Iff t when nil

$$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x t)} = \texttt{nil}$$

Proof.

`x` $\neq$ `nil` $\vee$ `(iff x y)` $=$ `(if y nil t)`	Th. iff lhs false		
`x` $\neq$ `nil` $\vee$ `(iff x t)` $=$ `(if t nil t)`	Instantiation	(*1)	
`(if t nil nil)` $=$ `nil`	If of t		
`x` $\neq$ `nil` $\vee$ `(if t nil nil)` $=$ `nil`	Expansion		
`x` $\neq$ `nil` $\vee$ `(iff x t)` $=$ `nil`	Dj. trans. $=$ *1		$\square$

Derived Rule A-5. $\neq$ nil from iff t

$$\frac{\texttt{(iff } a \texttt{ t)} \neq \texttt{nil}}{a \neq \texttt{nil}}$$

Derivation. (9)

`x` $\neq$ `nil` $\vee$ `(iff x t)` $=$ `nil`	Th. iff t when nil	
$a \neq$ `nil` $\vee$ `(iff` a `t)` $=$ `nil`	Instantiation	
`(iff` a `t)` $=$ `nil` $\vee$ $a \neq$ `nil`	Commute or	
`(iff` a `t)` $\neq$ `nil`	Given	
$a \neq$ `nil`	Modus ponens 2	$\square$

Derived Rule A-6. Disjoined $\neq$ nil from iff t

$$\frac{P \vee \texttt{(iff } a \texttt{ t)} \neq \texttt{nil}}{P \vee a \neq \texttt{nil}}$$

Derivation. (19)

`x` $\neq$ `nil` $\vee$ `(iff x t)` $=$ `nil`	Th. iff t when nil
$a \neq$ `nil` $\vee$ `(iff` a `t)` $=$ `nil`	Instantiation
`(iff` a `t)` $=$ `nil` $\vee$ $a \neq$ `nil`	Commute or

$P \vee$ `(iff a t)` $=$ `nil` $\vee\ a \neq$ `nil`	Expansion	
$P \vee$ `(iff a t)` $\neq$ `nil`	Given	
$P \vee a \neq$ `nil`	Dj. mp2	□

Formal Theorem A-11. Iff nil when nil

`x` $\neq$ `nil` $\vee$ `(iff x nil)` $=$ `t`

Proof.

`x` $\neq$ `nil` $\vee$ `(iff x y)` $=$ `(if y nil t)`	Th. iff lhs false		
`x` $\neq$ `nil` $\vee$ `(iff x nil)` $=$ `(if nil nil t)`	Instantiation	(*1)	
`(if nil nil t)` $=$ `nil`	If of nil		
`x` $\neq$ `nil` $\vee$ `(if nil nil t)` $=$ `nil`	Expansion		
`x` $\neq$ `nil` $\vee$ `(iff x nil)` $=$ `nil`	Dj. trans. $=$ *1		□

Formal Theorem A-12. Iff nil when not nil

`x` $=$ `nil` $\vee$ `(iff x nil)` $=$ `nil`

Proof.

`x` $=$ `nil` $\vee$ `(iff x y)` $=$ `(if y t nil)`	Th. iff lhs true		
`x` $=$ `nil` $\vee$ `(iff x nil)` $=$ `(if nil t nil)`	Instantiation	(*1)	
`(if nil t nil)` $=$ `nil`	If of nil		
`x` $=$ `nil` $\vee$ `(if nil t nil)` $=$ `nil`	Expansion		
`x` $=$ `nil` $\vee$ `(iff x nil)` $=$ `nil`	Dj. trans. $=$ *1		□

Formal Theorem A-13. Iff nil or t

`(iff x y)` $=$ `nil` $\vee$ `(iff x y)` $=$ `t`

Proof.

`x` $=$ `nil` $\vee$ `y` $=$ `nil` $\vee$ `(iff x y)` $=$ `t`	Th. iff both true
`x` $\neq$ `nil` $\vee$ `y` $=$ `nil` $\vee$ `(iff x y)` $=$ `nil`	Th. iff false true

$(\texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t})$ $\lor \texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil}$	Cut	
$\texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$ $\lor \texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil}$	Right assoc.	
$\texttt{y} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Generic subset	(*1)
$\texttt{x} = \texttt{nil} \lor \texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil}$	Th. iff true false	
$\texttt{x} \neq \texttt{nil} \lor \texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Th. iff both false	
$(\texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil})$ $\lor \texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Cut	
$\texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil}$ $\lor \texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Right assoc.	
$\texttt{y} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Generic subset	
$(\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t})$ $\lor \texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Cut *1	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{t}$	Contraction	□

Formal Theorem A-14. Reflexivity of iff

$$\texttt{(iff x x)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y t nil)}$	Th. iff lhs true	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x x)} = \texttt{(if x t nil)}$	Instantiation	(*1a)
$\texttt{x} = \texttt{nil} \lor \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$\texttt{x} = \texttt{nil} \lor \texttt{(if x t nil)} = \texttt{t}$	Instantiation	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x x)} = \texttt{t}$	Dj. trans. = *1a	(*1)
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{(if y nil t)}$	Th. iff lhs false	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x x)} = \texttt{(if x nil t)}$	Instantiation	(*2a)
$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x nil t)} = \texttt{t}$	Instantiation	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x x)} = \texttt{t}$	Dj. trans. = *2a	(*2)
$\texttt{(iff x x)} = \texttt{t} \lor \texttt{(iff x x)} = \texttt{t}$	Cut *1, *2	
$\texttt{(iff x x)} = \texttt{t}$	Contraction	□

Formal Theorem A-15. Symmetry of iff

$$\texttt{(iff x y)} = \texttt{(iff y x)}$$

Proof.

$\texttt{y} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if x t nil)}$	Th. iff rhs true	
$\texttt{x} = \texttt{nil} \vee \texttt{(iff y x)} = \texttt{(if y t nil)}$	Instantiation	
$\texttt{x} = \texttt{nil} \vee \texttt{(if y t nil)} = \texttt{(iff y x)}$	Dj. commute =	
$\texttt{x} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y t nil)}$	Th. iff lhs true	
$\texttt{x} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(iff y x)}$	Dj. trans. =	(*1)
$\texttt{y} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if x nil t)}$	Th. iff rhs false	
$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff y x)} = \texttt{(if y nil t)}$	Instantiation	
$\texttt{x} \neq \texttt{nil} \vee \texttt{(if y nil t)} = \texttt{(iff y x)}$	Dj. commute =	
$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(if y nil t)}$	Th. iff lhs false	
$\texttt{x} \neq \texttt{nil} \vee \texttt{(iff x y)} = \texttt{(iff y x)}$	Dj. trans. =	(*2)
$\texttt{(iff x y)} = \texttt{(iff y x)}$ $\quad \vee \texttt{(iff x y)} = \texttt{(iff y x)}$	Cut *1, *2	
$\texttt{(iff x y)} = \texttt{(iff y x)}$	Contraction	□

Derived Rule A-7. Iff t from not nil

$$\frac{\texttt{(iff } a\ b\texttt{)} \neq \texttt{nil}}{\texttt{(iff } a\ b\texttt{)} = \texttt{t}}$$

Derivation. (7)

$\texttt{(iff x y)} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$	Th. iff nil or t	
$\texttt{(iff } a\ b\texttt{)} = \texttt{nil} \vee \texttt{(iff } a\ b\texttt{)} = \texttt{t}$	Instantiation	
$\texttt{(iff } a\ b\texttt{)} \neq \texttt{nil}$	Given	
$\texttt{(iff } a\ b\texttt{)} = \texttt{t}$	Modus ponens 2	□

Derived Rule A-8. Disjoined iff t from not nil

$$\frac{P \vee \texttt{(iff } a\ b\texttt{)} \neq \texttt{nil}}{P \vee \texttt{(iff } a\ b\texttt{)} = \texttt{t}}$$

Derivation. (17)

$\texttt{(iff x y)} = \texttt{nil} \vee \texttt{(iff x y)} = \texttt{t}$	Th. iff nil or t	
$\texttt{(iff } a\ b\texttt{)} = \texttt{nil} \vee \texttt{(iff } a\ b\texttt{)} = \texttt{t}$	Instantiation	
$P \vee \texttt{(iff } a\ b\texttt{)} = \texttt{nil} \vee \texttt{(iff } a\ b\texttt{)} = \texttt{t}$	Expansion	
$P \vee \texttt{(iff } a\ b\texttt{)} \neq \texttt{nil}$	Given	
$P \vee \texttt{(iff } a\ b\texttt{)} = \texttt{t}$	Dj. mp2	□

Derived Rule A-9. Iff reflexivity

$$\frac{}{\texttt{(iff }\ a\ \ a\texttt{)} = \texttt{t}}$$

Derivation. (2)

$\texttt{(iff x x)} = \texttt{t}$	Th. refl. iff
$\texttt{(iff }\ a\ \ a\texttt{)} = \texttt{t}$	Instantiation □

Derived Rule A-10. Commute iff

$$\frac{\texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t}}{\texttt{(iff }\ b\ \ a\texttt{)} = \texttt{t}}$$

Derivation. (14)

$\texttt{(iff x y)} = \texttt{(iff y x)}$	Th. symmetry of iff
$\texttt{(iff }\ b\ \ a\texttt{)} = \texttt{(iff }\ a\ \ b\texttt{)}$	Instantiation
$\texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t}$	Given
$\texttt{(iff }\ b\ \ a\texttt{)} = \texttt{t}$	Trans. = □

Derived Rule A-11. Disjoined commute iff

$$\frac{P \vee \texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t}}{P \vee \texttt{(iff }\ b\ \ a\texttt{)} = \texttt{t}}$$

Derivation. (34)

$\texttt{(iff x y)} = \texttt{(iff y x)}$	Th. symmetry of iff
$\texttt{(iff }\ b\ \ a\texttt{)} = \texttt{(iff }\ a\ \ b\texttt{)}$	Instantiation
$P \vee \texttt{(iff }\ b\ \ a\texttt{)} = \texttt{(iff }\ a\ \ b\texttt{)}$	Expansion
$P \vee \texttt{(iff }\ a\ \ b\texttt{)} = \texttt{t}$	Given
$P \vee \texttt{(iff }\ b\ \ a\texttt{)} = \texttt{t}$	Dj. trans. = □

Formal Theorem A-16. Iff congruence lemma

$$\texttt{x} = \texttt{nil} \vee \texttt{y} = \texttt{nil} \vee \texttt{(if x a b)} = \texttt{(if y a b)}$$

Proof.

