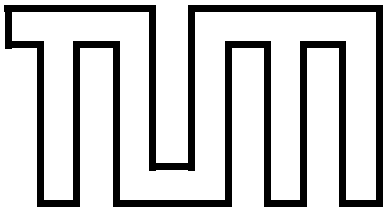
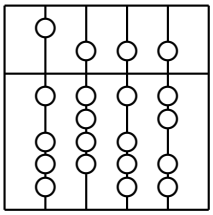




SAT-based Finite Model Generation for Higher-Order Logic

Tjark Weber

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

SAT-based Finite Model Generation for Higher-Order Logic

Tjark Weber

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. F. J. Esparza Estaun
Prüfer der Dissertation:	1. Univ.-Prof. T. Nipkow, Ph. D.
	2. Univ.-Prof. Dr. H. Veith, Technische Universität Darmstadt

Die Dissertation wurde am 30. April 2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30. September 2008 angenommen.

Kurzfassung

Diese Arbeit präsentiert zwei Erweiterungen des Theorembeweisers Isabelle/HOL, einem auf Logik höherer Stufe basierenden System.

Der Hauptbeitrag ist ein Modellgenerator für höherstufige Logik, der seine Eingabeformel in Aussagenlogik übersetzt, so dass ein herkömmlicher SAT-Solver für die eigentliche Modellsuche verwendet werden kann. Die Korrektheit der Übersetzung wird gezeigt. Der Modellgenerator ist in das Isabelle-System integriert worden, unterstützt verschiedene definitorische Mechanismen, wie sie in Isabelle/HOL zur Verfügung stehen, und ist auf mehrere Fallstudien angewandt worden.

Darüber hinaus sind SAT-Solver beweisgenerierend mit Isabelle integriert worden: Aussagenlogische Tautologien können von einem SAT-Solver bewiesen werden, und der von dem Solver gefundene Resolutionsbeweis wird von Isabelle verifiziert. Eine günstige Repräsentation des Problems erlaubt die Verifikation von Beweisen mit mehreren Millionen Resolutionsschritten.

Abstract

This thesis presents two extensions to the theorem prover Isabelle/HOL, a logical framework based on higher-order logic.

The main contribution is a model generator for higher-order logic that proceeds by translating the input formula to propositional logic, so that a standard SAT solver can be employed for the actual model search. The translation is proved correct. The model generator has been integrated with the Isabelle system, extended to support some of the definitional mechanisms provided by Isabelle/HOL, and applied to various case studies.

Moreover, SAT solvers have been integrated with Isabelle in a proof-producing fashion: propositional tautologies can be proved by a SAT solver, and the resolution proof found by the solver is verified by Isabelle. An adequate representation of the problem allows to verify proofs with millions of resolution steps.

Acknowledgements

I would like to thank Tobias Nipkow, my supervisor, for his invaluable support, advice, and patience. The current and former members of his research group have contributed to a fruitful work environment: Clemens Ballarin, Gertrud Bauer, Stefan Berghofer, Amine Chaieb, Florian Haftmann, Gerwin Klein, Farhad Mehta, Steven Obua, Norbert Schirmer, Sebastian Skalberg, Martin Strecker, Markus Wenzel, and Martin Wildmoser. I am particularly indebted to Lars Ebert and Marco Helmers for proof-reading my thesis, to Michael Fortleff for providing shelter, and to Nora for distracting me. I thank Helmut Veith for acting as a referee.

My thanks also go to Helmut Schwichtenberg, speaker of the "Graduiertenkolleg Logik in der Informatik", which provided both financial and intellectual support, and to the other members of the Graduiertenkolleg—in particular to Andreas Abel, Klaus Aehlig, Steffen Jost, and Ralph Matthes for inspiring discussions and more.

Many other people have influenced this thesis in one way or another. Among them are Reinhold Letz, Gernot Stenz, Jan Jürjens, and Manfred Broy from Technische Universität München, Martin Hofmann from Ludwig-Maximilians-Universität München, Hasan Amjad and Larry Paulson from the University of Cambridge, Sharad Malik and Zhaohui Fu from Princeton University, Pascal Fontaine, Stephan Merz, and Alwen Tiu from INRIA Lorraine, John Harrison from Intel Corporation, John Matthews from Galois, Inc., David Aspinall from the University of Edinburgh, Annabelle McIver from Macquarie University, and Moshe Vardi from Rice University.

Finally I would like to thank everyone who has played a part in making the past years in Munich a pleasant and successful time for me. Friends and family have been a steady source of encouragement.

Contents

All men by nature desire to know.
Aristotle, 350 B.C.

Chapter 1

Introduction

1.1 Motivation

Interactive theorem proving is about identifying false conjectures almost as much as it is about finding proofs for theorems. False conjectures may occur for a number of reasons. They can be caused by a trivial typographical error in a formalization, but also by a possibly subtle flaw in the design of a complex system. During interactive proof, false conjectures may be produced by proof strategies that can potentially render a goal unprovable, e.g. by dropping an essential premise.

False conjectures can be seen as intermediate stages of a validation. A true conjecture is often obtained from a sequence of false conjectures and their disproofs. Once identified, it is usually clear how a false conjecture needs to be fixed. Determining that a conjecture is false can however be a time-consuming process. Interactive theorem provers have been enhanced with numerous automatic proof procedures for various application domains over the past years, but failure of an automatic proof attempt is far from sufficient to conclude that a statement is false. It may just be that an additional lemma needs to be proved, or that an induction hypothesis needs to be generalized. The user typically gets little useful information about these reasons, and it may take hours (sometimes days even) of failed proof attempts before he realizes that the conjecture he is trying to prove was false to begin with. In such cases an automatic tool that can refute non-theorems could be highly useful.

