WASHINGTON UNIVERSITY

SEVER INSTITUTE

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BAGAHK: DEVELOPING SOUND AND COMPLETE DECISION PROCEDURES IN COQ

by

Benjamin J. Delaware, B.S, B.A.

Prepared under the direction of Professor Aaron Stump

A thesis presented to the Sever Institute of
Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

August 2007

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

BAGAHK: DEVELOPING SOUND AND COMPLETE DECISION PROCEDURES IN COQ

by

Benjamin J. Delaware

ADVISOR: Professor Aaron Stump

August 2007

Saint Louis, Missouri

Decision procedures are automated theorem proving algorithms which automatically recognize the theorems of some decidable theory. The correctness of these algorithms is important, since a design error could lead to the misidentification of a false statement as a theorem. In the past, decision procedures have been shown to be correct by mechanically verifying that they are sound, i.e. they only identify valid statements. Soundness does not entail correctness, however, as a decision procedure could still fail to recognize a true formula from the theory it decides. To rigorously verify that a decision procedure for a theory $\mathcal{T}$ is correct, it must also be shown to be complete in that it recognize all true propositions from $\mathcal{T}$.

We have developed a decision procedure called BAGAHK for the validity of formulas modulo the theory of ground equations $\mathcal{T}_=$, which we have proven sound and complete in the proof assistant Coq. In this thesis, we highlight the important lemmas and theorems of these proofs. As part of the soundness proof, we embed Coq-level proof terms into the meta-language of our solver using reflection. As a result of this, BAGAHK can also be used to assist users in the construction of other proofs. In addition, we develop a proof system for $\mathcal{T}_=$ and show that our decision procedure recognizes all $\mathcal{T}_=$-provable propositions, showing that BAGAHK is complete.

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank my advisor, Aaron Stump, for introducing me to the world of automated theorem proving in the fall of 2005. His help and guidance over the past year has been invaluable in the creation of this thesis, and in my growth as a researcher. Further thanks is due to Sally Goldman and Robert Pless for agreeing to serve on my committee.

I also appreciate my friends and family for continuing to ask questions being told that my research is technical, then smiling and nodding as I answer their questions. Thanks also goes to the Compuational Logic group for their help in the preparation of my defense.

Finally, many thanks are due to my wife Shannon, for waking up at 3 in the morning to find me hunched in front of my laptop, working on a proof that I said I would finish hours ago, and still supporting me. You remain an inspiration for all that I do.

Benjamin J. Delaware

Washington University in Saint Louis
August 2007

Preface

Throughout this thesis, Coq terms are set in typewriter font outside of figures: `check_valida, extern_dp` .... The lemmas and theorems presented here are labeled with the titles of their corresponding Coq proofs; when they are referenced within the text, they will appear in small capitals: CHECK_VALIDA_SOUND. The Coq definitions of the major theorems appear as figures in the chapters in which they appear; definitions of all lemmas and theorems can be found in Appendix A.

BAGAHK was developed and verified using version 8.1beta of the Coq Proof Assistant; the complete definitions and proof scripts of BAGAHK should be available at `http://cl.cse.wustl.edu/`.

Chapter 1

Introduction

Proof is the idol before whom the pure mathematician tortures himself.

Sir Arthur Eddington, *The Nature of the Physical World*

While there are certainly many students of calculus who would agree with Sir Arthur Eddington, proofs are the foundation of mathematics, stretching back to the syllogisms of Aristotle. A *proof* is simply an argument that a statement logically follows from a set of propositions assumed to be true, called *axioms*. Once a proof has shown a statement to be true, it is called a *theorem*. The proofs most people are familiar with are so-called 'social' proofs, natural language arguments that are rigorous enough to convince others that a statement is correct. A convincing argument for one person might not be persuasive to another, however, making it difficult to establish a standard for 'correct' social proofs. An alternate approach is to build a proof as a sequence of statements that follow one another based upon a well-defined set of *proof rules*. These are known as 'formal' proofs, and are the focus the field of proof theory. Showing that such a proof is correct is a matter of checking that the sequence obeys the set of proof rules. While their strict syntax gives formal proofs a mathematical rigor, it also makes them tedious to construct and difficult to comprehend. Social proofs, on the other hand, are able to gloss over some of the details of the formal proofs, instead focusing on the points that are important for understanding the intuition behind the argument. In theory, the two approaches are equivalent- a social proof should imply the existence of a formal one and vice-versa, explaining why the more approachable social proofs are prevalent in practice.

Broadly speaking, there are two methods for building computer-based proofs: automatically and interactively. The first is the province of *automated theorem proving*; this field attempts to find proofs algorithmically. Ideally, it would be possible to design a general-purpose algorithm which could determine whether any statement is true or false. Unfortunately, there are classes of problems for which no terminating algorithm exists. This set of *undecidable* problems includes integer arithmetic with addition and multiplication [23], the solvability of a Diophantine equation, and the halting problem from computability theory. Algorithms which recognize true statements can be designed for this class of problems, but their use is limited: Since they do not halt on all inputs, it is impossible to tell if a long-running computation is a sign that a statement is not a theorem or if the algorithm needs to run for more time. Denied an effective general-purpose solver, we instead turn to the creation of algorithms for decidable domains. These algorithms, called *decision procedures*, are designed to recognize the true statements of a particular theory and are guaranteed to halt on both true and false statements. Through the combination of decision procedures for a variety of theories, a robust theorem prover that can handle a wide range of problems can be built.

In the second approach, known as *interactive theorem proving*, a human user guides the construction of a proof using a software *proof assistant*. There are a number of mature proof assistants available today, including Coq [4] and Isabelle/HOL [20]. By allowing for the use of human intuition, this method can handle a greater range of problems than automated techniques, potentially proving any mathematical theorem. In its simplest form, the user directly manipulates the proof terms to show the desired goal. At this level, the software serves as a *proof checker* which verifies that the steps of the proof follow the set of proof rules. As a computerized version of formal proofs, this strategy shares their weaknesses: manually manipulating proof terms quickly grows tedious and produces proofs that are largely unreadable to others. Instead, most proof assistants help the user construct pieces of the proof automatically through the use of *tactics*, automated reasoning scripts which manipulate the underlying proof terms. Tactics lead to more intuitive proofs: since the user is able to use tactics corresponding to familiar approaches, they can construct the proof more easily; these proofs are in turn easier to read and maintain. Furthermore, tactics can incorporate techniques from automated theorem proving to easily construct large portions of proofs automatically. Through these tactics, proof assistants are able to offer the best of both worlds: the user is able to direct the proof using their experience and intuition while bringing to bear the power of automated reasoning to dispatch many proof obligations.

The subject of this thesis is the design, implementation, and verification of such a tactic for the Coq proof assistant. Our tactic recognizes the theorems of the theory of propositional validity modulo the theory of ground equations. *Ground equations* are equations between two constants, such as $a = b$. The formulas handled by our reasoner contain a mix of boolean values, propositional variables, and ground equations, joined together by the standard boolean connectives: $\rightarrow, \wedge, \vee, \neg$. $(p \vee q) \rightarrow (a = c)$ and $(p \rightarrow (a = b)) \vee (\neg(b = a) \wedge \mathbf{T})$ are examples of such formulas. The first formula is not valid in this theory: with no a priori knowledge of the truth values of the variables, p or q could be true and $a = b$ could be false, falsifying the formula. The second formula, on the other hand, is valid. If p and $a = b$ are false and $b = a$ is true, the formula is false; however, this assignment is contradictory, since the symmetry of equality states that $a = b$ and $b = a$ have the same truth value. Thus we can say this formula is valid *modulo* the theory of ground equations.

Program correctness is nothing new to computer scientists; while the advent of the transistor eliminated the threat of insects to programs, software bugs continue to plague programmers today. *Program verification* is concerned with demonstrating the correctness of programs- that they perform according to some specification. The concept of mathematically verifying the correctness of a program dates back to the axiomatic approach of Hoare [15], and work continues today on providing an effective and practical means of program verification. The question of correctness is of particular interest in the design of algorithms in automated theorem proving. Clearly, decision procedures should only affirm that true statements are theorems, and never report that a false statement is true. Such a decision procedure is said to be *sound*. Traditionally, a decision procedure which has been proven sound is said to be verified; such algorithms exist for domains such as tautology checking [24], membership in polynomial ideals [26], and the combination of canonizable and solvable theories [12]. This definition of correctness is unsatisfying, however: a decision procedure which declares every statement to be false is certainly sound, since it finds no proposition, true or false, to be a theorem, but such an algorithm can hardly be said to be correct.

In addition to being sound, we also want a *complete* decision procedure which finds all theorems of the theory it decides. A bulk of the discussion in this thesis is devoted to showing that the decision procedure we have developed is both sound and complete.

1.1 Contributions

1. A major contribution of this project is the creation of the BAGAHK artifact itself. While the decision procedure proper is only 200 lines of code, the proofs of its soundness and completeness are over 2500 lines. Since these formal proofs have passed through Coq's proof checker, the correctness of the algorithm has been proven with precise mathematical rigor[1].

2. By formulating our decision procedure as a Coq tactic, we have provided a new tool to assist in user-directed proofs. While there are other tactics that handle propositional tautology checking, none handle validity modulo the theory of ground equations. Furthermore, the modular nature of our implementation of the decision procedure allows for the inclusion of additional theories in BAGAHK.

3. Finally, while other verified decision procedures are confined to being shown sound, our solver is both sound and complete- it will affirm that a statement is a theorem if and only if a corresponding Coq proof exists. The basic methodology of our proofs lends itself well to extension; with many of the major design hurdles cleared, we are confident that these proofs can be extended to any additional theories handled by the solver.

[1]It should be noted that the proofs in this thesis are decidedly social. Rather than bombard the reader with the painstaking level of detail in the Coq proof scripts, we provide instead the major lemmas and theorem of these proofs in enough detail to persuade the reader of their correctness.

Chapter 2

Background

This section overviews some of the background material necessary for the rest of this thesis. The first section covers the translation from Coq-level problems to problems handled by our decision procedure. The second section covers the theory handled by the reasoner and an approach to the larger class of problems to which it belongs. The third discusses a definition of soundness and completeness from proof theory that agrees with the reasoning used by our decision procedure.

2.1 Proof by Computation through Reflection

The mathematical foundation for Coq is the Calculus of Inductive Constructions [7], a typed λ-calculus [1]. Proofs in this system are *constructive*: they are thought of as functions from a set of assumptions to the conclusion. A proof of $P \to Q$ in this system, for instance, is a function which builds a proof of Q from a proof of P. Under this interpretation of proofs, there is a direct correspondence between the λ-calculus of Church and natural deduction proofs, known as the Curry-Howard Isomorphism [8] [16]. In this correspondence, statements and functions in a functional language represent proofs and their type is the statement they prove. A sorting function which takes in a list and returns a sorted list, for instance, can be thought of as a proof that any list can be sorted. The Calculus of Inductive Constructions provides a theory with enough power to express and prove several interesting problems, including the four color theorem [13], Gödel's Incompleteness Theorem [21], and Fermat's Last Theorem for $n = 4$ [10]. As a consequence of the equivalence of proofs and programs, Coq allows users to design programs and then reason about their properties in addition to proving mathematical theorems. Since users can certify that programs meet their specifications, Coq also serves as a tool for program verification. Furthermore, after writing a function and proving it correct, Coq has facilities for extracting the verified code to the OCaml language for use in other programs.