$\mathtt{x} = \mathtt{nil} \lor \mathtt{(if\ x\ y\ z)} = \mathtt{z}$	Ax. if when nnil	
$\mathtt{x} = \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{b}$	Instantiation	
$(\mathtt{x} = \mathtt{nil} \lor \mathtt{y} = \mathtt{nil}) \lor \mathtt{(if\ x\ a\ b)} = \mathtt{b}$	Multi assoc exp.	(*1)
$\mathtt{y} = \mathtt{nil} \lor \mathtt{(if\ y\ a\ b)} = \mathtt{b}$	Instantiation	
$(\mathtt{x} = \mathtt{nil} \lor \mathtt{y} = \mathtt{nil}) \lor \mathtt{(if\ y\ a\ b)} = \mathtt{b}$	Multi assoc exp.	
$(\mathtt{x} = \mathtt{nil} \lor \mathtt{y} = \mathtt{nil}) \lor \mathtt{b} = \mathtt{(if\ y\ a\ b)}$	Dj. commute =	
$(\mathtt{x} = \mathtt{nil} \lor \mathtt{y} = \mathtt{nil}) \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$	Dj. trans. = *1	
$\mathtt{x} = \mathtt{nil} \lor \mathtt{y} = \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$	Right assoc.	□

Formal Theorem A-17. Iff congruence lemma 2

$$\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} \neq \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$$

Proof.

$\mathtt{x} \neq \mathtt{nil} \lor \mathtt{(if\ x\ y\ z)} = \mathtt{y}$	Axiom if when nil	
$\mathtt{x} \neq \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{b}$	Instantiation	
$(\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} \neq \mathtt{nil}) \lor \mathtt{(if\ x\ a\ b)} = \mathtt{b}$	Multi assoc exp.	(*1)
$\mathtt{y} \neq \mathtt{nil} \lor \mathtt{(if\ y\ a\ b)} = \mathtt{b}$	Instantiation	
$(\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} \neq \mathtt{nil}) \lor \mathtt{(if\ y\ a\ b)} = \mathtt{b}$	Multi assoc exp.	
$(\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} \neq \mathtt{nil}) \lor \mathtt{b} = \mathtt{(if\ y\ a\ b)}$	Dj. commute =	
$(\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} \neq \mathtt{nil}) \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$	Dj. trans. = *1	
$\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} \neq \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} \neq \mathtt{(if\ y\ a\ b)}$	Right assoc.	□

Formal Theorem A-18. Iff congruent if 1

$$\mathtt{(iff\ x\ y)} = \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$$

Proof.

$\mathtt{x} = \mathtt{nil} \lor \mathtt{y} = \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$	Th. iff congruence lm.	
$\mathtt{x} \neq \mathtt{nil} \lor \mathtt{y} = \mathtt{nil} \lor \mathtt{(iff\ x\ y)} = \mathtt{nil}$	Th. iff false true	
$(\mathtt{y} = \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)})$ $\lor \mathtt{y} = \mathtt{nil} \lor \mathtt{(iff\ x\ y)} = \mathtt{nil}$	Cut	
$\mathtt{y} = \mathtt{nil} \lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$ $\lor \mathtt{y} = \mathtt{nil} \lor \mathtt{(iff\ x\ y)} = \mathtt{nil}$	Right assoc.	
$\mathtt{y} = \mathtt{nil} \lor \mathtt{(iff\ x\ y)} = \mathtt{nil}$ $\lor \mathtt{(if\ x\ a\ b)} = \mathtt{(if\ y\ a\ b)}$	Generic subset	(*1)

`x = nil ∨ y ≠ nil ∨ (iff x y) = nil`	Th. iff true false	
`x ≠ nil ∨ y ≠ nil ∨ (if x a b) = (if y a b)`	Th. iff congruence lm. 2	
`(y ≠ nil ∨ (iff x y) = nil)`	Cut	
`    ∨ y ≠ nil ∨ (if x a b) = (if y a b)`		
`y ≠ nil ∨ (iff x y) = nil`	Right assoc.	
`    ∨ y ≠ nil ∨ (if x a b) = (if y a b)`		
`y ≠ nil ∨ (iff x y) = nil`	Generic subset	
`    ∨ (if x a b) = (if y a b)`		
`((iff x y) = nil ∨ (if x a b) = (if y a b))`	Cut *1	
`    ∨ (iff x y) = nil`		
`    ∨ (if x a b) = (if y a b)`		
`(iff x y) = nil ∨ (if x a b) = (if y a b)`	Contraction	□

Formal Theorem A-19. Iff congruent iff 2

`(iff x y) = nil ∨ (iff z x) = (iff z y)`

Proof.

`x = nil ∨ (iff x y) = (if y t nil)`	Th. iff lhs true	
`z = nil ∨ (iff z x) = (if x t nil)`	Instantiation	(*1a)
`z = nil ∨ (iff z y) = (if y t nil)`	Instantiation	
`z = nil ∨ (if y t nil) = (iff z y)`	Dj. commute =	(*1b)
`(iff x y) = nil ∨ (if x a b) = (if y a b)`	Th. iff congruent if 1	
`(iff x y) = nil`	Instantiation	(*1c)
`    ∨ (if x nil t) = (if y t nil)`		
`z = nil ∨ (iff x y) = nil`	Multi assoc exp. *1a	
`    ∨ (iff z x) = (if x t nil)`		
`z = nil ∨ (iff x y) = nil`	Multi assoc exp. *1c	
`    ∨ (if x nil t) = (if y t nil)`		
`z = nil ∨ (iff x y) = nil`	Dj. trans. =	
`    ∨ (iff z x) = (if y t nil)`		
`z = nil ∨ (iff x y) = nil`	Multi assoc exp. *1b	
`    ∨ (if y t nil) = (iff z y)`		
`z = nil ∨ (iff x y) = nil`	Dj. trans. =	
`    ∨ (iff z x) = (iff z y)`		
`z = nil ∨ (iff x y) = nil`	Right assoc.	(*1)
`    ∨ (iff z x) = (iff z y)`		
`x ≠ nil ∨ (iff x y) = (if y nil t)`	Th. iff lhs false	
`z ≠ nil ∨ (iff z x) = (if x nil t)`	Instantiation	(*2a)
`z ≠ nil ∨ (iff z y) = (if y nil t)`	Instantiation	
`z ≠ nil ∨ (if y nil t) = (iff z y)`	Dj. commute =	(*2b)

$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(if x a b)} = \texttt{(if y a b)}$	Th. iff congruent if 1	
$\texttt{(iff x y)} = \texttt{nil}$ $\quad \lor \texttt{(if x nil t)} = \texttt{(if y nil t)}$	Instantiation	(*2c)
$(\texttt{z} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil})$ $\quad \lor \texttt{(iff z x)} = \texttt{(if x nil t)}$	Multi assoc exp. *2a	
$(\texttt{z} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil})$ $\quad \lor \texttt{(if x nil t)} = \texttt{(if y nil t)}$	Multi assoc exp. *2c	
$(\texttt{z} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil})$ $\quad \lor \texttt{(iff z x)} = \texttt{(if y nil t)}$	Dj. trans. =	
$(\texttt{z} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil})$ $\quad \lor \texttt{(if y nil t)} = \texttt{(iff z y)}$	Multi assoc exp. *2b	
$(\texttt{z} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil})$ $\quad \lor \texttt{(iff z x)} = \texttt{(iff z y)}$	Dj. trans. =	
$\texttt{z} \neq \texttt{nil} \lor \texttt{(iff x y)} = \texttt{nil}$ $\quad \lor \texttt{(iff z x)} = \texttt{(iff z y)}$	Right assoc.	(*2)
$(\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff z x)} = \texttt{(iff z y)})$ $\quad \lor \texttt{(iff x y)} = \texttt{nil}$ $\quad \lor \texttt{(iff z x)} = \texttt{(iff z y)}$	Cut *1, *2	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff z x)} = \texttt{(iff z y)}$	Contraction	□

Formal Theorem A-20. Iff congruent iff 1

$$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x z)} = \texttt{(iff y z)}$$

Proof.

$\texttt{(iff x y)} = \texttt{(iff y x)}$	Th. symmetry of iff	
$\texttt{(iff z y)} = \texttt{(iff y z)}$	Instantiation	
$\texttt{(iff z x)} = \texttt{(iff x z)}$	Instantiation	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff z y)} = \texttt{(iff y z)}$	Expansion	(*1a)
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff z x)} = \texttt{(iff x z)}$	Expansion	(*1b)
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff z x)} = \texttt{(iff z y)}$	Th. iff congruent iff 2	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff z x)} = \texttt{(iff y z)}$	Dj. trans. = *1a	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff y z)} = \texttt{(iff z x)}$	Dj. commute =	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff y z)} = \texttt{(iff x z)}$	Dj. trans. = *1b	
$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x z)} = \texttt{(iff y z)}$	Dj. commute =	□

Formal Theorem A-21. Iff of if x t nil

$$\texttt{(iff (if x t nil) x)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{nil} \lor \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$\texttt{x} = \texttt{nil} \lor \texttt{(if x t nil)} = \texttt{t}$	Instantiation	(*1a)
$\texttt{x} = \texttt{x}$	Reflexivity	
$\texttt{x} = \texttt{nil} \lor \texttt{x} = \texttt{x}$	Expansion	(*1b)
$\texttt{x} = \texttt{nil} \lor \texttt{(iff (if x t nil) x)} = \texttt{(iff t x)}$	Dj. = args *1ab	(*1c)
$\texttt{(iff x y)} = \texttt{(iff y x)}$	Th. symmetry of iff	
$\texttt{(iff t x)} = \texttt{(iff x t)}$	Instantiation	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff t x)} = \texttt{(iff x t)}$	Expansion	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff (if x t nil) x)} = \texttt{(iff x t)}$	Dj. trans. = *1c	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff x t)} = \texttt{t}$	Th. iff t, nnil	
$\texttt{x} = \texttt{nil} \lor \texttt{(iff (if x t nil) x)} = \texttt{t}$	Dj. trans. =	(*1)
$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(if x t nil)} = \texttt{nil}$	Instantiation	(*2a)
$\texttt{x} = \texttt{x}$	Reflexivity	
$\texttt{x} \neq \texttt{nil} \lor \texttt{x} = \texttt{x}$	Expansion	(*2b)
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff (if x t nil) x)} = \texttt{(iff nil x)}$	Dj. = args *2a, *2b	(*2c)
$\texttt{(iff x y)} = \texttt{(iff y x)}$	Th. symmetry of iff	
$\texttt{(iff nil x)} = \texttt{(iff x nil)}$	Instantiation	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff nil x)} = \texttt{(iff x nil)}$	Expansion	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff (if x t nil) x)} = \texttt{(iff x nil)}$	Dj. trans. = *2c	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff x nil)} = \texttt{t}$	Th. iff nil, nil	
$\texttt{x} \neq \texttt{nil} \lor \texttt{(iff (if x t nil) x)} = \texttt{t}$	Dj. trans. =	(*2)
$\texttt{(iff (if x t nil) x)} = \texttt{t}$ $\quad \lor \texttt{(iff (if x t nil) x)} = \texttt{t}$	Cut *1, *2	
$\texttt{(iff (if x t nil) x)} = \texttt{t}$	Contraction	□

Formal Theorem A-22. Transitivity of iff

$$\texttt{(iff x y)} \neq \texttt{t} \lor \texttt{(iff y z)} \neq \texttt{t} \lor \texttt{(iff x z)} = \texttt{t}$$

Proof.

$\texttt{(iff x y)} = \texttt{nil} \lor \texttt{(iff x z)} = \texttt{(iff y z)}$	Th. iff congruent iff 1	
$\texttt{(iff x z)} = \texttt{(iff y z)} \lor \texttt{(iff x y)} = \texttt{nil}$	Commute or	
$\texttt{(iff x z)} = \texttt{(iff y z)} \lor \texttt{(iff x y)} \neq \texttt{t}$	Dj. not t from nil	
$\texttt{(iff x y)} \neq \texttt{t} \lor \texttt{(iff x z)} = \texttt{(iff y z)}$	Commute or	
$(\texttt{(iff x y)} \neq \texttt{t} \lor \texttt{(iff y z)} \neq \texttt{t})$ $\quad \lor \texttt{(iff x z)} = \texttt{(iff y z)}$	Multi assoc exp.	(*1)
$\texttt{(iff y z)} \neq \texttt{t} \lor \texttt{(iff y z)} = \texttt{t}$	Prop. schema	

$((\texttt{iff x y}) \neq \texttt{t} \lor (\texttt{iff y z}) \neq \texttt{t})$ $\quad \lor (\texttt{iff y z}) = \texttt{t}$	Multi assoc exp.	
$((\texttt{iff x y}) \neq \texttt{t} \lor (\texttt{iff y z}) \neq \texttt{t})$ $\quad \lor (\texttt{iff x z}) = \texttt{t}$	Dj. trans. = *1	
$(\texttt{iff x y}) \neq \texttt{t} \lor (\texttt{iff y z}) \neq \texttt{t} \lor (\texttt{iff x z}) = \texttt{t}$	Right assoc.	□

Derived Rule A-12. Transitivity of iff

$$\begin{array}{l} (\texttt{iff}\ a\ b) = \texttt{t} \\ (\texttt{iff}\ b\ c) = \texttt{t} \\ \hline (\texttt{iff}\ a\ c) = \texttt{t} \end{array}$$

Derivation. (12)

$(\texttt{iff x y}) \neq \texttt{t} \lor (\texttt{iff y z}) \neq \texttt{t} \lor (\texttt{iff x z}) = \texttt{t}$	Th. trans. iff	
$(\texttt{iff}\ a\ b) \neq \texttt{t} \lor (\texttt{iff}\ b\ c) \neq \texttt{t} \lor (\texttt{iff}\ a\ c) = \texttt{t}$	Instantiation	(*1)
$(\texttt{iff}\ a\ b) = \texttt{t}$	Given	
$(\texttt{iff}\ b\ c) \neq \texttt{t} \lor (\texttt{iff}\ a\ c) = \texttt{t}$	Modus ponens *1	(*2)
$(\texttt{iff}\ b\ c) = \texttt{t}$	Given	
$(\texttt{iff}\ a\ c) = \texttt{t}$	Modus ponens *2	□

Derived Rule A-13. Disjoined transitivity of iff

$$\begin{array}{l} P \lor (\texttt{iff}\ a\ b) = \texttt{t} \\ P \lor (\texttt{iff}\ b\ c) = \texttt{t} \\ \hline P \lor (\texttt{iff}\ a\ c) = \texttt{t} \end{array}$$

Derivation. (31)

$(\texttt{iff x y}) \neq \texttt{t} \lor (\texttt{iff y z}) \neq \texttt{t} \lor (\texttt{iff x z}) = \texttt{t}$	Th. trans. iff	
$(\texttt{iff}\ a\ b) \neq \texttt{t} \lor (\texttt{iff}\ b\ c) \neq \texttt{t} \lor (\texttt{iff}\ a\ c) = \texttt{t}$	Instantiation	
$P \lor (\texttt{iff}\ a\ b) \neq \texttt{t}$ $\quad \lor (\texttt{iff}\ b\ c) \neq \texttt{t} \lor (\texttt{iff}\ a\ c) = \texttt{t}$	Expansion	
$P \lor (\texttt{iff}\ a\ b) = \texttt{t}$	Given	
$P \lor (\texttt{iff}\ b\ c) \neq \texttt{t} \lor (\texttt{iff}\ a\ c) = \texttt{t}$	Dj. modus ponens	
$P \lor (\texttt{iff}\ b\ c) = \texttt{t}$	Given	
$P \lor (\texttt{iff}\ a\ c) \neq \texttt{t}$	Dj. modus ponens	□

Formal Theorem A-23. Iff from =

$$\texttt{x} \neq \texttt{y} \lor \texttt{(iff x y)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{x}$	Reflexivity	
$\texttt{x} \neq \texttt{y} \lor \texttt{x} = \texttt{x}$	Expansion	(*1a)
$\texttt{x} \neq \texttt{y} \lor \texttt{x} = \texttt{y}$	Prop. schema	
$\texttt{x} \neq \texttt{y} \lor \texttt{y} = \texttt{x}$	Dj. commute =	(*1b)
$\texttt{x} \neq \texttt{y} \lor \texttt{(iff x y)} = \texttt{(iff x x)}$	Dj. = args *1ab	(*1)
$\texttt{(iff x x)} = \texttt{t}$	Th. refl. iff	
$\texttt{x} \neq \texttt{y} \lor \texttt{(iff x x)} = \texttt{t}$	Expansion	
$\texttt{x} \neq \texttt{y} \lor \texttt{(iff x y)} = \texttt{t}$	Dj. trans. = *1	□

Derived Rule A-14. Iff from =

$$\frac{a = b}{\texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derivation. (7)

$\texttt{x} \neq \texttt{y} \lor \texttt{(iff x y)} = \texttt{t}$	Th. iff from =	
$a \neq b \lor \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Instantiation	
$a = b$	Given	
$\texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Modus ponens	□

Derived Rule A-15. Disjoined iff from =

$$\frac{P \lor a = b}{P \lor \texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derivation. (17)

$\texttt{x} \neq \texttt{y} \lor \texttt{(iff x y)} = \texttt{t}$	Th. iff from =	
$a \neq b \lor \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Instantiation	
$P \lor a \neq b \lor \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Expansion	
$P \lor a = b$	Given	
$P \lor \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Dj. modus ponens	□

Formal Theorem A-24. Iff from equal

$$\texttt{(equal x y)} \neq \texttt{t} \vee \texttt{(iff x y)} = \texttt{t}$$

Proof.

$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} = \texttt{nil}$	Ax. eq. when diff
$\texttt{x} = \texttt{y} \vee \texttt{(equal x y)} \neq \texttt{t}$	Dj. not t from nil
$\texttt{x} \neq \texttt{y} \vee \texttt{(iff x y)} = \texttt{t}$	Th. iff from =
$\texttt{(equal x y)} \neq \texttt{t} \vee \texttt{(iff x y)} = \texttt{t}$	Cut □

Derived Rule A-16. Iff from equal

$$\frac{\texttt{(equal } a \ b\texttt{)} = \texttt{t}}{\texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derivation. (7)

$\texttt{(equal x y)} \neq \texttt{t} \vee \texttt{(iff x y)} = \texttt{t}$	Th. iff from equal
$\texttt{(equal } a \ b\texttt{)} \neq \texttt{t} \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Instantiation
$\texttt{(equal } a \ b\texttt{)} = \texttt{t}$	Given
$\texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Modus ponens □

Derived Rule A-17. Disjoined iff from equal

$$\frac{P \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}}{P \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}}$$

Derivation. (17)

$\texttt{(equal x y)} \neq \texttt{t} \vee \texttt{(iff x y)} = \texttt{t}$	Th. iff from equal
$\texttt{(equal } a \ b\texttt{)} \neq \texttt{t} \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Instantiation
$P \vee \texttt{(equal } a \ b\texttt{)} \neq \texttt{t} \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Expansion
$P \vee \texttt{(equal } a \ b\texttt{)} = \texttt{t}$	Given
$P \vee \texttt{(iff } a \ b\texttt{)} = \texttt{t}$	Dj. modus ponens □

Derived Rule A-18. Negative lit from $\neq$ nil

$$\frac{a \neq \texttt{nil}}{\texttt{(not }a\texttt{)} = \texttt{nil}}$$

Derivation. (7)

$\texttt{x} = \texttt{nil} \vee \texttt{(not x)} = \texttt{nil}$	Th. not when nnil
$a = \texttt{nil} \vee \texttt{(not }a\texttt{)} = \texttt{nil}$	Instantiation
$a \neq \texttt{nil}$	Given
$\texttt{(not }a\texttt{)} = \texttt{nil}$	Modus ponens 2 $\square$

Derived Rule A-19. Disjoined negative lit from = nil

$$\frac{P \vee a = \texttt{nil}}{P \vee \texttt{(not }a\texttt{)} \neq \texttt{nil}}$$

Derivation. (34)

$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} = \texttt{t}$	Th. not when nil
$\texttt{x} \neq \texttt{nil} \vee \texttt{(not x)} \neq \texttt{nil}$	Dj. not nil from t
$a \neq \texttt{nil} \vee \texttt{(not }a\texttt{)} \neq \texttt{nil}$	Instantiation
$P \vee a \neq \texttt{nil} \vee \texttt{(not }a\texttt{)} \neq \texttt{nil}$	Expansion
$P \vee a = \texttt{nil}$	Given
$P \vee \texttt{(not }a\texttt{)} \neq \texttt{nil}$	Dj. modus ponens $\square$

Derived Rule A-20. Substitute iff into literal

$$\frac{\begin{array}{l} b \neq \texttt{nil} \\ \texttt{(iff }a\ b\texttt{)} = \texttt{t} \end{array}}{a \neq \texttt{nil}}$$

Derivation. (35)

$b \neq \texttt{nil}$	Given
$\texttt{(iff }b\texttt{ t)} = \texttt{t}$	Iff t from $\neq$ nil
$\texttt{(iff }a\ b\texttt{)} = \texttt{t}$	Given
$\texttt{(iff }a\texttt{ t)} = \texttt{t}$	Transitivity of iff
$\texttt{(iff }a\texttt{ t)} \neq \texttt{nil}$	Not nil from t
$a \neq \texttt{nil}$	$\neq$ nil from iff t $\square$

Derived Rule A-21. Disjoined substitute iff into literal

$$\frac{\begin{array}{l} P \vee b \neq \texttt{nil} \\ P \vee \texttt{(iff}\ \ a\ \ b\texttt{)} = \texttt{t} \end{array}}{P \vee a \neq \texttt{nil}}$$

Derivation. (84)

$P \vee b \neq \texttt{nil}$	Given
$P \vee \texttt{(iff}\ \ b\ \ \texttt{t)} = \texttt{t}$	Dj. iff t fr. $\neq$ nil
$P \vee \texttt{(iff}\ \ a\ \ b\texttt{)} = \texttt{t}$	Given
$P \vee \texttt{(iff}\ \ a\ \ \texttt{t)} = \texttt{t}$	Dj. trans. iff
$P \vee \texttt{(iff}\ \ a\ \ \texttt{t)} \neq \texttt{nil}$	Dj. not nil from t
$P \vee a \neq \texttt{nil}$	Dj. $\neq$ nil fr. iff t $\square$

Appendix B

Derivations for Clause Splitting

In this appendix, we present the details behind the formal theorems and derived rules which were only summarized in Section 7.5. To make some derivations more efficient, we begin by introducing a few optimized rules for certain kinds of propositional manipulation.