A further result of the inclusion of a programming language in Coq is that proofs can be constructed computationally [22]. Consider a proof of $0+(x+y) = (0+x)+y$: this clearly holds by the transitivity of addition, but since plus is a computable function, it is also possible to run a few steps of computation to reduce both sides to $x + y$, at which point the proof is immediate from the reflexivity of equality. Alternatively, suppose that there is a function that checks whether a natural number is prime, `is_prime : nat` $\rightarrow$ `bool`. If we are trying to show that $\forall xyz, y \neq x \rightarrow z \neq x \rightarrow x <> y * z$, it suffices to show that `is_prime` x `= true`. Of course, we will have to prove $\forall xyz$, `is_prime` x `= true` $\rightarrow x <> y * zy \neq x \rightarrow z \neq x$. Once we have proven this, though, any proof of a specific instance of this predicate, such as $23 <> 4 * z$, is reduced to a computational task, namely checking that 23 is prime. Computation is included in Coq's notion of definitional equality, making is_prime 23 definitionally equal to true, so no proof of that fact is needed.

Reducing proofs to computationally decidable tasks is precisely the goal of automated theorem proving. Because `Prop`, the type of the formulas which are the type of proofs in Coq, is not an inductive product, it is impractical for use in a case-based reasoning procedure. Instead, the proof terms can be embedded into a inductive data type which can then be used in a decision procedure, a method known as *reflection* [14]. Also called the *two-level* approach, in this technique the object language is 'reflected' into a meta language; theorems in the meta language then correspond to theorems in the object language. There are five pieces necessary to use reflection to create a decision procedure for a general class of problems:

1. There must be an encoding of the problems as an inductive data type, in this case the type `form`.
2. There is an encoding function, $\lceil - \rceil$: `Prop` $\rightarrow$ `form` from the object language to the meta language.

3. There is a decoding function, $\lfloor - \rfloor$: `form` $\rightarrow$ `Prop` from the meta language to the object language.

4. A decision procedure, `dp : form` $\rightarrow$ `bool` for the class of problems.

5. A proof that a true statement in the meta language corresponds to a theorem in the object language, i.e. $\forall$ `(`ψ `: Prop), dp` $\lceil\psi\rceil$ `= true` $\rightarrow$ $\lfloor\lceil\psi\rceil\rfloor$ (Note that if the decoding function is the inverse of encoding function, $\lfloor\lceil\psi\rceil\rfloor = \psi$).

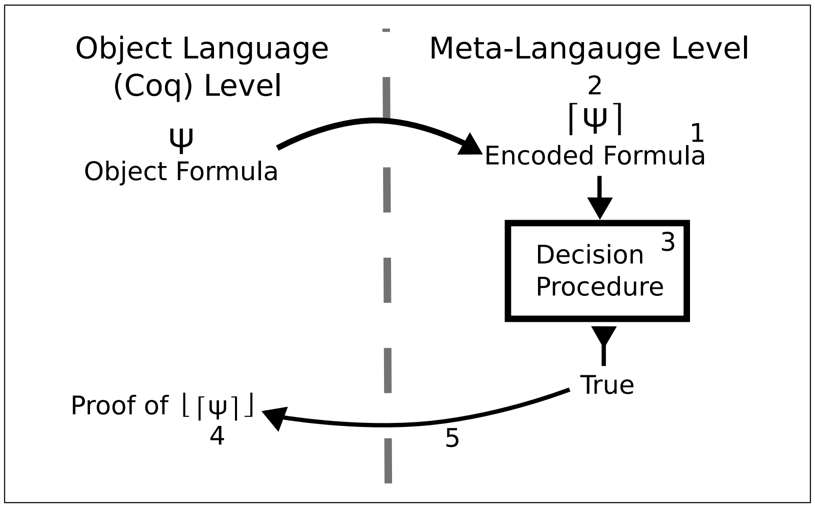

Figure 2.1: A Graphical Representation of the Reflection Method.

Using this paradigm, a proposition ψ can be shown by first embedding it into the data type, showing that `dp` $\lceil\psi\rceil$ returns `true`, and then applying the proof from 5. Once these 5 pieces are in hand, the key piece of any proof is computing the `dp` function, a purely mechanical task which is easily dispatched. Since the decision procedure described in (4) is generally designed to handle a class of problems, reflection is commonly used in the design of tactics.

2.2 Satisfiability Modulo Theories

An alternative approach to proving a propositional formula,ψ, is to assign each variable an *interpretation*, either **T** or **F**. We say that an interpretation is a *model* of a formula if the formula is true under that interpretation. If all possible assignments are models of ψ, it follows that whatever the interpretation, ψ will be true, and we say that it is (classically) *valid.* The formula $(P \wedge Q) \rightarrow (P \rightarrow Q)$, for example, can be proven by constructing the truth table in Fig. 2.2.

P	Q	$P \wedge Q$	$P \rightarrow Q$	$(P \wedge Q) \rightarrow (P \rightarrow Q)$
T	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**
F	**T**	**F**	**T**	**T**
F	**F**	**F**	**T**	**T**

Figure 2.2: Truth table for $(P \wedge Q) \rightarrow (P \rightarrow Q)$.

This proof technique is well-suited for computation, since the validity of a formula can be established through rote enumeration of all possible interpretations. This is the dual of the satisfiability problem (SAT) which asks if there is *some* assignment which makes the formula true. As the quintessential NP-complete problem, designing efficient SAT solvers remains an active and important research area [17]; any advances in this field translate immediately to validity checking.

The Satisfiability Modulo Theorem (SMT) problem is an extension of SAT which determines the satisfiability of a formula with respect to a theory. In the SMT problem, the standard propositional formulas of SAT are extended to include atoms from some other theory, $\mathcal{T}$, such as the theory of linear arithmetic or Equality with Uninterpreted Functions. In the former case, in addition to propositional atoms, the formula could have atoms such as $x + y \leq 2 \cdot z$. The variables $x, y, z \in \mathbb{N}$ have an infinite set of interpretations, but the only interpretations for $x + y \leq 2 \cdot z$ are **T** and **F**. The truth value of this predicate depends entirely on the interpretations of x, y, and z. From a traditional SAT viewpoint, the formula $(5 < x) \rightarrow (A \wedge B) \rightarrow (A \rightarrow 2 \cdot x \leq y) \rightarrow (B \rightarrow x + 5 \leq y)$ is not valid: there is a falsifying assignment $(5 < x) \leftrightarrow \mathbf{T}, A \leftrightarrow \mathbf{T}, B \leftrightarrow \mathbf{T}, 2 \cdot x \leq y \leftrightarrow \mathbf{T}, x + 5 \leq y \leftrightarrow \mathbf{F}$. This formula is valid modulo the theory of linear algebra however, as this assignment is inconsistent with that theory: $x < 5 \rightarrow 2 \cdot x \leq y \rightarrow x + 5 \leq y$. The increased expressive power of these formulas has lead to applications in a number of domains, particularly in hardware and software verification. Events such as the recent SMT Competition [25] continue to encourage improvements among modern SMT solvers, including CVC3 [3], Barcelogic [19], MathSAT [5], Yices [11].

There are two approaches generally used by these solvers [2], an *eager* approach, which translates in the SMT into an instance for a standard SAT solver in a manner which preserves satisfiability, and a *lazy* approach, which treats the formula as a standard SAT instance. Once the solver finds a model of the formula, it uses a separate decision procedure to check that the model is consistent with the theory $\mathcal{T}$. If there is an assignment which is a propositional model of the formula and which is consistent with $\mathcal{T}$, the formula is satisfiable. The eager approach is the more direct of the two, since the converted formula can be plugged into any SAT-solver. The lazy approach is the more modular of the two, allowing for the easy integration of new decision procedures for additional theories. Furthermore, the lazy approach retains high-level information which is lost in the encoding for the eager approach; information which can be used to intelligently guide the search. We'll consider the lazy approach for the dual problem of testing the validity of the formula $(5 < x) \rightarrow (A \wedge B) \rightarrow (A \rightarrow 2 \cdot x \leq y) \rightarrow (B \rightarrow x + 5 \leq y)$. The validity checker will proceed by splitting on the truth values for both the propositional variables and the inequalities. When a propositional counter-model such as $(5 < x) \leftrightarrow \mathbf{T}, A \leftrightarrow \mathbf{T}, B \leftrightarrow \mathbf{T}, 2 \cdot x \leq y \leftrightarrow \mathbf{T}, x + 5 \leq y \leftrightarrow \mathbf{F}$ is found, the decision procedure for $\mathcal{T}$ will determine that the assignment $(5 < x) \leftrightarrow \mathbf{T}, 2 \cdot x \leq y \leftrightarrow \mathbf{T}, x + 5 \leq y \leftrightarrow \mathbf{F}$ is not consistent with the theory of linear arithmetic. This result is passed to the satisfiability checker; since all other propositional truth assignments are models, it concludes that the formula is valid.

The theory decided by our algorithm is the theory of ground equations, $\mathcal{T}_=$. The propositions of $\mathcal{T}_=$ are equations between uninterpreted symbols which are treated as individual objects. The standard axioms of equality provide the basis for $\mathcal{T}_=$: equality is reflexive, $x = x$, symmetric, $x = y \rightarrow y = x$, and transitive, $x = y \rightarrow y = z \rightarrow x = z$. A truth assignment to a set of propositions is inconsistent with this theory if the set contains equations between the same terms with different truth values or if it is possible to build such a pair using the axioms of $\mathcal{T}_=$. The propositions $x = y, y = z, z \neq x$, for instance, are inconsistent: from the first two equations, the transitivity axiom derives that $x = z$; from this $z = x$ by symmetry, producing a set which contains both $z = x$ and $z \neq x$, a contradiction. The consistency of a set of propositions in $\mathcal{T}_=$ can be checked algorithmically by rewriting the set of *disequations* (negated equations) using the set of equations. After this rewriting, if any of the disequations are between two of the same term, the set is contradictory. Intuitively, the set of equations induces a set of equivalence classes which can be obtained by rewriting all members of a class to a common term. If any disequations are between members of the same equivalence class, the assignment is inconsistent with $\mathcal{T}_=$.

2.3 Soundness and Completeness

In proof theory, a *proof system* is a set of axioms and proof rules that act on well-defined formulas. In a proof system $\mathcal{P}$, a formula is said to be $\mathcal{P}$-provable if it is the conclusion of a sequence of formulas, each of which is either an axiom or a consequence of an application of a proof rule to the preceding formula. Such a sequence is called a *deduction*, and we write $\vdash_{\mathcal{P}} \psi$ to denote that formula ψ has a deduction in $\mathcal{P}$. Similarly, if a formula ψ is valid in $\mathcal{P}$, we write $\models_{\mathcal{P}} \psi$. If every $\mathcal{P}$-provable formula is valid, we say that $\mathcal{P}$ is sound: $\vdash_{\mathcal{P}} \psi \rightarrow \models_{\mathcal{P}} \psi$. Alternatively, if there is a proof in $\mathcal{P}$ for every valid formula, $\mathcal{P}$ is complete: $\models_{\mathcal{P}} \psi \rightarrow \vdash_{\mathcal{P}} \psi$.

In the literature on verified tactics, the definition of soundness is that the decision procedure will only recognize true statements: $\forall$ `(`ψ `: Prop), dp` $\lceil\psi\rceil$ `= true` $\rightarrow$ $\lfloor\lceil P\rceil\rfloor$. In Coq, this corresponds to building a Coq-level proof from an affirmative result of the decision procedure. Since this is precisely the final piece of the reflection method, the soundness of a reflection-based tactic is immediate. Normally, the completeness proof would show the inverse, that if the decision procedure says that a formula is not a theorem, no proof of that statement exists: $\forall$ `(`ψ `: Prop), dp` $\lceil\psi\rceil$ `= false` $\rightarrow$ $\neg\exists\lfloor\lceil\psi\rceil\rfloor$. The problem with this approach is clear: a statement could be proved using a theory other than $\mathcal{T}_=$. Our reasoner would not be able to prove $2 + 2 = 4$, for example, but a Coq proof is easily constructed using basic arithmetic. Instead, we will tackle the converse, showing our decision procedure will recognize all true statements: $\forall$ `(`ψ `: Prop),` $\psi \rightarrow$ `dp` $\lceil\psi\rceil$ `= true`. This approach allows us to constrain ψ to the set of propositions provable in $\mathcal{T}_=$- the theorems that our reasoner recognizes.