Derived Rule B-1. Aux split twiddle lemma 1

$$\frac{(A \vee C) \vee B \vee C}{((B \vee C) \vee A \vee B \vee C) \vee A}$$

Derivation. (10)

$(A \vee C) \vee B \vee C$	Given
$(B \vee C) \vee A \vee C$	Commute or
$A \vee (B \vee C) \vee A \vee C$	Expansion
$(A \vee B \vee C) \vee A \vee C$	Associativity
$((A \vee B \vee C) \vee A) \vee C$	Associativity
$C \vee (A \vee B \vee C) \vee A$	Commute or
$B \vee C \vee (A \vee B \vee C) \vee A$	Expansion
$(B \vee C) \vee (A \vee B \vee C) \vee A$	Associativity
$((B \vee C) \vee A \vee B \vee C) \vee A$	Associativity □

Derived Rule B-2. Aux split twiddle

$$\frac{(A \vee C) \vee B \vee C}{A \vee B \vee C}$$

Derivation. (14)

$(A \vee C) \vee B \vee C$	Given

$((B \lor C) \lor A \lor B \lor C) \lor A$	Aux split twiddle lm. 1
$A \lor (B \lor C) \lor A \lor B \lor C$	Commute or
$(A \lor B \lor C) \lor A \lor B \lor C$	Associativity
$A \lor B \lor C$	Contraction $\square$

Derived Rule B-3. Aux split twiddle2 lemma 1a

$$\frac{Q \lor A \lor C}{A \lor B \lor C \lor P \lor Q}$$

Derivation. (18)

$Q \lor A \lor C$	Given
$P \lor Q \lor A \lor C$	Expansion
$(P \lor Q) \lor A \lor C$	Associativity
$(A \lor C) \lor P \lor Q$	Commute or
$A \lor C \lor P \lor Q$	Right assoc.
$A \lor B \lor C \lor P \lor Q$	Dj. left expansion $\square$

Derived Rule B-4. Aux split twiddle2 lemma 1

$$\frac{Q \lor A \lor C}{((P \lor Q) \lor A \lor B) \lor C}$$

Derivation. (23)

$Q \lor A \lor C$	Given
$A \lor B \lor C \lor P \lor Q$	Aux split twiddle2 lm. 1a
$(A \lor B) \lor C \lor P \lor Q$	Associativity
$((A \lor B) \lor C) \lor P \lor Q$	Associativity
$(P \lor Q) \lor (A \lor B) \lor C$	Commute or
$((P \lor Q) \lor A \lor B) \lor C$	Associativity $\square$

Derived Rule B-5. Aux split twiddle2 lemma 2a

$$\frac{C \lor B \lor P}{A \lor B \lor (P \lor Q) \lor C}$$

Derivation. (10)

$C \vee B \vee P$	Given
$(C \vee B) \vee P$	Associativity
$Q \vee (C \vee B) \vee P$	Expansion
$(Q \vee C \vee B) \vee P$	Associativity
$P \vee Q \vee C \vee B$	Commute or
$(P \vee Q) \vee C \vee B$	Associativity
$((P \vee Q) \vee C) \vee B$	Associativity
$B \vee (P \vee Q) \vee C$	Commute or
$A \vee B \vee (P \vee Q) \vee C$	Expansion □

Derived Rule B-6. Aux split twiddle2 lemma 2

$$\frac{C \vee B \vee P}{C \vee (P \vee Q) \vee A \vee B}$$

Derivation. (34)

$C \vee B \vee P$	Given
$A \vee B \vee (P \vee Q) \vee C$	Aux split twiddle2 lm. 2a
$(A \vee B) \vee (P \vee Q) \vee C$	Associativity
$((A \vee B) \vee P \vee Q) \vee C$	Associativity
$C \vee (A \vee B) \vee P \vee Q$	Commute or
$C \vee (P \vee Q) \vee A \vee B$	Dj. commute or □

Derived Rule B-7. Aux split twiddle2

$$\frac{(A \vee B \vee P) \vee Q}{(P \vee Q) \vee A \vee B}$$

Derivation. (60)

$(A \vee B \vee P) \vee Q$	Given
$Q \vee A \vee B \vee P$	Commute or
$((P \vee Q) \vee A \vee B) \vee B \vee P$	Aux split twiddle2 lm. 1
$((P \vee Q) \vee A \vee B) \vee (P \vee Q) \vee A \vee B$	Aux split twiddle2 lm. 2
$(P \vee Q) \vee A \vee B$	Contraction □

We can now introduce the derivations used throughout the proof of CS-AUX.

The rules for Line 2 are straightforward.

Derived Rule B-8. Aux split double negate lemma1

$$\frac{(b \neq \texttt{nil} \vee P) \vee Q \quad \texttt{(iff}\ a\ b\texttt{)} = \texttt{t}}{(a \neq \texttt{nil} \vee P) \vee Q}$$

Derivation. (98)

$(b \neq \texttt{nil} \vee P) \vee Q$	Given	
$b \neq \texttt{nil} \vee P \vee Q$	Right assoc.	
$(P \vee Q) \vee b \neq \texttt{nil}$	Commute or	(*1)
$\texttt{(iff}\ a\ b\texttt{)} = \texttt{t}$	Given	
$(P \vee Q) \vee \texttt{(iff}\ a\ b\texttt{)} = \texttt{t}$	Expansion	
$(P \vee Q) \vee a \neq \texttt{nil}$	Dj. sub. iff into literal *1	
$a \neq \texttt{nil} \vee P \vee Q$	Commute or	
$(a \neq \texttt{nil} \vee P) \vee Q$	Associativity	□

Derived Rule B-9. Aux split double negate lemma2

$$\frac{b \neq \texttt{nil} \vee P \quad \texttt{(iff}\ a\ b\texttt{)} = \texttt{t}}{a \neq \texttt{nil} \vee P}$$

Derivation. (89)

$b \neq \texttt{nil} \vee P$	Given	
$P \vee b \neq \texttt{nil}$	Commute or	(*1)
$\texttt{(iff}\ a\ b\texttt{)} = \texttt{t}$	Given	
$P \vee \texttt{(iff}\ a\ b\texttt{)} = \texttt{t}$	Expansion	
$P \vee a \neq \texttt{nil}$	Dj. sub. iff into literal *1	
$a \neq \texttt{nil} \vee P$	Commute or	□

For line 3, we begin by introducing a theorem which performs for the main part of the derivation, and by developing the usual rules to allow us to instantiate this theorem.

Formal Theorem B-1. Aux split negative

$$\neg(\texttt{(not x)} \neq \texttt{nil} \vee \texttt{(not y)} \neq \texttt{nil}) \\ \vee \neg(\texttt{x} \neq \texttt{nil} \vee \texttt{(not z)} \neq \texttt{nil}) \vee \texttt{(not (if x y z))} \neq \texttt{nil}$$

Proof.

In the derivation below, we let P be $(\texttt{(not x)} \neq \texttt{nil} \vee \texttt{(not y)} \neq \texttt{nil})$, and let Q be $(\texttt{x} \neq \texttt{nil} \vee \texttt{(not z)} \neq \texttt{nil})$.

$\texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$\neg P \vee \texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{y}$	Expansion	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{(if x y z)} = \texttt{y}$	Associativity	(*1a)
$\neg P \vee \texttt{(not x)} \neq \texttt{nil} \vee \texttt{(not y)} \neq \texttt{nil}$	Prop. schema	
$(\neg P \vee \texttt{(not x)} \neq \texttt{nil}) \vee \texttt{(not y)} \neq \texttt{nil}$	Associativity	
$(\neg P \vee \texttt{(not x)} \neq \texttt{nil}) \vee \texttt{y} = \texttt{nil}$	Dj. = nil fr. neg. lit	
$\neg P \vee \texttt{(not x)} \neq \texttt{nil} \vee \texttt{y} = \texttt{nil}$	Right assoc.	
$\neg P \vee \texttt{y} = \texttt{nil} \vee \texttt{(not x)} \neq \texttt{nil}$	Dj. commute or	
$(\neg P \vee \texttt{y} = \texttt{nil}) \vee \texttt{(not x)} \neq \texttt{nil}$	Associativity	
$(\neg P \vee \texttt{y} = \texttt{nil}) \vee \texttt{x} = \texttt{nil}$	Dj. = nil fr. neg. lit	
$\neg P \vee \texttt{y} = \texttt{nil} \vee \texttt{x} = \texttt{nil}$	Right assoc.	
$\neg P \vee \texttt{x} = \texttt{nil} \vee \texttt{y} = \texttt{nil}$	Dj. commute or	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{y} = \texttt{nil}$	Associativity	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{(if x y z)} = \texttt{nil}$	Dj. trans. = *1a	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{(not (if x y z))} \neq \texttt{nil}$	Dj. neg. lit fr. = nil	
$\texttt{(not (if x y z))} \neq \texttt{nil} \vee \neg P \vee \texttt{x} = \texttt{nil}$	Commute or	
$(\texttt{(not (if x y z))} \neq \texttt{nil} \vee \neg P) \vee \texttt{x} = \texttt{nil}$	Associativity	
$\texttt{x} = \texttt{nil} \vee \texttt{(not (if x y z))} \neq \texttt{nil} \vee \neg P$	Commute or	
$\texttt{x} = \texttt{nil} \vee \neg P \vee \texttt{(not (if x y z))} \neq \texttt{nil}$	Dj. commute or	(*1)
$\texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$\neg Q \vee \texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Expansion	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{(if x y z)} = \texttt{z}$	Associativity	(*2a)
$\neg Q \vee \texttt{x} \neq \texttt{nil} \vee \texttt{(not z)} \neq \texttt{nil}$	Prop. schema	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{(not z)} \neq \texttt{nil}$	Associativity	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{z} = \texttt{nil}$	Dj. = nil fr. neg. lit	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{(if x y z)} = \texttt{nil}$	Dj. trans. = *2a	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{(not (if x y z))} \neq \texttt{nil}$	Dj. neg. lit fr. = nil	
$\texttt{(not (if x y z))} \neq \texttt{nil} \vee \neg Q \vee \texttt{x} \neq \texttt{nil}$	Commute or	
$\texttt{(not (if x y z))} \neq \texttt{nil} \vee \texttt{x} \neq \texttt{nil} \vee \neg Q$	Dj. commute or	
$(\texttt{x} \neq \texttt{nil} \vee \neg Q) \vee \texttt{(not (if x y z))} \neq \texttt{nil}$	Commute or	
$\texttt{x} \neq \texttt{nil} \vee \neg Q \vee \texttt{(not (if x y z))} \neq \texttt{nil}$	Right assoc.	(*2)

$(\neg P \vee$ `(not (if x y z))` $\neq$ `nil`$)$ $\vee \neg Q \vee$ `(not (if x y z))` $\neq$ `nil`	Cut *1, *2
$\neg P \vee \neg Q \vee$ `(not (if x y z))` $\neq$ `nil`	Aux split twiddle $\square$

Derived Rule B-10. Aux split negative

$$\frac{\begin{array}{l}\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil} \\ a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}\end{array}}{\texttt{(not (if } a\ b\ c\texttt{))} \neq \texttt{nil}}$$

Derivation. (12)

$\neg($`(not x)` $\neq$ `nil` $\vee$ `(not y)` $\neq$ `nil`$)$ $\vee \neg($`x` $\neq$ `nil` $\vee$ `(not z)` $\neq$ `nil`$)$ $\vee$ `(not (if x y z))` $\neq$ `nil`	Th. aux split neg.
$\neg($`(not` a`)` $\neq$ `nil` $\vee$ `(not` b`)` $\neq$ `nil`$)$ $\vee \neg(a \neq$ `nil` $\vee$ `(not` c`)` $\neq$ `nil`$)$ $\vee$ `(not (if` $a\ b\ c$`))` $\neq$ `nil`	Instantiation
`(not` a`)` $\neq$ `nil` $\vee$ `(not` b`)` $\neq$ `nil`	Given
$\neg(a \neq$ `nil` $\vee$ `(not` c`)` $\neq$ `nil`$)$ $\vee$ `(not (if` $a\ b\ c$`))` $\neq$ `nil`	Modus ponens
$a \neq$ `nil` $\vee$ `(not` c`)` $\neq$ `nil`	Given
`(not (if` $a\ b\ c$`))` $\neq$ `nil`	Modus ponens $\square$

Derived Rule B-11. Disjoined aux split negative

$$\frac{\begin{array}{l}P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil} \\ P \vee a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}\end{array}}{P \vee \texttt{(not (if } a\ b\ c\texttt{))} \neq \texttt{nil}}$$

Derivation. (31)

$\neg($`(not x)` $\neq$ `nil` $\vee$ `(not y)` $\neq$ `nil`$)$ $\vee \neg($`x` $\neq$ `nil` $\vee$ `(not z)` $\neq$ `nil`$)$ $\vee$ `(not (if x y z))` $\neq$ `nil`	Th. aux split neg.
$\neg($`(not` a`)` $\neq$ `nil` $\vee$ `(not` b`)` $\neq$ `nil`$)$ $\vee \neg(a \neq$ `nil` $\vee$ `(not` c`)` $\neq$ `nil`$)$ $\vee$ `(not (if` $a\ b\ c$`))` $\neq$ `nil`	Instantiation
$P \vee \neg($`(not` a`)` $\neq$ `nil` $\vee$ `(not` b`)` $\neq$ `nil`$)$ $\vee \neg(a \neq$ `nil` $\vee$ `(not` c`)` $\neq$ `nil`$)$ $\vee$ `(not (if` $a\ b\ c$`))` $\neq$ `nil`	Expansion

$P \lor (\texttt{not}\ a) \neq \texttt{nil} \lor (\texttt{not}\ b) \neq \texttt{nil}$	Given	
$P \lor \neg(a \neq \texttt{nil} \lor (\texttt{not}\ c) \neq \texttt{nil})$ $\lor\ (\texttt{not}\ (\texttt{if}\ a\ b\ c)) \neq \texttt{nil}$	Dj. modus ponens	
$P \lor a \neq \texttt{nil} \lor (\texttt{not}\ c) \neq \texttt{nil}$	Given	
$P \lor (\texttt{not}\ (\texttt{if}\ a\ b\ c)) \neq \texttt{nil}$	Dj. modus ponens	$\square$

We are now ready for the main rules that we use in the proof of Line 3.

Derived Rule B-12. Aux split negative 1 lemma 1

$$\begin{array}{l} ((\texttt{not}\ a) \neq \texttt{nil} \lor (\texttt{not}\ b) \neq \texttt{nil} \lor P) \lor Q \\ (a \neq \texttt{nil} \lor (\texttt{not}\ c) \neq \texttt{nil} \lor P) \lor Q \\ \hline (P \lor Q) \lor (\texttt{not}\ (\texttt{if}\ a\ b\ c)) \neq \texttt{nil} \end{array}$$

Derivation. (151)

$((\texttt{not}\ a) \neq \texttt{nil} \lor (\texttt{not}\ b) \neq \texttt{nil} \lor P) \lor Q$	Given	
$(P \lor Q) \lor (\texttt{not}\ a) \neq \texttt{nil} \lor (\texttt{not}\ b) \neq \texttt{nil}$	Aux split twiddle2	(*1)
$(a \neq \texttt{nil} \lor (\texttt{not}\ c) \neq \texttt{nil} \lor P) \lor Q$	Given	
$(P \lor Q) \lor a \neq \texttt{nil} \lor (\texttt{not}\ c) \neq \texttt{nil}$	Aux split twiddle2	
$(P \lor Q) \lor (\texttt{not}\ (\texttt{if}\ a\ b\ c)) \neq \texttt{nil}$	Dj. aux split neg. *1	$\square$

Derived Rule B-13. Aux split negative 1 lemma 2

$$\begin{array}{l} (P \lor Q) \lor (\texttt{not}\ a) \neq \texttt{nil} \\ t1 = (\texttt{not}\ a) \\ \hline (t1 \neq \texttt{nil} \lor P) \lor Q \end{array}$$

Derivation. (35)

$(P \lor Q) \lor (\texttt{not}\ a) \neq \texttt{nil}$	Given	(*1)
$t1 = (\texttt{not}\ a)$	Given	
$(P \lor Q) \lor t1 = (\texttt{not}\ a)$	Expansion	
$(P \lor Q) \lor t1 \neq \texttt{nil}$	Dj. sub. into $\neq$ *1	
$t1 \neq \texttt{nil} \lor P \lor Q$	Commute or	
$(t1 \neq \texttt{nil} \lor P) \lor Q$	Associativity	$\square$

Derived Rule B-14. Aux split negative 1

$$\frac{\begin{array}{l}(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil} \vee P) \vee Q \\ (a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil} \vee P) \vee Q \\ \mathit{t1} = \texttt{(not (if } a\ b\ c\texttt{))}\end{array}}{(\mathit{t1} \neq \texttt{nil} \vee P) \vee Q}$$

Derivation. (186)

$(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil} \vee P) \vee Q$	Given	
$(a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil} \vee P) \vee Q$	Given	
$(P \vee Q) \vee \texttt{(not (if } a\ b\ c\texttt{))} \neq \texttt{nil}$	Aux split neg. 1 lm. 1	
$\mathit{t1} = \texttt{(not (if } a\ b\ c\texttt{))}$	Given	
$(\mathit{t1} \neq \texttt{nil} \vee P) \vee Q$	Aux split neg. 1 lm. 2	□

Derived Rule B-15. Aux split negative 2 lemma 1

$$\frac{\begin{array}{l}(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil}) \vee P \\ (a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}) \vee P\end{array}}{P \vee \texttt{(not (if } a\ b\ c\texttt{))} \neq \texttt{nil}}$$

Derivation. (35)

$(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil}) \vee P$	Given	
$P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee \texttt{(not } b\texttt{)} \neq \texttt{nil}$	Commute or	(*1)
$(a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}) \vee P$	Given	
$P \vee a \neq \texttt{nil} \vee \texttt{(not } c\texttt{)} \neq \texttt{nil}$	Commute or	(*2)
$P \vee \texttt{(not (if } a\ b\ c\texttt{))} \neq \texttt{nil}$	Dj. aux split neg. *1, *2	□

Derived Rule B-16. Aux split negative 2 lemma 2

$$\frac{\begin{array}{l}\mathit{t1} = \texttt{(not } a\texttt{)} \\ P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil}\end{array}}{\mathit{t1} \neq \texttt{nil} \vee P}$$

Derivation. (34)

$\mathit{t1} = \texttt{(not } a\texttt{)}$	Given	
$P \vee \mathit{t1} = \texttt{(not } a\texttt{)}$	Expansion	
$P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil}$	Given	
$P \vee \mathit{t1} \neq \texttt{nil}$	Dj. sub. into $\neq$	
$\mathit{t1} \neq \texttt{nil} \vee P$	Commute or	□

Derived Rule B-17. Aux split negative 2

$$\frac{\begin{array}{l}(\texttt{(not }a\texttt{)} \neq \texttt{nil} \vee \texttt{(not }b\texttt{)} \neq \texttt{nil}) \vee P \\ (a \neq \texttt{nil} \vee \texttt{(not }c\texttt{)} \neq \texttt{nil}) \vee P \\ t1 = \texttt{(not (if }a\ b\ c\texttt{))}\end{array}}{t1 \neq \texttt{nil} \vee P}$$

Derivation. (69)

$(\texttt{(not }a\texttt{)} \neq \texttt{nil} \vee \texttt{(not }b\texttt{)} \neq \texttt{nil}) \vee P$	Given
$(a \neq \texttt{nil} \vee \texttt{(not }c\texttt{)} \neq \texttt{nil}) \vee P$	Given
$P \vee \texttt{(not (if }a\ b\ c\texttt{))} \neq \texttt{nil}$	Aux split neg. 2 lm. 1
$t1 = \texttt{(not (if }a\ b\ c\texttt{))}$	Given
$t1 \neq \texttt{nil} \vee P$	Aux split neg. 2 lm. 2 □

Our work for line 4 is similar. We begin with a theorem that does the main part of the derivation. We can then instantiate that theorem and manipulate the result as needed.