Chapter 3

Reflection and Computation

The remainder of this thesis describes BAGAHK, an implementation of a decision procedure for satisfiability modulo $\mathcal{T}_=$ and the proofs that it is sound and complete. This chapter discusses the details of the first four steps of the reflection method:

1. We first discuss the `form` recursive data type used to embed Coq propositions.
2. The encoding function is a straightforward implementation in the Ltac scripting language.
3. Next, we discuss our choice to implement the decoding function as the `encodes` relation which relates a meta-language formula ϕ to the Coq formula ψ it embeds.
4. Finally, we discuss the major pieces of the `check_vailda` decision procedure which is written in the programming language included as part of Coq's logic. In particular, we discuss the `subst_var_asgn` function which checks the whether a formula is satisfied by a given assignment and the `extern_dp` function which checks if a truth assignment is consistent with $\mathcal{T}_=$.

Because we have implemented `check_vailda` in Coq's programming language, we are able to reason about its correctness, showing that it is sound and complete. Chapter 4 describes the soundness proof, which shows that `check_vailda` recognizes a formula as a theorem, there is a proof of the Coq formula that it embeds. This proof allows us to translates the computational results from `check_vaild` back to the Coq level, and is the fifth and final step of the reflection method. Finally, Chapter 5 shows that our decisions procedure is truly correct by detailing the proof that it is complete- that `check_vaild` will recognize all the theorems of $\mathcal{T}_=$.

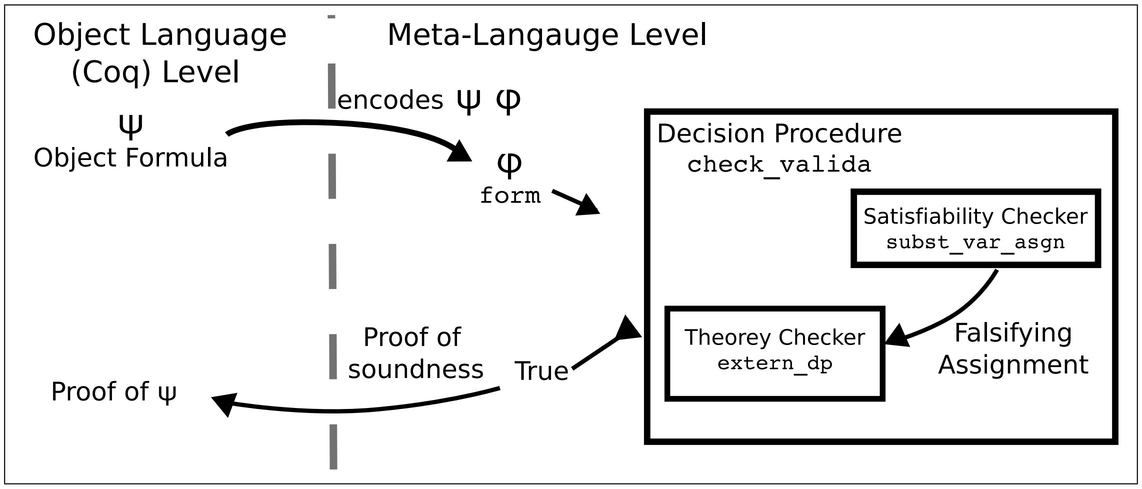

Figure 3.1: Our implementation of the Reflection Method.

3.1 Embedding the Coq Terms

The first step of the reflection method is the construction of the inductive data type of the meta-language into which the Coq formulas will be embedded. Each type of atomic proposition handled by the reasoner has a base constructor; our representation uses one for True and False, one for equations, and a generic atomic constructor for propositional variables. The latter two atoms are indexed by a natural number; the equational atom also has natural number indices for the two sides of the equation. Combining these with inductive constructors for the standard propositional connectives provides the framework for our meta-language representation of Coq terms, shown in Fig. 3.2. While this structure is sufficient for use by the reasoner, it is important to consider what information is needed to lift the embedded formula back to the Coq level. We looked at two methods for accomplishing this: a lifting function, `lift : form` $\rightarrow$ `Prop`, which directly constructs the Coq term from a given embedded formula and an encoding relation, `lift : form` $\rightarrow$ `Prop` $\rightarrow$ `Prop`, which associates an embedded formula to a Coq term.

```
Inductive form : Type :=
Atom : nat → form
| BoolF : bool → form
| Eqn : nat → nat → nat → form
| AndF : form → form → form
| OrF : form → form → form
| NotF : form → form
| ImpF : form → form → form.
```

Figure 3.2: The definition for embedded formulas.

The initial iteration of BAGAHK was a simple tautology checker without equational reasoning, and we used the first approach. In this case there were only the generic atoms, so the lifting function simply replaced each atom with a universally quantified Coq proposition and each inductive constructor with the corresponding propositional connective. Introducing equational reasoning complicates this lifting function: since equality is defined on terms of the same type, extra type information must be added to the embedded equations. Consider lifting an equation built using the constructor in 3.2: `Eqn 0 1 2`. The lifting function needs to map these indices to their corresponding Coq-level terms and then build a proposition that these are equal, but Coq needs some proof that they are of the same type. There are two options for dealing with this: using an alternate definition of equality that allows for terms of different types, or including extra information in the data type of the embedded formula. The first method significantly complicates the soundness proof as it unnecessarily discards information about the formula.

The second method requires adding an function, τ, which maps the term indices to their types; we can then use this function to create a dependently-typed constructor for equations. In order to insure that lifted terms match the type mapped to by τ, it is also necessary to include the Coq-level term in the type; adding a new constructor for each term whose type is dependent on the type of the embedded term. Finally, once this is added to the equation constructor we need to include it in all of the constructors for consistency, resulting in the type definition in 3.3. This additional information is not needed for the reasoning code, but maintaining the type information adds complexity to the code, particularly in the equational reasoner. More importantly, proofs about code working on this updated embedding requires sophisticated reasoning about dependent types, complicating our soundness and completeness proofs. Regardless of which method is used, this a frustrating complication: we know that our lifting is valid since the embedded formula was derived from a well-typed term, but we are unable to use this because the lifting function constructs a new formula.

```
Inductive term (τ: nat → Type) : Type → Type :=
  grnd : forall (i : nat), τ i → term τ (τ i).

Inductive form (τ : nat → Type) : Type :=
Atom : nat → form τ
| BoolF : bool τ → form τ
| Eqn : ∀ i : nat, term τ i → term τ i → form τ
| AndF : form τ → form τ → form τ
| OrF : form τ → form τ → form τ
| NotF : form τ → form τ
| ImpF : form τ → form τ → form τ.
```

Figure 3.3: The definition for embedded formulas, modified for the lifting function.

By using a relation, we are able to directly associate the embedded formula to the Coq term that generated it, allowing us to use the fact that this term is well-typed. Furthermore, this allows us to avoid including extraneous type information in embedded formulas, which in turn simplifies the reasoning algorithms. All the necessary information is contained in the encodes relation in Fig. 3.4, completely separating the embedded formula from the Coq term it represents. Generic and equational atoms are mapped to their corresponding Coq terms using the F and G functions; indexing the encodes relation with these functions insures consistency throughout. The propositional connectives are related to their embedded counterpart as expected. When reasoning about this structure, we are able to induct on the structure of the embedded formula. With the proposition directly related to a proposition, we can use inversion to expose the underlying structure of encoded proposition.

Inductive encodes (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup)
: form $\rightarrow$ Prop $\rightarrow$ Prop :=
encodes_atom : $\forall$ (i : nat) (ϕ : Prop),
  G i = prop_atom ϕ $\rightarrow$ encodes F G (Atom i) ϕ
| encodes_true : encodes F G (BoolF true) (True)
| encodes_false : encodes F G (BoolF false) (False)
| encodes_not : $\forall$ (ϕ : form) (ψ : Prop),
  encodes F G ϕ ψ $\rightarrow$ encodes F G (NotF ϕ) ($\neg\psi$)
| encodes_and : $\forall$ (ϕ_1 ϕ_2 : form) (ψ_1 ψ_2 : Prop), (encodes F G ϕ_1 ψ_1) $\rightarrow$
  (encodes F G ϕ_2 ψ_2) $\rightarrow$ encodes F G (AndF ϕ_1 ϕ_2) ($\psi 1 \wedge \psi_2$)
| encodes_or : $\forall$ (ϕ_1 ϕ_2 : form) (ψ_1 ψ_2 : Prop), (encodes F G ϕ_1 ψ_1) $\rightarrow$
  (encodes F G ϕ_2 ψ_2) $\rightarrow$ encodes F G (OrF ϕ_1 ϕ_2) ($\psi 1 \vee \psi 2$)
| encodes_imp : $\forall$ (ϕ_1 ϕ_2 : form) (ψ_1 ψ_2 : Prop), (encodes F G ϕ_1 ψ_2) $\rightarrow$
  (encodes F G ϕ_2 ψ_2) $\rightarrow$ encodes F G (ImpF ϕ_1 ϕ_2) ($\psi_1 \rightarrow \psi_2$)
| encodes_eqn : $\forall$ (A : Type) (i j n: nat) (a b : A),
G n = forml_atom i j $\rightarrow$ F i = wrap a $\rightarrow$ F j = wrap b $\rightarrow$
  encodes F G (FormlF n i j) (a = b).

Figure 3.4: The definition of the encodes relation.

3.2 The Reasoner

With the encoded formulas defined in the previous section, we now turn to the the construction of the decision procedure which acts on these meta-language objects. We implement this procedure as a lazy SMT solver, with the propositional and equational reasoning handled in two distinct phases contained within the `check_valida` function. For the validity checking, `check_valida` splits on each variable, and recursively calls itself with the updated assignment until it has created an assignment with the size specified by a natural number parameter. It then substitutes each variable in with the given assignment and simplifies the resulting formula. If the formula reduces to true, `check_valida` has found a satisfying assignment and returns true. Otherwise, it uses the decision procedure for $\mathcal{T}_=$ to see if the current assignment is consistent in $\mathcal{T}_=$ and returns true if it is not. After the recursive call, `check_valida` logically ands the results of the split together; therefore if it returns true, each split resulted in either a satisfying assignment or an inconsistent one and the supplied formula is a tautology modulo $\mathcal{T}_=$.

3.2.1 Tautology Functions

The validity checker relies on three main functions which deal with the simplification of formulas and the substitution of atomic variables in formulas. The `simplify` function takes in an encoded formula and recursively simplifies it according to the rules of boolean algebra, i.e. `simplify (AndF (BoolF True) (Atom 0) = Atom 0`. The `subst_var` function takes in a natural number, a boolean value, and a formula, and recurses through the formula to replace all atoms identified by the natural number with the provided boolean: `subst_var 0 true (ImpF (Eqn 0 2 4) (Atom 2) = ImpF (BoolF true) (Atom 2))`. Finally, the `subst_var_asgn` function calls `subst_var` on a formula for each variable from 0 to a given natural number, using a supplied list of booleans for the truth assignment. `check_valida` calls this function with the assignment it has generated, then checks the simplified result to see if it is a satisfying assignment.

```
Fixpoint check_valida (subst : nat) (φ : form)
   (asgn : (ilist bool)) struct subst : bool :=
match subst with
S n ⇒ (check_valida n φ (ilistc true asgn)) &&
   (check_valida n φ (ilistc false asgn))
| 0 ⇒ match (simplify (subst_var_asgn asgn φ (max_var phi))) with
   BoolF true ⇒ true
   | _ ⇒ match extern_dp asgn φ with
      Some true ⇒ true
      | _ ⇒ false
    end
  end
end.
```

Figure 3.5: The definition of `check_valida`.