Formal Theorem B-2. Aux split positive

$$\begin{array}{l}\neg(\texttt{(not x)} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil}) \\ \quad \vee \neg(\texttt{x} \neq \texttt{nil} \vee \texttt{z} \neq \texttt{nil}) \vee \texttt{(if x y z)} \neq \texttt{nil}\end{array}$$

Proof.

In the derivation below, we let P be $(\texttt{(not x)} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil})$ and let Q be $(\texttt{x} \neq \texttt{nil} \vee \texttt{z} \neq \texttt{nil})$.

$\texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{y}$	Ax. if when nnil	
$\neg P \vee \texttt{x} = \texttt{nil} \vee \texttt{(if x y z)} = \texttt{y}$	Expansion	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{(if x y z)} = \texttt{y}$	Associativity	(*1a)
$\neg P \vee \texttt{(not x)} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil}$	Prop. schema	
$\neg P \vee \texttt{y} \neq \texttt{nil} \vee \texttt{(not x)} \neq \texttt{nil}$	Dj. commute or	
$(\neg P \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{(not x)} \neq \texttt{nil}$	Associativity	
$(\neg P \vee \texttt{y} \neq \texttt{nil}) \vee \texttt{x} = \texttt{nil}$	Dj. = nil fr. neg. lit	
$\neg P \vee \texttt{y} \neq \texttt{nil} \vee \texttt{x} = \texttt{nil}$	Right assoc.	
$\neg P \vee \texttt{x} = \texttt{nil} \vee \texttt{y} \neq \texttt{nil}$	Dj. commute or	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{y} \neq \texttt{nil}$	Associativity	
$(\neg P \vee \texttt{x} = \texttt{nil}) \vee \texttt{(if x y z)} \neq \texttt{nil}$	Dj. sub. into $\neq$ *1a	

$\texttt{(if x y z)} \neq \texttt{nil} \vee \neg P \vee \texttt{x} = \texttt{nil}$	Commute or	
$\texttt{(if x y z)} \neq \texttt{nil} \vee \texttt{x} = \texttt{nil} \vee \neg P$	Dj. commute or	
$(\texttt{x} = \texttt{nil} \vee \neg P) \vee \texttt{(if x y z)} \neq \texttt{nil}$	Commute or	
$\texttt{x} = \texttt{nil} \vee \neg P \vee \texttt{(if x y z)} \neq \texttt{nil}$	Right assoc.	(*1)
$\texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Axiom if when nil	
$\neg Q \vee \texttt{x} \neq \texttt{nil} \vee \texttt{(if x y z)} = \texttt{z}$	Expansion	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{(if x y z)} = \texttt{z}$	Associativity	(*2a)
$\neg Q \vee \texttt{x} \neq \texttt{nil} \vee \texttt{z} \neq \texttt{nil}$	Prop. schema	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{z} \neq \texttt{nil}$	Associativity	
$(\neg Q \vee \texttt{x} \neq \texttt{nil}) \vee \texttt{(if x y z)} \neq \texttt{nil}$	Dj. sub. into $\neq$ *2a	
$\texttt{(if x y z)} \neq \texttt{nil} \vee \neg Q \vee \texttt{x} \neq \texttt{nil}$	Commute or	
$\texttt{(if x y z)} \neq \texttt{nil} \vee \texttt{x} \neq \texttt{nil} \vee \neg Q$	Dj. commute or	
$(\texttt{x} \neq \texttt{nil} \vee \neg Q) \vee \texttt{(if x y z)} \neq \texttt{nil}$	Commute or	
$\texttt{x} \neq \texttt{nil} \vee \neg Q \vee \texttt{(if x y z)} \neq \texttt{nil}$	Right assoc.	(*2)
$(\neg P \vee \texttt{(if x y z)} \neq \texttt{nil})$ $\vee \neg Q \vee \texttt{(if x y z)} \neq \texttt{nil}$	Cut *1, *2	
$\neg P \vee \neg Q \vee \texttt{(if x y z)} \neq \texttt{nil}$	Aux split twiddle	□

Derived Rule B-18. Aux split positive

$$\frac{\begin{array}{l}\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \\ a \neq \texttt{nil} \vee c \neq \texttt{nil}\end{array}}{\texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}}$$

Derivation. (12)

$\neg(\texttt{(not x)} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil})$ $\vee \neg(\texttt{x} \neq \texttt{nil} \vee \texttt{z} \neq \texttt{nil}) \vee \texttt{(if x y z)} \neq \texttt{nil}$	Th. aux split positive	
$\neg(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil})$ $\vee \neg(a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Instantiation	
$\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}$	Given	
$\neg(a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Modus ponens	
$a \neq \texttt{nil} \vee c \neq \texttt{nil}$	Given	
$\texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Modus ponens	□

Derived Rule B-19. Disjoined aux split positive

$$\frac{\begin{array}{l}P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \\ P \vee a \neq \texttt{nil} \vee c \neq \texttt{nil}\end{array}}{P \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}}$$

Derivation. (31)

$\neg(\texttt{(not x)} \neq \texttt{nil} \vee \texttt{y} \neq \texttt{nil})$ $\quad\vee \neg(\texttt{x} \neq \texttt{nil} \vee \texttt{z} \neq \texttt{nil}) \vee \texttt{(if x y z)} \neq \texttt{nil}$	Th. aux split positive	
$\neg(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil})$ $\quad\vee \neg(a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Instantiation	
$P \vee \neg(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil})$ $\quad\vee \neg(a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Expansion	
$P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}$	Given	
$P \vee \neg(a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Dj. modus ponens	
$P \vee a \neq \texttt{nil} \vee c \neq \texttt{nil}$	Given	
$P \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Dj. modus ponens	$\square$

Derived Rule B-20. Aux split positive 1

$$\frac{\begin{array}{l}(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \vee P) \vee Q \\ (a \neq \texttt{nil} \vee c \neq \texttt{nil} \vee P) \vee Q\end{array}}{(\texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil} \vee P) \vee Q}$$

Derivation. (154)

$(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil} \vee P) \vee Q$	Given	
$(P \vee Q) \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}$	Aux split twiddle2	(*1)
$(a \neq \texttt{nil} \vee c \neq \texttt{nil} \vee P) \vee Q$	Given	
$(P \vee Q) \vee a \neq \texttt{nil} \vee c \neq \texttt{nil}$	Aux split twiddle2	(*2)
$(P \vee Q) \vee \texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil}$	Dj. aux split positive *1, *2	
$\texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil} \vee P \vee Q$	Commute or	
$(\texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil} \vee P) \vee Q$	Associativity	$\square$

Derived Rule B-21. Aux split positive 2

$$\frac{\begin{array}{l}(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}) \vee P \\ (a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee P\end{array}}{\texttt{(if } a\ b\ c\texttt{)} \neq \texttt{nil} \vee P}$$

Derivation. (37)

$(\texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}) \vee P$	Given	
$P \vee \texttt{(not } a\texttt{)} \neq \texttt{nil} \vee b \neq \texttt{nil}$	Commute or	(*1)
$(a \neq \texttt{nil} \vee c \neq \texttt{nil}) \vee P$	Given	
$P \vee a \neq \texttt{nil} \vee c \neq \texttt{nil}$	Commute or	(*2)

$P \lor$ `(if a b c)` $\neq$ `nil`	Dj. aux split positive *1, *2	
`(if a b c)` $\neq$ `nil` $\lor P$	Commute or	□

Finally, the rules used in the proof of line 5 are quite simple to derive.

Derived Rule B-22. Aux split default 1

$$\frac{P \lor b \neq \texttt{nil} \lor Q \qquad a = b}{(a \neq \texttt{nil} \lor P) \lor Q}$$

Derivation. (59)

$P \lor b \neq \texttt{nil} \lor Q$	Given	
$(P \lor b \neq \texttt{nil}) \lor Q$	Associativity	
$Q \lor P \lor b \neq \texttt{nil}$	Commute or	
$(Q \lor P) \lor b \neq \texttt{nil}$	Associativity	(*1)
$a = b$	Given	
$(Q \lor P) \lor a = b$	Expansion	
$(Q \lor P) \lor a \neq \texttt{nil}$	Dj. sub. into $\neq$ *1	
$a \neq \texttt{nil} \lor Q \lor P$	Commute or	
$a \neq \texttt{nil} \lor P \lor Q$	Dj. commute or	
$(a \neq \texttt{nil} \lor P) \lor Q$	Associativity	□

Derived Rule B-23. Aux split default 2

$$\frac{P \lor b \neq \texttt{nil} \qquad a = b}{a \neq \texttt{nil} \lor P}$$

Derivation. (34)

$a = b$	Given	
$P \lor a = b$	Expansion	
$P \lor b \neq \texttt{nil}$	Given	
$P \lor a \neq \texttt{nil}$	Dj. sub. into $\neq$	
$a \neq \texttt{nil} \lor P$	Commute or	□

Appendix C

Main Lemma for the Fast Rewriter

As mentioned in Section 9.10, our most complicated proof is to show that our fast rewriter, FAST-CRW, produces the TRACE-IMAGE of our slow rewriter, CRW, when given the proper arguments. In this appendix we present the ACL2 `defthm` command for the main lemma relating the two flag functions.

To make this lemma more concise, we use ACL2's macro facility to introduce abbreviations for calls of FAST-CRW and CRW, which hide the numerous arguments which are unchanged. These macros may be identified by the use of the $ symbol in their names.