3.2.2 Decision Procedure for $\mathcal{T}_=$

The main function for equation reasoning is `rewrite_form`. This function takes in a list of equations and a list of disequations as a list of formulas and uses the equations to assign an equivalence class to the terms in each of the disequations. Once this is done, it checks to see if there is a disequation between two elements of the same equivalence class, e.g. the assignment is inconsistent with $\mathbb{T}_=$. It does this by recursively using the head of the equation list to rewrite the remaining equations and the disequations. Once it has exhausted the equations, `rewrite_form` calls `false_diseqns`, which returns true if any of the rewritten disequations are between the same term. In this case, the assignment is contradictory and `rewrite_form` returns true. `check_valida` does not call `rewrite_form` directly; it instead calls a wrapper function, `extern_dp`, which generates a list of equations found in the formula that `check_valida` is considering and separates them into lists of equations and disequations based on the current truth assignment.

`check_valida`	Decision procedure for satisfiability modulo $\mathcal{T}_=$
`max_var`	Returns the maximum natural number atomic identifier in a given formula
`simplify_not`	Simplifies the negation of an embedded formula according to the rules of boolean algebra
`simplify`	Simplifies an embedded formula according to rules of boolean algebra, calls `simplify_not` on negated formulas
`subst_var`	Substitutes a truth value for a single atom
`subst_var_asgn`	Substitutes all atoms with truth values in an assignment
`false_diseqns`	Checks a list of equations for any equations between the same term
`rewrite_form`	Rewrites a list of disequations using a list of equations and calls `false_diseqns` on the resulting list
`conj_form_list'`	Builds a meta-language formula embedding the equivalences of atoms with their truth values
`build_atom_list`	Builds a list of all the propositional variables in a formula
`build_eqn_list`	Builds a list of all the equations in a formula
`eval_interp`	Evaluates an encoded formula according to a given interpretation
`eval_asgn`	Evaluates an encoded formula according to a given truth assignment

Figure 3.6: Descriptions of functions used by BAGAHK.

Chapter 4

Soundness

The first step in verifying our decision procedure is to show that it is sound- that whenever `check_valid` returns true, it is possible to construct the corresponding Coq-level proof term. In other words,

$$\texttt{check\_valid form} = \texttt{true} \rightarrow \texttt{encodes}\ \phi\ \texttt{form} \rightarrow \phi \qquad (4.1)$$

`check_valid` explores every possible truth assignment, returning true if each assignment either satisfies the formula or produces a contradiction; the two major theorems of our proof of soundness closely mirror this execution. For the satisfiability checker, we need to show that when an assignment is a model of a formula, the Coq-level formula is equivalent to $\mathbf{T}$, from which we can immediately prove the Coq-level formula. For the decision procedure for $\mathcal{T}_=$, we show that when an assignment results in a contradictory set of equations, we can derive $\perp$. Since $\forall\phi, \perp \rightarrow phi$, we can again easily prove the Coq-level formula. These two pieces build a proof of ϕ: by splitting on every proposition in the formula; these two theorems construct a proof of ϕ regardless of the its truth value since `check_valid` checks each assignment.

Throughout the soundness proof there is an assumption that the encoded propositions are equivalent to the truth values supplied to them in a given assignment. To express this we use a function, `conj_form_list`, that conjoins atoms and their truth assignments in an embedded formula using a list of atoms and their assignments. We can then use our existing `encodes` relation to connect this to a corresponding Coq-level term, including all the necessary information about the truth values of the atoms in our formula. Since we include no prior information about the truth value of propositions, an important step of the final proof is building the proposition encoded by this formula as `check_valid` explores each assignment.

4.1 Contradictory Truth Assignments

The first step of the proof that we can derive $\perp$ from contradictory assignments is to show that we can prove $\perp$ when `false_diseqns` returns true.

Lemma 4.1.1 (false_diseqns_False)**.** *For every list of equations,ϕ, all of which are assigned false in a given equation, we can derive $\perp$ whenever `false_diseqns` returns true, the truth equivalences of ϕ are encoded in some proposition ψ, and we assume ψ.*

Proof. We'll proceed by induction on ϕ. If ϕ is the empty list, `false_diseqns` will return false, contradicting our assumption that it returned true. In the inductive case, we first check the equation on the head of the list; if it is between the same natural number, i,we can use the encodes relation to peel off the first conjunct in ψ, which has the form $Fi = Fi \leftrightarrow \mathbf{F}$, a contradiction. If the two sides of the equation are not the same, we know that `false_diseqns` calls itself recursively and returns true on the rest of the list. We can derive a new proposition, ψ', that is encoded by the rest of the list by removing the first conjunct from ψ. We can use these two facts to apply our inductive hypothesis and derive $\perp$. $\square$

Our next goal is to show that we can derive a contradiction when `rewrite_form` returns true. Since `rewrite_form` essentially calls `false_diseqns` with a list that has been rewritten several times, it would seem that the previous lemma suffices to prove this goal. A critical assumption in that lemma, however, was that we had a proposition encoding the truth values of the atoms; each rewrite of the list also changes the Coq-level proposition that it encodes. In order to use the previous lemma, we must therefore show that we can derive the rewritten proposition from the previous one.

Lemma 4.1.2 (encodes_rewrite_list)**.** *Let ψ be a proposition which is encoded by a list of equations, ϕ, whose head is an equation,τ, which is equivalent to $\mathbf{T}$ in a given* *assignment. We can construct a new proposition, ψ', that is encoded by rewriting the tail of ϕ using τ.*

Proof. We'll induct on the length of ϕ. In the base case, τ is the only element in ϕ, so rewriting ϕ with τ results in the empty list; since this encodes $\mathbf{T}$, it follows immediately. For the inductive case, we can use the inductive hypothesis to build the proposition encoding the tail of the rewritten list and then use τ to rewrite the head of ϕ. Having updated the propositions, we also need to update the function used in the encodes relation to insure that equations with the same identifier refer to the same terms so that the formula correctly encodes the new proposition. We can then combine the two to build the proposition for the entire list. □

With this new lemma, we can now derive $\perp$ when `rewrite_form` returns true.

Lemma 4.1.3 (rewrite_form_False)**.** *Let ϕ_1 be a list of equations, ϕ_2 be a list of disequations, and ψ be the proposition they both encode. If `rewrite_form` ϕ_1 ϕ_2 returns true, we can derive $\perp$ from ψ.*

Proof. We'll again induct on the length of ϕ_1. In the base case, `rewrite_form` simply calls `false_diseqns` on ϕ_2, which returns true because `rewrite_form` does, so we can use REWRITE_FORM_FALSE to show $\perp$. In the inductive case, we first rewrite the equations and disequations using the equation on the head of the list. Next, we use ENCODES_REWRITE_LIST to build the proposition encoding the rewritten list from ψ. With this updated proposition, we can use the inductive hypothesis to arrive at the desired contradiction. □

Theorem 4.1.4 (extern_dp_False)**.** *Let ϕ be a formula and ψ be a proposition holding the truth values to the atoms in ϕ as given in some assignment. If `extern_dp` returns true, $\psi \to \perp$*

Proof. Since `extern_dp` is a wrapper which calls `rewrite_form` with the appropriate arguments, this follows immediately from the previous lemma. □

Theorem extern_dp_False : forall (F : nat → Wrap) (G : nat → atom_lookup) (asgn : ilist bool) (ϕ : form) (ψ: Prop), extern_dp asgn ϕ = Some true → encodes F G (conj_form_list asgn (rm_dupl_eqn (build_eqn_list ϕ))) ψ → ψ → False.

Figure 4.1: Coq formulations of extern_dp_False.

4.2 Satisfying Assignments

Since a formula is satisfied if it simplifies to true, we first show that a proposition that is encoded by such a formula is equivalent to **T**. Because `simplify` uses `simplify_not` to simplify a negated formula, the formulation of the proof must also state that a proposition encoded by a negated formula which simplifies to true is equivalent to **F**.

Lemma 4.2.1 (simplify_true)**.** *Given a formula ϕ which encodes a proposition ψ, if `simplify` ϕ returns true, $\psi \leftrightarrow \mathbf{T}$ and if `simplify_not` ϕ returns true, $\psi \leftrightarrow \mathbf{F}$.*

Proof. We'll proceed by structural induction on ϕ. For the base cases, if ϕ is either a propositional variable or an equation it will not simplify to true or false, contradicting that assumption. Alternatively, if ψ is **T** then $\psi \leftrightarrow \mathbf{T}$ trivially follows, and the case is symmetric when ψ is **F**. When ϕ is of the form $\mathbf{AndF}\phi_1\phi_2$, we know that ϕ_1 and ϕ_2 both simplify to true. We can use the appropriate conjunct of the inductive hypothesis to show the propositions they encode are equivalent to **T** and derive that their conjunction is as well. A similar argument holds for the **OrF** and **ImpF** constructors. Finally, in the case that ϕ is a negated formula, i.e. $\phi = \mathbf{NotF}\phi'$, we split on the value of `simplify_not` ϕ'. Because we included the stronger statement about `simplify_not` in the statement of the proof, we can now apply the induction hypothesis to `simplify_not`ϕ' to get that $\psi' \leftrightarrow \mathbf{F}$. From this, we can get $\psi = \neg\psi' \leftrightarrow \mathbf{T}$, completing our inductive proof that $\phi \leftrightarrow \mathbf{T}$. □

The previous lemma allows us to show that the after a formula has been completely rewritten by a satisfying assignment, the proposition it encodes is equivalent to true. We now need to connect this proposition to the one encoded by the original formula, constructing a string of equivalences showing that the original proposition is equivalent to true: $\psi \leftrightarrow \psi' \leftrightarrow \ldots \leftrightarrow \psi'' \leftrightarrow \mathbf{T}$. To establish this sequence, we'll first show how to construct a proposition encoded by a formula in which one of the atoms has been substituted with a truth value. We'll then show that this proposition is equivalent to that encoded by the original formula, under the assumption that the Coq-level atom is equivalent to the substituted truth value.

$$\begin{tikzcd}
\text{simplify (subst\_var\_asgn } \alpha\ \phi \text{ n') } = \arrow[d, dash, "\text{encodes F G } \phi\ \psi"] & \text{simplify (subst\_var\_asgn } \alpha \text{ (subst\_var (fetch\_elb' } \alpha \text{ n') } \phi \text{) n) } = \ldots = \arrow[d, dash, "\text{encodes F G (subst\_var (fetch\_elb' } \alpha \text{ n') } \phi\ \psi'"] & \text{BoolF true} \arrow[d, dash, "\text{encodes F G BoolF true } \mathbf{T}"'] \\
\psi \arrow[r, leftrightarrow] & \psi' \arrow[r, leftrightarrow] & \mathbf{T}
\end{tikzcd}$$

Figure 4.2: Equivalence of substituted formulas.

Lemma 4.2.2 (exists_subst_prop)**.** *Given a formula ϕ which encodes a proposition ψ, there exists a proposition, ψ', which is encoded by* ***subst_var*** *$bn\phi$, where b is a boolean and n is a natural number.*

Proof. We'll proceed by structural induction on ϕ. For the base cases, if ϕ is an atom with n as its identifier, we will instantiate ψ' with the appropriate proposition and leave it alone otherwise. For the inductive constructors, we can use the inductive hypotheses to construct propositions encoded by the subformulas. We can then build the correct proposition using the corresponding boolean operators and use this as a witness for the existence of the correct proposition. □

Lemma 4.2.3 (subst_var_eqv)**.** *Let ϕ be a formula which encodes a proposition, ψ, and let ψ' be a proposition which is encoded by* ***subst_var*** *$b\ n\ \phi$, where n is a natural number and b is the truth value given to the atom with identifier n by an assignment α. Assuming that each of the propositions encoded by the atoms in ϕ are equivalent to the truth values given in α, $\psi \leftrightarrow \psi'$.*

Proof. This proof is similar to the previous one, inducting on ϕ. In the base case, the equivalence of ψ and ψ' follows immediately from the assumption that the propositions are equivalent to the truth values in α. The inductive hypothesis can be used as above to prove the inductive case. The case when ϕ is an equation and its identifier is not equal to n provides an illustration of the motivation for using the encodes relation to associate propositions with formulas. Letting ϕ be `Eqn m j k`, ψ be $a = b$ and ψ' be $a' = b'$, we can use the information in the encodes relation for that $Fj =$ `wrap` a and $Fj =$ `wrap` a' to show that `wrap` $a =$ `wrap` a'. We can then use one of the dependent types axioms to show that $a = a'$. Using the same strategy, we can use these equalities to rewrite $a' = b'$ to $a = b$; the proof that $a = b \leftrightarrow a = b$ then immediately follows. □

Finally, the proposition encoding the truth values relies on ϕ and is changed after each substitution. We need to prove that we are able to create a new proposition at each step in order to successively apply the previous lemma. Since the statement of the final soundness proof has propositions to hold the truth values of the generic atoms and the equations, there will be separate proofs for each. With this in hand, we are able to prove that a formula which is satisfiable according to an assignment encodes a proposition that is equivalent to **T**.

Lemma 4.2.4 (encodes_subst_conja)**.** *If the proposition ψ_a encodes the truth values given by an assignment α for the atoms in a formula ϕ, there exists a proposition, ψ_a, which encodes the truth values for the atoms in the formula* ***`subst_var`*** *n (**`fetch_elb'`**α n) ϕ.*

Proof. Once again we'll induct on the structure of ϕ. In the base cases, if the atom is selected for substitution we'll instantiate with either **T** or **F** and the atom will remain unchanged otherwise. In the inductive case, we can use the inductive hypothesis to generate the correct propositions and then show that these encode the appended lists of atoms. □

Lemma 4.2.5 (encodes_subst_conje)**.** *If the proposition ψ_a encodes the truth values given by an assignment α for the equations in a formula ϕ, there exists a proposition, ψ_a, which encodes the truth values for the equations in the formula* ***`subst_var`*** *n (**`fetch_elb'`**α n) ϕ.*

Proof. Symmetric to the proof of ENCODES_SUBST_CONJA. □

Theorem 4.2.6 (subst_var_true)**.** *Let ϕ be a formula which encodes a proposition ψ and let α be an assignment to the variables of ϕ. If we assume that the propositional* *variables in ψ have the truth values given in α and* ***(subst_var_asgn phi*** *α* ***n)*** *simplifies to true for some natural number n, then ψ holds.*

Proof. We'll proceed by induction on n. In the base case, when n is 0, we can unfold `subst_var_asgn` to conclude that `subst_var (fetch_elb'` α `0) 0` ϕ simplifies to true. We can then use EXISTS_SUBST_PROP to show that the proposition encoded by this formula, ψ', exists and SIMPLIFY_TRUE to show that ψ' is equivalent to $\mathbf{T}$. By SUBST_VAR_EQV, $\psi \leftrightarrow \psi'$, so ψ is also equivalent to $\mathbf{T}$, i.e. ψ holds. In the inductive case, we again unfold `subst_var_asgn` to `subst_var (fetch_elb'` α `(S n')) (S n') (subst_var_asgn` α ϕ `n')`. Since substitution is commutative, we can conclude that `simplify (subst_var_asgn` α `(subst_var (fetch_elb'` α `(S n)) (S n)` ϕ`) n')` equals true. By ENCODES_SUBST_CONJA and ENCODES_SUBST_CONJE, we can constuct the propositions representing the truth values for the atoms in this new formula; we can also build the proposition representing this new formula using EXISTS_SUBST_PROP. With these updated propositions, we can apply the inductive hypothesis to show that this new proposition is equivalent to true. Finally, by SUBST_VAR_EQV, the original proposition ψ is equivalent to true as well, showing that ψ holds for any satisfying assignment.

□

Theorem subst_var_true : ∀ (F : nat → Wrap) (G : nat → atom_lookup)
  (n : nat) (asgn :ilist bool) (ϕ : form) (ψ_a ψ_e ψ: Prop),
  simplify (subst_var_asgn asgn ϕ n) = BoolF true →
  encodes F G (conj_form_list asgn (build_atom_list ϕ)) ψ_a →
  encodes F G (conj_form_list asgn (build_eqn_list ϕ)) ψ_e →
  encodes F G ϕ ψ → ψ_a → ψ_e → ψ.

Figure 4.3: Coq formulation of subst_var_true

4.3 Soundness of `check_valida`

A key assumption for EXTERN_DP_FALSE and SUBST_VAR_TRUE was that there was a proposition which embedded the truth values for the set of propositional atoms encoded by the formula being considered. The final step in proving that `check_valida` is sound is to show how to construct these propositions as `check_valid` explores the possible assignments. To this end, we show that for each truth value and type of proposition, it is possible to create an updated embedding of the truth values. With this done, we can show that regardless of the actual truth values of these propositions, if `check_valida` returns true, the encoded formula holds.

Lemma 4.3.1 (S_encodes_conj_atom_t)**.** *Let ψ be a proposition encoded by a formula ϕ, let α be an assignment to that formula, and let ψ_a be a proposition embedding the truth values of the atoms of ψ. If an atom n in ϕ doesn't have a value in α and the proposition which it encodes is equivalent to $\mathbf{T}$, then we can create a new proposition which embeds the truth values for ψ with an updated assignment that maps n to true.*

Proof. We will proceed with induction on the structure of ϕ. In the case that ϕ is an atom which has an assignment in α or if ϕ is an equation or a boolean, ϕ_a already embeds the truth values of ϕ. Alternatively, if ϕ is an unassigned atom, we can use the assumption that the proposition it embeds is equivalent to $\mathbf{T}$ to build ϕ'_a. In the inductive cases, we can use our inductive hypothesis to build the embeddings for the subformulas and then combine them to create ϕ'_a for the entire formula. □

We can use the same argument to construct a proof in the case that an atom n embeds a proposition which is false or when n indexes an equation that is either true or false. We'll call these lemmas S_ENCODES_CONJ_ATOM_F, S_ENCODES_CONJ_EQN_T, and S_ENCODES_CONJ_EQN_F respectively.

Theorem 4.3.2 (check_valida_sound')**.** *Let ψ be a proposition encoded by a formula ϕ, let α be an assignment to that formula, and let ψ_a and psi_e be propositions embedding the truth values of the atoms and equations of ψ respectively. If* ***`check_valida n`*** *α ϕ is true, and if n is greater than the difference of maximum identifier in ϕ and the maximum variable currently assigned in α, then ψ holds.*

Proof. We'll induct on n. If n is 0, either `subst_var_asgn` α ϕ (`max_var` ϕ) or `extern_dp` α ϕ is equal to true. In the former case, SUBST_VAR_TRUE proves ψ. In the latter case, EXTERN_DP_FALSE derives $\perp$ from ψ_e and ψ trivially holds. In the inductive step, $n = Sm$ for some m. Unfolding `check_valida` shows that `check_valida` `n` α_t ϕ and `check_valida n` α_f ϕ are both true, where α_t and α_f are modified versions of α which map Sm to true and false respectively. We then case split on the truth value of the proposition encoded by Sm: if this is **T** and Sm is an atom, we can use S_encodes_conj_atom_t to build a new proposition embedding the truth values assigned by α_t. We can use this and the fact that `check_valida n` α_t ϕ is true to apply the inductive hypothesis to prove ψ. The same argument applies when Sm is equivalent to **F** or Sm an equation; in all cases, the inductive hypothesis shows that ψ holds. □

```
Theorem check_valida_sound' : ∀ (F : nat → Wrap) (G : nat → atom_lookup)
  (n : nat) (asgn : ilist bool) (φ : form) (ψa ψe ψ: Prop),
   max_var φ < n + (ilength asgn) → check_valida n φ asgn = true →
  encodes F G (conj_form_list' asgn (build_atom_list φ)) ψa →
  encodes F G (conj_form_list' asgn (build_eqn_list φ)) ψe →
  encodes F G φ ψ → ψa → ψe → ψ.
```

Figure 4.4: Coq formulation of check_valida_sound'.

In order to have an strong enough induction hypothesis in the previous theorem, we had to generalize over all assignments and include propositions encoding the truth values given for all atoms currently assigned. The statement of this theorem is more general than is needed when `check_valida` is used as a tactic, however. In this case, we want `check_valida` to explore every possible assignment and its initial assignment should be empty. Thus, our final proof of soundness will be to show that if `check_valida` returns true when called on a formula with an empty assignment and a natural number large enough to explore every assignment, the proposition encoded by that formula is true.

Theorem 4.3.3 (check_valida_sound)**.** *Let ψ be a proposition encoded by a formula ϕ. If `check_valida (S (max_var` ϕ`)) ilistn` ϕ returns true, ψ holds.*

Proof. This is a specific instance of CHECK_VALIDA_SOUND'. Clearly, `S (max_var` ϕ`)` is greater than `(max_var` ϕ`) + ilength ilistn`, since the length of the empty list is zero. Furthermore, since the assignment is empty, the formulas embedding the truth values of the propositional variables and equations are simply `BoolF true`.

These formulas each encode **T**, which we can derive immediately. Since all of the assumptions of CHECK_VALIDA_SOUND' hold when `check_valida` is called with `S (max_var` ϕ`)` and an empty assignment, this theorem directly directly follows from the previous proof of CHECK_VALIDA_SOUND'. □

Theorem check_valida_sound : ∀ (F : nat → Wrap) (G : nat → atom_lookup) (ϕ : form) (ψ: Prop), check_valida n ϕ asgn = true → encodes F G ϕ ψ → ψ.

Figure 4.5: Coq formulation of check_valida_sound.

Chapter 5

Completeness

Having shown that our decision procedure is sound, we will now prove that it is complete, i.e. that if there is a proof of a Coq formula, `check_valida` will recognize its embedded representation as a theorem. Unfortunately, while the soundness proof allowed us to build a deduction of ψ, $\vdash_{\mathcal{P}} \psi$, assuming the existence of a proof of ψ provides no information on how it was derived. ψ could have been built using a theory other than $\mathcal{T}_=$: while a proof that $3 <> 4$ is easily shown in Coq, our decision procedure will not recognize it as a theorem. Instead, the completeness proof will constrain ψ to the set of propositions to $\mathcal{T}_=$ by showing how to construct a proof system for $\mathcal{T}_=$ and then proving that any derivable formula is valid in $\mathcal{T}_=$. Interestingly, this statement that $\vdash_{\mathcal{T}_=} P \rightarrow \models_{\mathcal{T}_=} P$ is the traditional proof of soundness. The final step of the proof is to show that our decision procedure recognizes any $\mathcal{T}_=$-valid formula.