ACL2 Code

```
(defthm lemma-for-rw.trace-fast-image-of-rw.crewrite-core
  (implies
   (and (rw.assmsp assms)
        (rw.controlp control)
        (rw.cachep cache))
   (cond
    ((equal flag 'term)
     (implies
      (and (logic.termp x)
           (booleanp iffp))
      (and
       (equal (rw.cresult->alimitedp (rw.crewrite-core$ x))
              (rw.cresult->alimitedp
               (rw.fast-crewrite-core$ x
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))
       (equal (rw.cache-fast-image
               (rw.cresult->cache (rw.crewrite-core$ x)))
              (rw.cresult->cache
               (rw.fast-crewrite-core$ x
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))

       (equal (rw.trace-fast-image
               (rw.cresult->data (rw.crewrite-core$ x)))
              (rw.cresult->data
               (rw.fast-crewrite-core$ x
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache)))))))


    ((equal flag 'list)
     (implies
      (and (logic.term-listp x)
           (booleanp iffp))
      (and

       (equal (rw.cresult->alimitedp (rw.crewrite-core-list$ x))
              (rw.cresult->alimitedp
               (rw.fast-crewrite-core-list$ x
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))

       (equal (rw.cache-fast-image
               (rw.cresult->cache (rw.crewrite-core-list$ x)))
              (rw.cresult->cache
               (rw.fast-crewrite-core-list$ x
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))

       (equal (rw.trace-list-fast-image
               (rw.cresult->data (rw.crewrite-core-list$ x)))
              (rw.cresult->data
               (rw.fast-crewrite-core-list$ x
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache)))))))
```

```
((equal flag 'rule)
 (implies
  (and (logic.termp x)
       (booleanp iffp)
       (rw.rulep rule[s]))
  (and
   (equal (rw.cresult->alimitedp
           (rw.crewrite-try-rule$ x rule[s]))
          (rw.cresult->alimitedp
           (rw.fast-crewrite-try-rule$ x rule[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (equal (rw.cache-fast-image
           (rw.cresult->cache (rw.crewrite-try-rule$ x rule[s])))
          (rw.cresult->cache
           (rw.fast-crewrite-try-rule$ x rule[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (if (rw.cresult->data (rw.crewrite-try-rule$ x rule[s]))
       (equal (rw.trace-fast-image
               (rw.cresult->data
                (rw.crewrite-try-rule$ x rule[s])))
              (rw.cresult->data
               (rw.fast-crewrite-try-rule$ x rule[s]
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))
     t)
   (iff (rw.cresult->data (rw.crewrite-try-rule$ x rule[s]))
        (rw.cresult->data
         (rw.fast-crewrite-try-rule$ x rule[s]
          :assms (rw.assms-fast-image assms)
          :cache (rw.cache-fast-image cache)))))))


((equal flag 'rules)
 (implies
  (and (logic.termp x)
       (booleanp iffp)
       (rw.rule-listp rule[s]))
  (and
   (equal (rw.cresult->alimitedp
           (rw.crewrite-try-rules$ x rule[s]))
          (rw.cresult->alimitedp
           (rw.fast-crewrite-try-rules$ x rule[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (equal (rw.cache-fast-image
           (rw.cresult->cache
            (rw.crewrite-try-rules$ x rule[s])))
          (rw.cresult->cache
           (rw.fast-crewrite-try-rules$ x rule[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (if (rw.cresult->data (rw.crewrite-try-rules$ x rule[s]))
       (equal (rw.trace-fast-image
               (rw.cresult->data
                (rw.crewrite-try-rules$ x rule[s])))
              (rw.cresult->data
               (rw.fast-crewrite-try-rules$ x rule[s]
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))
     t)
   (iff (rw.cresult->data (rw.crewrite-try-rules$ x rule[s]))
        (rw.cresult->data
         (rw.fast-crewrite-try-rules$ x rule[s]
          :assms (rw.assms-fast-image assms)
          :cache (rw.cache-fast-image cache)))))))

((equal flag 'match)
 (implies
  (and (logic.termp x)
       (booleanp iffp)
       (rw.rulep rule[s])
       (logic.sigmap sigma[s]))
  (and
   (equal (rw.cresult->alimitedp
           (rw.crewrite-try-match$ x rule[s] sigma[s]))
          (rw.cresult->alimitedp
           (rw.fast-crewrite-try-match$ x rule[s] sigma[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (equal (rw.cache-fast-image
           (rw.cresult->cache
            (rw.crewrite-try-match$ x rule[s] sigma[s])))
          (rw.cresult->cache
           (rw.fast-crewrite-try-match$ x rule[s] sigma[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (if (rw.cresult->data
        (rw.crewrite-try-match$ x rule[s] sigma[s]))
       (equal
        (rw.trace-fast-image
         (rw.cresult->data
          (rw.crewrite-try-match$ x rule[s] sigma[s])))
        (rw.cresult->data
         (rw.fast-crewrite-try-match$ x rule[s] sigma[s]
          :assms (rw.assms-fast-image assms)
          :cache (rw.cache-fast-image cache))))
     t)
   (iff (rw.cresult->data
         (rw.crewrite-try-match$ x rule[s] sigma[s]))
        (rw.cresult->data
         (rw.fast-crewrite-try-match$ x rule[s] sigma[s]
          :assms (rw.assms-fast-image assms)
          :cache (rw.cache-fast-image cache))))))
```

```
((equal flag 'matches)
 (implies
  (and (logic.termp x)
       (booleanp iffp)
       (rw.rulep rule[s])
       (logic.sigma-listp sigma[s]))
  (and
   (equal (rw.cresult->alimitedp
            (rw.crewrite-try-matches$ x rule[s] sigma[s]))
           (rw.cresult->alimitedp
            (rw.fast-crewrite-try-matches$ x rule[s] sigma[s]
             :assms (rw.assms-fast-image assms)
             :cache (rw.cache-fast-image cache))))
   (equal (rw.cache-fast-image
            (rw.cresult->cache
             (rw.crewrite-try-matches$ x rule[s] sigma[s])))
           (rw.cresult->cache
            (rw.fast-crewrite-try-matches$ x rule[s] sigma[s]
             :assms (rw.assms-fast-image assms)
             :cache (rw.cache-fast-image cache))))
   (if (rw.cresult->data
        (rw.crewrite-try-matches$ x rule[s] sigma[s]))
       (equal (rw.trace-fast-image
               (rw.cresult->data
                (rw.crewrite-try-matches$ x rule[s] sigma[s])))
              (rw.cresult->data
               (rw.fast-crewrite-try-matches$ x rule[s] sigma[s]
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))
     t)
   (iff (rw.cresult->data
         (rw.crewrite-try-matches$ x rule[s] sigma[s]))
        (rw.cresult->data
         (rw.fast-crewrite-try-matches$ x rule[s] sigma[s]
          :assms (rw.assms-fast-image assms)
          :cache (rw.cache-fast-image cache)))))))
```

```
((equal flag 'hyp)
 (implies
  (and (rw.hypp x)
       (rw.rulep rule[s])
       (logic.sigmap sigma[s]))
  (and
   (equal (rw.cresult->alimitedp
           (rw.crewrite-relieve-hyp$ x rule[s] sigma[s]))
          (rw.cresult->alimitedp
           (rw.fast-crewrite-relieve-hyp$ x rule[s] sigma[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (equal (rw.cache-fast-image
           (rw.cresult->cache
            (rw.crewrite-relieve-hyp$ x rule[s] sigma[s])))
          (rw.cresult->cache
           (rw.fast-crewrite-relieve-hyp$ x rule[s] sigma[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))
   (if (rw.cresult->data
        (rw.crewrite-relieve-hyp$ x rule[s] sigma[s]))
       (equal (rw.trace-fast-image
               (rw.cresult->data
                (rw.crewrite-relieve-hyp$ x rule[s] sigma[s])))
              (rw.cresult->data
               (rw.fast-crewrite-relieve-hyp$ x rule[s] sigma[s]
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))
     t)
   (iff (rw.cresult->data
         (rw.crewrite-relieve-hyp$ x rule[s] sigma[s]))
        (rw.cresult->data
         (rw.fast-crewrite-relieve-hyp$ x rule[s] sigma[s]
          :assms (rw.assms-fast-image assms)
          :cache (rw.cache-fast-image cache)))))))


(t
 (implies
  (and (rw.hyp-listp x)
       (rw.rulep rule[s])
       (logic.sigmap sigma[s]))
  (and
   (equal (rw.hypresult->alimitedp
           (rw.crewrite-relieve-hyps$ x rule[s] sigma[s]))
          (rw.hypresult->alimitedp
           (rw.fast-crewrite-relieve-hyps$ x rule[s] sigma[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))

   (equal (rw.cache-fast-image
           (rw.hypresult->cache
            (rw.crewrite-relieve-hyps$ x rule[s] sigma[s])))
          (rw.hypresult->cache
           (rw.fast-crewrite-relieve-hyps$ x rule[s] sigma[s]
            :assms (rw.assms-fast-image assms)
            :cache (rw.cache-fast-image cache))))

   (if (rw.hypresult->successp
        (rw.crewrite-relieve-hyps$ x rule[s] sigma[s]))
       (equal (rw.trace-list-fast-image
               (rw.hypresult->traces
                (rw.crewrite-relieve-hyps$ x rule[s] sigma[s])))
              (rw.hypresult->traces
               (rw.fast-crewrite-relieve-hyps$ x rule[s] sigma[s]
                :assms (rw.assms-fast-image assms)
                :cache (rw.cache-fast-image cache))))
     t)

   (equal (rw.hypresult->successp
           (rw.crewrite-relieve-hyps$ x rule[s] sigma[s]))
          (rw.hypresult->successp
           (rw.fast-crewrite-relieve-hyps$ x rule[s] sigma[s]
:assms (rw.assms-fast-image assms)
:cache (rw.cache-fast-image cache))))))))))
```

Bibliography

[1] Algirdas A. Avižienis. The methodology of n–version programming. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 23–46. Wiley, 1995. 3

[2] Bruno Barras. Coq en Coq. Technical Report 3026, INRIA, October 1996. 466, 467

[3] Eli Barzilay. Quotation and reflection in Nuprl and Scheme. Technical Report 2001-1832, Cornell University, 2001. 20

[4] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics (TPHOLS '00)*, volume 1869 of *LNCS*, pages 38–52. Springer-Verlag, 2000. 462

[5] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Types for Proofs and Programs (Types '00)*, volume 2277 of *LNCS*, pages 24–40. Springer-Verlag, 2002. 469, 472

[6] Piergiorgio Bertoli and Paolo Traverso. Design verification of a safety-critical embedded verifier. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 14. Kluwer Academic Publishers, 2000. 20

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004. 460, 472

[8] William R. Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas at Austin, December 1987. 20

[9] Richard J. Boulton. Boyer-Moore automation for the HOL system. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOLS '92)*, volume A-20 of *IFIP Transactions*, pages 133–142. Elsevier Science Publisher, September 1992. 464

[10] Richard John Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge, December 1993. 292

[11] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1988. 468

[12] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, second edition, October 1997. 20, 161, 384

[13] R. S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof proceedures. In R. S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981. 467

[14] R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, 1996. 20

[15] Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991. 4

[16] Robert S. Boyer and Warren A. Hunt, Jr. Function memoization and unique object representation for ACL2 functions. In *ACL2 '06*, August 2006. 6, 326, 414, 420, 421, 473

[17] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995. 2

[18] Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979. 318, 380, 384, 386, 389

[19] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and its Applications, Essays in Honor of Larry Wos*. MIT Press, 1996. 20