There are three main styles of formalization for classical logic: natural deduction, Gentzen systems, and Hilbert systems [27]. In all three, deductions are thought of as *prooftrees* which have a formula at each node. Each node is a *conclusion* derived through the application of one of the system's proof rules to its children, called the *premises* of the rule application. The root of the tree is the conclusion of the entire deduction and the formulas at the leaves of the tree are the set of *assumptions* for the deduction. An assumption is initially *open*, it can be *closed* at a later point; assumptions are labeled with identifiers to designate which assumption is closed by a rule. An important feature of Hilbert systems is that assumption in deductions are never closed, obviating the need for labels. In addition, Hilbert systems have only one inference rule; the rest are replaced with a set of axiom schemes. Since all three formalisms are theoretically equivalent [27], these two features were the major motivation behind the decision to use a Hilbert system to formalize the proofs used in the completeness argument. Since all assumptions are open, there is no need to develop a method for creating and maintaining assumption labels. Furthermore, updating the completeness argument can be done by adding axioms for any new theories handled by the reasoner.

$$
\boxed{
\begin{array}{cc}
A \rightarrow (B \rightarrow A) \quad \mathbf{k} & (A \rightarrow (B \rightarrow C) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)) \quad \mathbf{s} \\
A \rightarrow A \vee B \quad \vee \mathbf{I_L} & (A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow (A \vee B \rightarrow C)) \quad \vee \mathbf{E} \\
B \rightarrow A \vee B \quad \vee \mathbf{I_R} & A \wedge B \rightarrow A \quad \wedge \mathbf{E_L} \\
A \rightarrow (B \rightarrow (A \wedge B) \quad \wedge \mathbf{I} & A \wedge B \rightarrow B \quad \wedge \mathbf{E_R} \\
\bot \rightarrow A \quad \bot \mathbf{I} & \neg\neg A \rightarrow A \quad \mathbf{D_n} \\
a = a \quad =_{\mathbf{R}} & a = b \rightarrow b = a \quad =_{\mathbf{S}} \\
a = b \rightarrow (b = c \rightarrow a = c) \quad =_{\mathbf{T}} & \\
\dfrac{A \quad A \rightarrow B}{B}\ \mathbf{MP} &
\end{array}
}
$$

Figure 5.1: The axioms and inference rule for classical logic with equational reasoning.

Example 1. *A deduction of $A \rightarrow A$:*

$$
\dfrac{\dfrac{[A \rightarrow ((A \rightarrow A) \rightarrow A)] \rightarrow [(A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)] \quad A \rightarrow ((A \rightarrow A) \rightarrow A)}{(A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)} \qquad A \rightarrow (A \rightarrow A)}{A \rightarrow A}
$$

As the above example demonstrates, deductions in Hilbert systems can grow quite large and are often difficult to construct; asking a user to build such a proof would be an unreasonable burden. The important point here, however, is that such a proof never has to be built by BAGAHK, nor by a user. Unlike the proof of soundness, which was critical for building a proof term from a valid formula, the completeness argument serves only as a proof of correctness. It suffices that such a proof *could* be built using these rules, since a deduction for every valid formula is derivable from these rules [27]. Under this framework, the proof obligation for completeness is to show that if a proposition, ψ, is the conclusion of some deduction using the rules in Fig. 5.1, then `check_valida` will return true on any formula encoding ψ.

A final question before proceeding to the completeness proof is how to represent a deduction of a proposition using the Hilbert system in Fig. 5.1. As with the `encodes` relation, these prooftrees can be expressed as instances of an inductive data type. This `prop_eq_prf` data type is indexed by a formula and a proposition encoded by that formula, with the eleven axioms providing the base constructors. There is a single inductive constructor for the modus ponens inference rule; the `prop_eq_prf` arguments passed to this constructor are the premises of the rule, and the resulting `prop_eq_prf` is its conclusion. Through successive applications of this inductive constructor, a prooftree for any valid proposition can be built.

Inductive prop_eq_prf : form $\rightarrow$ Prop $\rightarrow$ Prop:=
K : $\forall$ (A B : Prop) (ϕ_A ϕ_B : form),
  prop_eq_prf (ImpF ϕ_a (ImpF ϕ_B ϕ_A)) (A $\rightarrow$ (B $\rightarrow$ A))
| eq_refl : $\forall$ (T : Type) (a: T) (i n : nat), prop_eq_prf (Eqn n i i) (a = a)
| MP : $\forall$ (A B : Prop) (ϕ_A ϕ_B : form),
  prop_eq_prf(ImpF ϕ_A ϕ_B) (A $\rightarrow$ B) $\rightarrow$ prop_eq_prf ϕ_A A $\rightarrow$ prop_eq_prf ϕ_B B.

Figure 5.2: Coq definitions for part of the constructors of `prop_eq_prf`. The three shown here represent for the axioms **k** and $=_{\mathbf{R}}$ and Modus Ponens.

With this formulation in hand, we can now constrain the formulas in the statement of the completeness proof to those which are derivable in this proof system. To show completeness, we will first show that if there exists a proof of a Coq formula ψ using the rules in Fig. fig:proof-rules, the embedding of that formula, ϕ, is satisfied by all truth assignments consistent with $\mathcal{T}_=$. Secondly, if `check_valida` fails to report that ϕ is a theorem, there must exist some truth assignment which is consistent with $\mathcal{T}_=$ and which falsifies ϕ. Since all interpretations satisfy ϕ, the contrapositive of the previous statement tells us that `check_valida` will report that ϕ is valid. Thus, any Coq formula derivable in our proof system will be reported valid by `check_valida` and this decision procedure is complete.

5.1 Consistent Truth Assignments

The first step of the completeness proof is to show that every proposition provable by the axioms in 5.1 is satisfied by every truth assignment consistent with $\mathcal{T}_=$– that $\vdash_{\mathcal{T}_=} \psi$ implies $\models_{\mathcal{T}_=} \psi$. When we discuss a truth *assignment*, we mean a mapping from the atoms of a formula to their truth values. The validity checker generates all possible truth assignments, and `extern_dp` verifies that any falsifying assignment is consistent with $\mathcal{T}_=$. For the purpose of this proof, we wish to define consistent truth assignments independently of `extern_dp`. Instead of directly mapping the atoms of a formula to a truth value, we can evaluate a formula using an *interpretation* which assigns a meaning to each of the atoms of the formula; the truth values of the atoms are assigned implicitly based on their interpretation. For example, consider the equation $x = y$: x and y each belong to an equivalence class. If they belong to the same equivalence class, this proposition is true; if they are members of distinct classes, it is false. We will define an interpretation in $\mathcal{T}_=$ as a function which maps propositional variables to truth values and the terms in equations to equivalence classes.

The first step of this proof requires the creation of a method of expressing that the truth values implied by an interpretation satisfy a formula. This is accomplished through the use of the recursive `eval_interp` function, which takes in a formula, ϕ, a function, I_P, mapping natural numbers to booleans for the interpretation of propositional atoms, and a function, I_E, mapping natural numbers to a natural number representing a distinct equivalence class. Equations evaluate to true when the terms on either side are mapped to the same equivalence class and false otherwise. Propositional atoms are evaluated using I_P and the other constructors are mapped to the corresponding boolean connective.

If ϕ evaluates to true, I_P and I_E generate a model of ϕ, allowing us to define the models of a formula in terms of `eval_interp`. We can now use this function to express the first step of the completeness proof: if a proposition ψ is provable using the proof system in Fig. 5.1, then that formula is satisfied by all interpretations.

Theorem 5.1.1 (prop_eq_prf_interp)**.** *For a formula ϕ and a proposition ψ, if there is a propositional proof of ψ and ϕ, ϕ evaluates to true for all interpretations I_P and I_E.*

Proof. Each of the base cases representing a propositional reasoning axiom is a logical tautology, a case split on the values to which each of propositional atoms is mapped yields in a boolean formula equal to true. The $=_{\mathbf{R}}$ constructor trivially evaluates to true. For the $=_{\mathbf{S}}$ and $=_{\mathbf{T}}$, a case split on the equality of the equivalence classes of the terms will expose the underlying truth values of the equations; in both cases, the resulting formulas will evaluate to true. In the inductive case, we assume there is a propositional proof of ϕ_B and we need to show that `eval_interp` I_P I_E ϕ_B equals true. In order to construct the propositional proof of ϕ_B using the inductive inference constructor, there must be propositional proofs for ϕ_A and $\phi_A \rightarrow \phi_B$. We can use the inductive hypothesis to conclude that for all I'_P and I'_E, `eval_interp` I'_P I'_E ϕ_A and `eval_interp` I'_P I'_E $\phi_A \rightarrow \phi_B$ must be true. If there are falsifying interpretations of ϕ_B, I_P and I_E, then `eval_interp` I_P I_E ϕ_A is true and `eval_interp` I'_P I'_E ϕ_B is false, but `eval_interp` I'_P I'_E $\phi_A \rightarrow \phi_B$ must also be false, contradicting a conclusion of the inductive hypothesis. We can thus conclude that `eval_interp` I_P I_E ϕ_B for all I_P I_E. □

Theorem prop_eq_prf_interp :
  $\forall$ (ϕ : form) (ψ : Prop), prop_eq_prf ϕ ψ $\rightarrow$
  $\forall$ (I_P : nat $\rightarrow$ bool) (I_E : nat $\rightarrow$ nat), eval_interp I_P I_E ϕ = true.

Figure 5.3: Coq definition of `prop_eq_prf_interp`.

5.2 Interpretation and Assignment Agreement

In order to connect the propositional proofs to `check_valida`, we must associate the interpretations from the previous section with the truth assignments checked by `check_valida`. We will introduce a new function, `eval_asgn`, which evaluates a formula according to a given truth assignment. We will say that an interpretation and an assignment *agree* on a list of atoms if the same truth value is given by the interpretation and the assignment to each element in the list.

Lemma 5.2.1 (ia_agree_eval)**.** *If an interpretation, I_P and I_E, and an assignment α agree on a list of all atoms in a formula ϕ, then the output of* `eval_asgn` *is equal to the output of*`eval_interp`*.*

Proof. This proof is by induction on ϕ. In the base cases, the assumption that the interpretation and the atom agree guarantees that the two have the same truth value. In the inductive cases, the inductive hypothesis shows that the evaluations of the subformulas agree; combining the evaluations with the appropriate boolean connective shows that the evaluations of the whole formula are also equal. $\square$

We introduced interpretations as a means of expressing consistent truth assignments independently of `extern_dp` for the first step of the completeness proof. In order to connect derivable formulas to `check_valida`, we will now show that every interpretation corresponds to a truth assignment recognized as consistent by `extern_dp`. To make the proofs easier, we will further divide agreement in terms of propositional atoms and equations, first showing that an interpretation always exists which agrees with an assignment to just propositional atoms. We'll then show that a consistent assignment agrees with *some* equational interpretation, then combine the two to show how to build the complete interpretation. Having shown the link between interpretations and truth assignments, we will then show that the `subst_var_asgn` function used by `check_valida` to check satisfiability is equivalent to `eval_asgn`. Thus, if `subst_var_asgn` reports that a truth assignment falsifies a formula ϕ, there must exist a corresponding interpretation which also falsifies ϕ.