[20] Bishop Brock, Matt Kaufmann, and J Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996. 20

[21] Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with equivalence relations in ACL2. *Journal of Automated Reasoning*, 40(4):293–306, May 2008. 4

[22] James L. Caldwell and John Cowles. Representing Nuprl proof objects in ACL2: Toward a proof checker for Nuprl. In Dominique Borrione, Matt Kaufmann, and J Moore, editors, *ACL2 '02*, April 2002. 467

[23] Amine Chaieb and Tobias Nipkow. Verifying and reflecting quantifier elimination for Presburger arithmetic. In *Logic Programming, Artificial Intelligence,*

and Reasoning (LPAR '05), volume 3835 of *LNCS*, pages 367–380. Springer-Verlag, 2005. 4, 469

[24] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *Principles of Programming Languages (POPL '77)*, pages 206–214. ACM Press, 1977. 2

[25] Solomon Feferman, editor. *Kurt Gödel: Collected Works*, volume 1. Oxford University Press, 1986. 521

[26] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988. 1

[27] Ruben A. Gamboa. The correctness of the Fast Fourier Transform: a structured proof in ACL2. *Formal Methods in System Design*, 20(1):91–106, January 2002. 333

[28] Kurt Gödel. Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I. *Monatshefte für mathematik und physik*, 38:173–198, 1931. English translation in [25], pages 145–195: On formally undecidable propositions of *Principia Mathematica* and related systems I. 465

[29] Wolfgang Goerigk. Compiler verification revisited. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 15. Kluwer Academic Publishers, 2000. 20

[30] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979. 361, 393, 461

[31] Michael J. C. Gordon. From LCF to HOL: A short history. In G. Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*. The MIT Press, 2000. 4

[32] Michael J. C. Gordon, James Reynolds, Warren A. Hunt, Jr., and Matt Kaufmann. An integration of HOL and ACL2. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 153–160, Nov 2006. 472

[33] Mike Gordon, Avra Cohn, Tom Melham, Konrad Slind, Michael Norrish, and et al. The HOL system: Description, September 2005. For HOL Kananaskis-3. 4, 361, 403

[34] Mike Gordon, Avra Cohn, Tom Melham, Konrad Slind, Michael Norrish, and et al. The HOL system: Tutorial, September 2005. For HOL Kananaskis-3. 459

[35] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS '05)*, volume 3603 of *LNCS*, pages 98–113. Springer Berlin, 2005. 469

[36] David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 8. Kluwer Academic Publishers, 2000. 20

[37] David A. Greve, Matt Kaufmann, Panagiotis Manoilos, J Strother Moore, Sandip Ray, José Ruiz-Reina, Rob Sumners, Daron Vroon, and Matthew Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(1), January 2008. 6

[38] David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In Jim Gundy and Malcom Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLS '98)*, volume 1479 of *LNCS*, pages 123–142. Springer-Verlag, September 1998. 462

[39] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. 4, 469

[40] John Harrison. HOL light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 265–269. Springer-Verlag, 1996. 4, 361, 459

[41] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNAI*, pages 177–191. Springer-Verlag, August 2006. 466

[42] Joe Hendrix. Matricies in ACL2. In *ACL2 '03*, July 2003. 333

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969. 1

[44] Douglas J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *Conference on Automated Deduction (CADE '88)*, LNCS, pages 238–257. Springer-Verlag, March 1988. 20

[45] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor.* LNCS. Springer, June 1994. 20

[46] Warren A. Hunt, Jr., Matt Kaufmann, Robert Bellarmine Krug, J Moore, and Eric Whitman Smith. Meta reasoning in ACL2. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS '05)*, volume 3603 of *LNCS*, pages 163–178. Springer Berlin, 2005. 4, 321, 322, 468

[47] Warren A. Hunt, Jr., Robert Bellarmine Krug, and J Moore. Linear and non-linear arithmetic in ACL2. In D. Geist, editor, *Correct Hardware Design and Verification Methods (CHARME '03)*, volume 2860 of *LNCS*, pages 319–333. Springer-Verlag, 2003. 4

[48] Warren A. Hunt, Jr. and Erik Reeber. Applications of the DE2 language. In Mary Sheeran and Tom Melham, editors, *Designing Correct Circuits (DCC '06)*. ETAPS '06, March 2006. 20

[49] Warren A. Hunt, Jr. and Sol Swords. Centaur technology media unit verification. Case study: Floating-point addition. In *Computer Aided Verification (CAV)*, volume 5643 of *LNCS*, pages 353–367. Springer-Verlag, June 2009. 20, 472

[50] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000. 2, 4, 19, 161, 471, 475

[51] Matt Kaufmann and J Moore. The ACL2 user's manual, 2009. Version 3.6.1. Available online: http://www.cs.utexas.edu/users/moore/acl2/v3-6/new/v3-6-1/HTML/acl2-doc-index.html. 460

[52] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997. 332, 459

[53] Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic, April 1998. 5, 19, 58, 471

[54] Matt Kaufmann and J Strother Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001. 471

[55] Matt Kaufmann, J Strother Moore, Sandip Ray, and Erik Reeber. Integrating external deduction tools with ACL2. In Christoph Benzmüller, Bernd Fischer, and Geoff Sutcliffe, editors, *6th International Workshop on the Implementation of Logics*, November 2006. 4, 468

[56] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Logic in Computer Science (LICS '86)*, pages 237–248. IEEE Computer Society, June 1986. 470

[57] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems (TOPLAS '99)*, 21(3):502–526, May 1999. 19

[58] Hanbing Liu and J Strother Moore. Executable JVM model for analytical reasoning: A study. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 15–23, 2003. 20

[59] Hanbing Liu and J Strother Moore. Java program verification via a JVM deep embedding in ACL2. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLS '04)*, pages 184–200, 2004. 20

[60] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust.* The MIT Press, October 2001. 3, 19

[61] Panagiotis Manolios and Matt Kaufmann. Adding a total order to ACL2. In *ACL2 '02*, April 2002. 269

[62] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, pages 1–37, 2006. 46

[63] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3(4):184–195, April 1960. 19, 218

[64] William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 16. Kluwer Academic Publishers, 2000. 466

[65] J Moore. Symbolic simulation: An ACL2 approach. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *LNCS*, pages 334–350. Springer-Verlag, November 1998. 20

[66] J Strother Moore and Qiang Zhang. Proof pearl: Dijkstra's shortest path algorithm verified with ACL2. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS '05)*, volume 3603 of *LNCS*, pages 373–384. Springer Berlin, 2005. 20

[67] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 4, 460

[68] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNAI*, pages 298–302. Springer-Verlag, August 2006. 462, 466

[69] Russell O'Connor. Essential incompleteness of arithmetic verified by coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS '05)*, volume 3603 of *LNCS*, pages 245–260. Springer Berlin, 2005. 465

[70] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992. 4, 462

[71] S. Owre, J. M. Rushby, N. Shankar, and D. W. J. Stringer-Calvert. PVS: An experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods–FM-Trends 98*, volume 1641 of *LNCS*, pages 338–345. Springer-Verlag, October 1998. 460

[72] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, July 1990. 403

[73] David L. Rager and Warren A. Hunt, Jr. Implementing a parallelism library for a functional subset of LISP. In *International Lisp Conference (ILC)*, pages 18–30, March 2009. 6, 420, 474

[74] Sandip Ray, John Matthews, and Mark Tuttle. Certifying compositional model checking algorithms in ACL2. In *ACL2 '03*, July 2003. 20

[75] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS '05)*, volume 3603 of *LNCS*, pages 294–309. Springer Berlin, 2005. 465

[76] J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo, and F.-J. Martín-Mateos. A formally verified quadratic unification algorithm. In Matt Kaufmann and J Moore, editors, *ACL2 '04*, November 2004. 20

[77] David M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:147–200, December 1998. 20

[78] David M. Russinoff and Arthur Flatau. Mechanical verification of register-transfer logic: A floating-point multiplier. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 13. Kluwer Academic Publishers, 2000. 472

[79] David M. Russinoff and Arthur Flatau. RTL verification: A floating-point multiplier. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 13. Kluwer Academic Publishers, 2000. 20

[80] Jun Sawada. Verification of a simple pipelined machine model. In *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 9. Kluwer Academic Publishers, 2000. 20

[81] Peter Seibel. *Practical Common Lisp*. Apress, April 2005. 103

[82] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994. 161, 177, 186, 190, 465

[83] Joseph R. Shoenfield. *Mathematical Logic*. The Association for Symbolic Logic, 1967. 19, 161, 186, 190, 471

[84] Konrad Slind. Adding new rules to an LCF-style logic implementation: Preliminary report. In L. J. M. Claesan and M. J. C. Gordon, editors, *Higher*

Order Logic Theorem Proving and its Applications (TPHOLS '92), volume A-20 of *IFIP Transactions*. Elsevier Science Publisher, September 1992. 469

[85] Eric Smith, Serita Nelesen, David Greve, Matthew Wilding, and Raymond Richards. An ACL2 library for bags. In Matt Kaufmann and J Moore, editors, *ACL2 '04*, November 2004. 468

[86] Guy L. Steele. *Common LISP: The Language*. Digital press, second edition, June 1990. 103

[87] Sol Swords and William R. Cook. Soundness of the simply typed lambda calculus in ACL2. In Panagiotis Manolios and Matthew Wilding, editors, *ACL2 '06*, August 2006. 333

[88] Cornell University The PRL Group. Implementing mathematics with the Nuprl proof development system, October 1995. 460

[89] Diana Toma and Dominique Borrione. SHA formalization. In *ACL2 '03*, July 2003. 20

[90] J. von Wright. The formal verification of a proof checker, 1994. SRI Internal Report. 465

[91] J. von Wright. Representing higher-order logic proofs in HOL. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications (TPHOLS '94)*, volume 859 of *LNCS*. Springer-Verlag, September 1994. 465

[92] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3):233–248, May 2001. 472

[93] Wai Wong. Recording and checking HOL proofs. In E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOLS '95)*, volume 971 of *LNCS*, pages 353–368. Springer-Verlag, September 1995. 462

[94] William D. Young. *A Verified Code-Generator for a Subset of Gypsy.* PhD thesis, University of Texas at Austin, December 1988. 20

Index

Vita

Jared Curran Davis graduated from Westside High School, Omaha, Nebraska. In 1999 he entered the University of Nebraska at Omaha, and later graduated with a Bachelor of Science. He entered the Graduate School at the University of Texas at Austin in 2003. He has worked in software and hardware verification at Sandia National Laboratories, Rockwell Collins, Inc., and Centaur Technology.

Permanent address: 11410 Windermere Meadows
Austin, Texas 78759-4551

This dissertation was typeset by the author.