Lemma 5.2.2 (ex_ia_agree_atom)**.** *For any assignment α and list of propositional atoms τ, there exists an interpretation I_P which agrees with α on τ.*

Proof. We'll induct on τ. When τ is the empty list, any interpretation trivially agrees with α. In the inductive case, we can use the inductive hypothesis to derive a function I_P, which agrees with the tail of the list. We can then create a new interpretation, I'_P, which maps the head of the list to the truth value in the assignment and everything else to the values given by I_P. We can then again induct on τ to show that I'_P agrees with α: if the head of τ appears in the rest of the list, it must already be mapped to the truth value given in α, otherwise, the updated function will act the same as I_P on the tail of the list. $\square$

Lemma 5.2.3 (ia_agree_eqn_rewrite)**.** *For any assignment α, a list of propositional atoms τ, and two natural numbers i and j, if there is an interpretation I_E which* *agrees with α on* **`rewrite_list`** *i j τ, there is an interpretation, I'_E, mapping i to I_E j and all other numbers x to I_E x, which agrees with α on τ.*

Proof. We'll induct on τ. The base case is again trivial. In the inductive case, if the head of the list does not have an i, the two interpretations trivially map to the same truth value. Otherwise, since i has already been replaced by j in `rewrite_list` i j τ, I'_E returns the same value on τ as I_E does on the rewritten list. In either case, since the latter agrees with α on the head of the list, so must I'_E, and the inductive hypothesis proves that I'_E also agrees on the rest of the list. □

Theorem 5.2.4 (ia_agree_eqn_extern_dp)**.** *Let ϕ_1 be a list of equations, ϕ_2 be lists of disequations, both according to an assignment α. If* **`rewrite_form`** *ϕ_1 ϕ_2 returns true, there exists an interpretation, I_E, which agrees with α on both ϕ_1 and ϕ_2. Since* **`extern_dp`** *is a wrapper for* **`rewrite_form`***, the results of this theorem translate to any formula on which* **`extern_dp`** *returns true.*

Proof. This proof will induct on the list of equations, ϕ_1. When phi_1 is the empty list, we can construct a function which maps every term to its natural number identifier. Since `rewrite_form` did not find a contradiction, all of the equations in ϕ_2 are between distinct terms, so each of its element will evaluate to false. Since ϕ_2 is a set of disequations according to α, this interpretation will agree with α on ϕ_2. In the inductive case, the equation on the head of ϕ_1 will be used to rewrite both ϕ_1 and ϕ_2. Since this will produce a smaller list in the parameter of recursion, the inductive hypothesis can be applied to derive an interpretation which agrees with the rewritten ϕ_1 and ϕ_2. IA_AGREE_EQN_REWRITE shows that there is an updated interpretation,I'_E,agreeing with α on the original ϕ_2 and the tail of the original ϕ_1. Finally, since I'_E will assign both sides of the equation at the head of ϕ_1 to the same equivalence class, it agrees with α, which maps all the equations in ϕ_1 to true. Thus, I'_E serves as a witness for the existence of an interpretation that agrees with α, completing the proof. □

Lemma 5.2.5 (merge_ia_agree)**.** *Given an interpretation for propositional atoms, I_P, an interpretation for equations, I_E, and an assignment α, if the two interpretations agree separately with α on a list of atoms τ, then the combined interpretation does as well.*

Lemma ia_agree_eqn_extern_dp : ∀ (α: ilist bool) (n: nat) (ϕ_1 ϕ_2: ilist form),
rewrite_form (fst (sep_form α ϕ_1)) (snd (sep_form α ϕ_2)) n = Some false →
ilength (fst (sep_form α ϕ_1)) $\leq n$ →
∃ I_E, ia_agree_eqn I_E (fst (sep_form α ϕ_1)) α = true ∧
ia_agree_eqn I_E (snd (sep_form α ϕ_2)) α = true.

Figure 5.4: Coq definition of `ia_agree_eqn_extern_dp`.

Proof. Inducting on τ, the base case when τ is the empty list is again immediate. In the inductive case, the head of τ is either a propositional atom or an equation. In the former case, the assumption that I_P agrees with α ensures that they both assign the head the same truth value; the latter case is symmetric. Furthermore, since both I_P and I_E agree with all of τ, they both agree with its tail, allowing us to apply the inductive hypothesis to show that the combined interpretation agrees with α on the rest of τ as well. □

Lemma 5.2.6 (subst_var_eval_asgn)**.** *Let α be a complete assignment for a formula ϕ. Then `subst_var_asgn` α ϕ simplifies to `BoolF (eval_asgn` α ϕ`)`.*

Proof. The proof will proceed with structural induction on ϕ. In the base cases, both sides will simply replace the formula with the boolean value given by α. In the case that ϕ is `AndF` ϕ_1 ϕ_2, the inductive hypothesis can be used on the subformulas to show that they simplify to `BoolF (eval_asgn` α ϕ_1`)` and `BoolF (eval_asgn` α ϕ_2`)` respectively. From here, we can split on the values of the evaluations, when both are true, both sides reduce to `BoolF true`; in all other cases, both sides reduce to `BoolF false`. A similar approach works for the other inductive cases, completing the proof. □

5.3 Completeness of `check_valida`

We can now combine these lemmas to connect `check_valida` to the propositional proofs, showing that it explores all possible interpretations, returning true if a formula is valid modulo $\mathcal{T}$.

Theorem 5.3.1 (prop_eq_prf_complete)**.** *If there exists a proof of a proposition ψ and formula ϕ using the proof rules in Fig. 5.1,* ***`check_valida`*** *ϕ α* ***`(max_var`*** *ϕ**`)`** will return true for any assignment α.*

Proof. In the case that `check_valida` is true, the proof is completed. In the case that `check_valida` is false, we know that there must be some assignment,α', for which `simplify (subst_var_asgn` α' ϕ`)` is `BoolF false`, and for which `extern_dp` α' ϕ is also false. The second fact applied to IA_AGREE_EQN_EXTERN_DP yields an interpretation that can be combined with the interpretation from EX_IA_AGREE_ATOM by MERGE_IA_AGREE to produce an interpretation, I_P and I_E, which agrees with α'. Furthermore, by SUBST_VAR_EVAL_ASGN `eval_asgn` α ϕ is equal to false. By IA_AGREE_EVAL, the interpretation I_P I_E falsifies ϕ. However, since we assumed that there is a propositional proof of ϕ, by PROP_EQ_PRF_INTERP all interpretations of ϕ evaluate to true, contradicting the previous deduction that I_P I_E falsifies ϕ. Since the assumption that `check_valida` is false results in a contradiction, we can conclude that whenever there is a propositional proof of ϕ, `check_valida` α ϕ returns true. $\square$

Theorem prop_eq_prf_complete : $\forall$ (ϕ : form) (α : ilist bool) (ψ : Prop),
  prop_eq_prf phi $\psi \rightarrow$ check_valida (max_var ϕ) ϕ α = true.

Figure 5.5: Coq definition of `prop_eq_prf_complete`.

Chapter 6

Conclusion

In chapter 3, we presented the implementation of a decision procedure for testing the validity of a formula modulo the theory of ground equations, $\mathcal{T}_=$. We have extended the definition of 'verified' normally used for sound decision procedures by showing that ours is both sound and complete. This soundness proof also provides the critical piece for a the reflection method which allows our decision procedure to be used as a Coq tactic in assisting users in construction other proofs. All of these proofs have been formalized as Coq proof scripts which have been mechanically checked and which are assembled in the BAGAHK package.

6.1 Assumptions

As with any proof, our work makes some basic assumptions which, if incorrect, could compromise our conclusions. Most important is that Coq is sound, since Coq relies on a small trusted core of unverified code. If this core were to be unsound, the Coq formulation of our soundness and completeness arguments could be incorrect. Given the maturity of Coq and its widespread use in research, we are confident that it is a firm foundation for our work. We also assume that the axiomization in Fig. 5.1 is correct: if this were only a fragment of $\mathcal{T}_=$, there could be theorems of $\mathcal{T}_=$ which `check_valida` could not recognize. The propositional axioms and modus ponens rules have accompanying paper proofs in [27] showing they are sound and complete, and the equational axioms added to extend this to $\mathcal{T}_=$ are standard. Another source of concern is that the encoding relation could incorrectly map the embedded formulas to Coq terms. At one extreme, if `encodes` mapped every reflected formula to **T**, a proof would be immediate, but the term constructed by the soundness proof would not be correct. In both cases, we are confident that our formulations are correct, but these examples show some areas in which care must be taken when implementing additional theories.

6.2 Efficiency of `check_valida`

While the question of a polynomial time algorithm for SAT remains an important open question in computer science, there exist more efficient algorithms than the enumeration of all variables used in `check_valida`. In particular, the backtracking DPLL [9] algorithm has proven effective for many SAT-instances and remains the backbone for most modern SAT-solvers [18] and SMT-solvers [2]. In general, more complex algorithms are more difficult to mechanically verify, which is why we have chosen to implement a very simple and direct SAT-solver in `check_valida`. The run-time complexity of our algorithm cannot be ignored, however, particularly since it is intended for use as a tactic. For this reason, we have implemented a modified version of the decision procedure, called `check_valid'`, which simplifies the formula after every substitution. In this version of the satisfiability checker, portions of the formula can be simplified away after each substitution, reducing the size of the formula in subsequent iterations. Rather than repeat the proofs of soundness and correctness for this new version, we instead show that it is operationally equivalent to `check_valida`-that on the same input, the two have the same output. Once this lemma has been proven, `check_valid'` can then be proven sound and complete by simply rewriting the instances of `check_valida` in the corresponding proofs for that method.

Lemma chv_chva_eqv' : $\forall$ (n : nat) (ϕ : form), max_var ϕ ¡ n $\rightarrow$
check_valid' n ϕ ilistn (rm_dupl_eqn (build_eqn_list ϕ)) =
check_valida n ϕ ilistn.

Figure 6.1: Coq formulations of the proof that `check_valida` and `check_valid'` are equivalent.

6.3 Additional Theories

Adding a new theory to our decision procedure would require a number of updates to BAGAHK. For each of the atoms of the new theory, a new constructor would have to be added to `form` and the encodes relation would need to be updated accordingly. `extern_dp` would have to be exetend to include a new function which checked the consistency of a truth assignment with the new background theory. Most of the lemmas used in the soundness proof would require minor modifications to reflect the new constructor; these cases will closely follow those of the other atoms. The key piece in the revised proof would be a theorem deriving $\perp$ for any assignment which the updated `extern_dp` identified as contradictory. The completeness argument would require more work. Axioms for the new theory would need to be developed and included in the proof system; these new proof rules would need to carefully constructed so that they are not inconsistent and also able to derive all valid formulas. A new definition of interpretation would have to be developed for the new theory, most likely through the addition of a new interpretation function to `eval_interp`. In addition to updating the lemmas establishing the connection between interpretations and truth assignments, IA_AGREE_EQN_EXTERN_DP would need to be extended to show that any assignment deemed to be consistent by `extern_dp` corresponds to an interpretation. The final step of the completeness proof would be the same, showing that if there is a derivation using the updated proof rules, then all interpretations are true, which in turn implies that `check_valida` must recognize the embedded formula as a theorem.

6.4 Future Work

While our decision procedure for validity modulo the ground theory of equations adds functionality to Coq, there is much work to be done to bring it to the level of modern SMT-solvers. Most of these solvers use a DPLL-style satisfiability checker; [2] investigates integrating this with the theory decision procedures to achieve better performance. Any modifications to the validity checker could be done using the equivalence method from 6.1, leaving the soundness and completeness proofs unchanged. Furthermore, these solvers also handle a large number of theories which could also be added to BAGAHK according to the method outlined in the previous section. $\mathcal{T}_=$ is a subset of the theory of uninterpreted functions with equality; extending our current decision procedure to the full theory would be a logical next step. Additionally, the theories of the integers and real numbers are often found in Coq proofs, making them good candidates for inclusion in BAGAHK. Complementing the theoretical contributions that any of these improvements would make would be their practical use in the application of BAGAHK as a tactic.

Appendix A

Coq Formulations of Important Lemmas and Theorems

Section 4.1

Lemma false_diseqns_False : ∀ (F : nat → Wrap) (G : nat → atom_lookup) (ψ: Prop) (α : ilist bool) (ϕ : ilist form), false_diseqns (snd (sep_form α ϕ)) = true → encodes F G (conj_form_list α ϕ) ψ → ψ → False.
Lemma encodes_rewrite_list : ∀ (F: nat → Wrap) (G : nat → atom_lookup) (α : ilist bool) (n j k : nat) (ϕ : ilistform) (ψ: Prop), fetch_elb' α n = true → encodes F G (conj_form_list α (ilistc (Eqn n j k) ϕ)) ψ → ψ → ∃ ψ', encodes F (update_atom_lookup G j k) (conj_form_list α (rewrite_list ϕ j k (ilength ϕ))) ψ' $\wedge\psi'$.
Lemma rewrite_form_false : ∀ (F : nat → Wrap) (ilen : nat) (G : nat → atom_lookup) (α : ilist bool) (ϕ : ilist form) (ψ: Prop), ilength (fst (sep_form α ϕ)) ≤ ilen → rewrite_form (fst (sep_form α ϕ)) (snd(sep_form α ϕ)) ilen = Some true → encodes F G (conj_form_list α ϕ) ψ → ψ → False.
Theorem extern_dp_False : forall (F : nat → Wrap) (G : nat → atom_lookup) (α : ilist bool) (ϕ : form) (ψ: Prop), extern_dp α ϕ = Some true → encodes F G (conj_form_list α (rm_dupl_eqn (build_eqn_list ϕ))) ψ → ψ → False.

Section 4.2

Lemma simplify_true : ∀ (F : nat → Wrap) (G : nat → atom_lookup) (ϕ : form) (ψ: Prop), (encodes F G ϕ ψ) → ((simplify ϕ = BoolF true) → ψ) ∧

((simplify_not ϕ = BoolF true) $\rightarrow \neg\psi$).

Lemma exists_subst_prop : $\forall$ (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup) (b : bool) (n : nat) (ϕ : form) (ψ: Prop), encodes F G ϕ ψ $\rightarrow \exists\psi$', encodes F G (subst_var b n ϕ) ψ'.

Lemma subst_var_eqv : $\forall$ (F : nat $\rightarrow$ Wrap)(G : nat $\rightarrow$ atom_lookup) (n : nat) (α : ilist bool) (ϕ : form) (ψ_a ψ_e ψ ψ': Prop),
encodes F G (conj_form_list α (build_atom_list ϕ)) ψ_a $\rightarrow$
encodes F G (conj_form_list α (build_eqn_list ϕ)) ψ_e $\rightarrow$ (encodes F G ϕ ψ) $\rightarrow$
(encodes F G (subst_var (fetch_elb' α n) n ϕ) ψ') $\rightarrow \psi_a \rightarrow \psi_e \rightarrow (\psi' \leftrightarrow \psi)$.

Lemma encodes_subst_conja : $\forall$ (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup) (n : nat) (α : ilist bool) (ϕ : form) (ψ_a: Prop), ψ_a $\rightarrow$
encodes F G (conj_form_list α (build_atom_list ϕ)) ψ_a $\rightarrow \exists$ ψ'_a,
encodes F G (conj_form_list α (build_atom_list (subst_var (fetch_elb' α n) n ϕ))) $\psi'_a \wedge \psi'_a$.

Lemma encodes_subst_conje : $\forall$ (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup) (n : nat) (α : ilist bool) (ϕ : form) (ψ_e: Prop), ψ_e $\rightarrow$
encodes F G (conj_form_list α (build_eqn_list ϕ)) ψ_e $\rightarrow \exists$ ψ'_e,
encodes F G (conj_form_list α (build_eqn_list (subst_var (fetch_elb' α n) n ϕ))) $\psi_e \wedge \psi'_e$.

Theorem subst_var_true : $\forall$ (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup) (α :ilist bool) (n : nat)(ϕ : form) (ψ_a ψ_e ψ: Prop), simplify (subst_var_asgn asgn ϕ n) = BoolF true $\rightarrow$
encodes F G (conj_form_list α (build_atom_list ϕ)) ψ_a $\rightarrow$
encodes F G (conj_form_list α (build_eqn_list ϕ)) ψ_e $\rightarrow$
encodes F G ϕ ψ $\rightarrow \psi_a \rightarrow \psi_e \rightarrow \psi$.

Section 4.3

Lemma S_encodes_conj_formt : $\forall$ (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup) (ϕ : form) (α : ilist bool) (ψ ψ_a P: Prop), G (ilength α) = prop_atom P $\rightarrow$(P $\leftrightarrow$ **T**) $\rightarrow$
encodes F G ϕ ψ $\rightarrow$ encodes F G (conj_form_list' α (build_atom_list ϕ)) $\psi_a \rightarrow \psi_a \rightarrow \exists$ ψ'_a,
encodes F G (conj_form_list' (ilistc true α) (build_atom_list ϕ)) $\psi'_a \wedge \psi'_a$.

Theorem check_valida_sound : $\forall$ (F : nat $\rightarrow$ Wrap) (G : nat $\rightarrow$ atom_lookup) (n : nat) (α : ilist bool) (ϕ : form) (ψ_a ψ_e ψ: Prop), max_var ϕ ¡ n + (ilength α) $\rightarrow$
check_valida n ϕ α = true $\rightarrow$ encodes F G (conj_form_list' α (build_atom_list ϕ)) ψ_a $\rightarrow$
encodes F G (conj_form_list' α (build_eqn_list ϕ)) ψ_e $\rightarrow$ encodes F G ϕ ψ $\rightarrow \psi_a \rightarrow \psi_e \rightarrow \psi$.

Section 5.1

Theorem prop_prf_interp : ∀ (ϕ : form) (ψ : Prop), prop_prf ϕ ψ → ∀ (I_P : nat → bool) (I_E : nat → nat), eval_interp I_P I_E ϕ = true.

Section 5.2

Lemma ia_agree_eval : ∀ (α : ilist bool) (ϕ: form) (P : nat → bool) (E : nat → nat), ia_agree P E (build_atom_eqn_list ϕ) α = true → eval_asgn α ϕ = eval_interp P E ϕ.
Lemma ex_ia_agree_atom : ∀ (α : ilist bool) (ϕ : ilist form),∃ P, ia_agree_atom P ϕ α = true.
Lemma ia_agree_eqn_rewrite : ∀ (α : ilist bool) (ϕ : ilist form) (i j : nat) (E : nat → nat), ia_agree_eqn E (rewrite_list ϕ i j (ilength ϕ)) α = true → ia_agree_eqn (fun (x : nat) ⇒ if eq_nat_dec x i then E j else E x) ϕ α = true.
Lemma ia_agree_eqn_extern_dp : ∀ (α: ilist bool) (n: nat) (ϕ_1 ϕ_2: ilist form), rewrite_form (fst (sep_form α ϕ_1)) (snd (sep_form α ϕ_2)) n = Some false → ilength (fst (sep_form α ϕ_1)) $\leq n$ → ∃ I_E, ia_agree_eqn I_E (fst (sep_form α ϕ_1)) α = true ∧ ia_agree_eqn I_E (snd (sep_form α ϕ_2)) α = true.
Lemma merge_ia_agree : ∀ (α : ilist bool) (ϕ : ilist form) (P : nat → bool) (E : nat → nat), ia_agree_atom P ϕ α = true → ia_agree_eqn E ϕ α = true → ia_agree P E ϕ α = true.
Theorem subst_var_eval_asgn : ∀ (α : ilist bool) (len : nat) (ϕ : form), max_var ϕ ¡= len → simplify (subst_var_asgn α ϕ len) = BoolF (eval_asgnn α ϕ).

Section 5.3

Theorem prop_prf_complete : ∀ (ϕ : form) (α : ilist bool) (ψ : Prop), prop_prf phi ψ → check_valida (max_var ϕ) ϕ α = true.

References

[1] H. Barendregt. *Lambda Calculi with Types*, volume 2, pages 117–309.

[2] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in sat modulo theories. In *13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'06*, Lecture Notes in Computer Science. Springer, 2006.

[3] Clark Barrett and Cesare Tinelli. Cvc3. In *Procedings of the 19th International Conference on Computer Aided Verification, CAV 2007*, Lecture Notes in Computer Science. Springer, 2007.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, Berlin, 2004.

[5] Marco Bozzano, Robterto Bruttomesso, Alessandro Comatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. The mathsat 3 system. In *Procedings of the 20th International Conference on Automated Deduction, CADE 2005*, Lecture Notes in Computer Science. Springer, 2005.

[6] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[7] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[8] Haskell B. Curry and Robert Feys. *Combinatory Logic I*. North-Holland, 1958.

[9] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] David Delahaye and Micaela Mayero. Diophantus' 20th problem and fermat's last theorem for n=4. http://coq.inria.fr/contribs/Fermat4.html.

[11] Bruno Dutertre and Leonardo de Moura. The yices smt solver. Technical report, Computer Science Laboratory, SRI International, 2006. http://yices.csl.sri.com/tool-paper.pdf.

[12] Jonathan Ford and Natarajan Shankar. Formal verification of a combination decision procedure. In H. Kirchener and C. Kirchner, editors, *Procedings of the 18th International Conference on Automated Deduction, CADE 2002*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[13] Georges Gonthier. A computer-checked proof of the four color theorem. Technical report, Microsoft Research Cambridge, 2004.

[14] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995.

[15] C.A.R. Hoare. An axiomatic approach for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[16] William A. Howard. The formula-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on combinatory logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[17] Joao Marques-Silva and Kareem A. Sakallah, editors. *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501/2007 of *Lecture Notes in Computer Science*. Springer, May 2007.

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Procedings of the 38th Design Automation Conference*, 2001.

[19] Robert Nieuwenhuis and Albert Oliveras. Decision procedures for sat, sat moduluo theories and beyond: The barcelogictools (invited paper). In *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, Lecture Notes in Computer Science. Springer, 2005.

[20] Tobias Nipkow, Lawrence C. Paulson, and Marcus Wenzel. *A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, 2005.

[21] Russel O'Conner. Essential incompleteness of arithmetic verified in coq. In *Procedings of the 18th International Conference Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[22] Martijn Oostdijk and Herman Geuvers. Proof by computation in the coq system. *Theoretical Computer Science*, 272:293–314, 2002.

[23] Julia Robinson. Definability and decision problems in arithmetic. *The Journal of Symbolic Logic*, 14(2):98–114, June 1949.

[24] Natarajan Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[25] Satisfiability modulou theories competition, 2007. http://smtcomp.org.

[26] Laurent Théry. A certified version of buchberger's algorithm. In H. Kirchener and C. Kirchner, editors, *Procedings of the 15th International Conference on Automated Deduction, CADE 1998*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[27] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, Cambridge, 2nd edition, 2000.

Vita

Benjamin J. Delaware

Date of Birth	April 20, 1983
Place of Birth	Kirksville, Missouri
Degrees	B.S. Magna Cum Laude, Computer Science, August 2005 B.A. Magna Cum Laude, Russian, August 2005 Truman State University, Kirksville, Missouri
	M.S. Computer Science, August 2007 Washington University in St. Louis, St. Louis, Missouir
Honors	Math & Computer Science Department Honors, 2005 Truman State University, Kirksville, Missouri
	Distinguished Master's Fellowship, 2005 - 2006 Washington University in St. Louis, St. Louis, Missouri
	Microelectronics and Computer Development Doctoral Fellowship, 2007 The University of Texas, Austin, Texas
	College of Natural Sciences Dean's Excellence Award, 2007 The University of Texas, Austin, Texas
	Member, Phi Beta Kappa

August 2007

Short Title: Sound and Complete Decision Procedures   Delaware, M.S. 2007