One particularly illuminative way to identify a false conjecture is by providing a counterexample. More specifically, when the conjecture is a logical formula, a (counter-)example is given by a (counter-)model. A finite model generator is an automatic tool that attempts to find a finite model which satisfies (or, equivalently, refutes) a given formula. Finite model generation is an instance of the constraint satisfaction problem, with—apart from the generation of counterexamples—numerous applications in artificial intelligence [92], operations research [34], and finite mathematics [102, 175].

This thesis presents (in Chapters 2 and 3) a finite model generator for higher-order logic. A translation to propositional logic is employed and proved correct, so that a standard SAT solver can be used to search for a satisfying model. Such a model constitutes a counterexample to a false conjecture, which can be displayed to the user together with a suitable diagnostic message. This thesis is among the first to consider finite model generation in the context of interactive theorem proving; see Section 1.3 for a discussion of related work.

Interactive theorem provers can benefit from efficient SAT solvers in a second way. Many problems that occur in hardware or software verification can be encoded in propositional logic. The performance of an interactive theorem prover on propositional formulae can therefore be of great practical significance, and the use of SAT solvers for model generation naturally raises the question if they can be used for finding proofs as well. In Chapter 5 we give a positive answer by considering proof-generating SAT solvers, which output either a satisfying assignment for a propositional formula, or a resolution-style proof of unsatisfiability. Here the main challenge is to provide an efficient solution within the LCF-style framework that is underlying Isabelle [134], our theorem prover of choice. This framework ensures soundness of the prover by requiring that theorems are generated via a fixed set of simple functions only, which correspond to the inference rules of the prover's logic.

The next section (1.2) presents the contributions of this thesis. Section 1.3 discusses related work, and Section 1.4 gives some facts about the Isabelle system that are worth knowing. Section 1.5 contains a brief overview of the remaining chapters.

1.2 Contributions

The primary contribution of this thesis is a finite model generator for higher-order logic, integrated with the Isabelle system.

- This thesis presents in detail a translation from higher-order logic to propositional logic such that the resulting propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size. A proof of the translation's correctness is given.
- Based on the above translation, a finite model generator for higher-order logic has been implemented in Standard ML [115] and tightly integrated with Isabelle. The model generator supports datatypes, recursive functions, type classes, records, and other features of Isabelle/HOL [125].
- The model generator has been applied to several case studies which, despite the discouraging theoretical complexity of the underlying algorithm, demonstrate its practical utility. For one of these case studies, an abstract model of probabilistic programs was developed that is susceptible to counterexample search via finite model generation.

The second major contribution is an LCF-style [61] integration of state-of-the-art SAT solvers with Isabelle/HOL. Propositional tautologies (or instances thereof) are proved by the SAT solver, and the resolution proof found by the solver is verified by Isabelle.

- The proof found by the SAT solver is translated into the inference rules provided by Isabelle's kernel. This is, to our knowledge, the first LCF-style integration of highly efficient SAT solvers with an interactive theorem prover.

- The SAT solver approach dramatically outperforms the automatic procedures that were previously available for propositional logic in Isabelle/HOL. It allows many formulae that were previously out of the scope of built-in tactics to be proved—or refuted—automatically, often within seconds.

- A highly optimized implementation is presented that allows proof reconstruction for propositional logic to scale quite well even to large SAT problems and proofs with millions of resolution steps. Our optimization techniques are applicable to other LCF-style theorem provers apart from Isabelle.

- A prototype implementation for persistent proofs is discussed. Proof objects are stored on disk and re-imported into the theorem prover later. This makes the verification of proof scripts that rely on external tools (such as a SAT solver, or a first-order prover) possible on systems where either the external tool or the proof object is available.

Chapter 6 contains a more detailed discussion of the contributions.

1.3 Related Work

The model generator presented in this thesis was originally inspired by the Alloy Analyzer [82], which is developed by the Software Design Group (headed by Daniel Jackson) at MIT. In contrast to the work presented here however, the Alloy Analyzer is based on first-order logic, uses its own input language (incidentally called Alloy), and—unlike the Isabelle system—has no support for theorem proving.

Various other finite model generators exist, although not many for higher-order logic. SAT solvers, which are still a very active research area (resulting in substantial performance improvements over the past years, see e.g. [178]), can be seen as model generators for propositional logic. Finite model generators for first-order logic are typically classified into MACE-style generators (named after McCune's tool MACE [103, 104]), which use a translation to propositional logic, and into SEM-style generators (named after Zhang's and Zhang's tool SEM [177]), which perform the model search directly on a set of first-order clauses. Paradox [41] and Kodkod [159] (the model generator that is used in the most recent version of the Alloy Analyzer) are MACE-style generators, while FINDER [149] and FMSET [21] are SEM-style generators, to name only a few. The model generator presented in this thesis can be classified as a MACE-style generator (albeit for higher-order logic).

Most existing model generators are stand-alone tools that are not in any way integrated with theorem provers. McCune's automated theorem prover Otter [105] and its successor Prover9 [106] are among the few systems that search for a proof and a countermodel. Both Otter and Prover9 are first-order logic provers; they employ the MACE generator for model search. Based on Otter and MACE, Colton and Pease have developed the TM system [43] for repairing non-theorems, which automatically alters false first-order conjectures to provable statements that are (in a certain sense) related to the conjecture. Earlier work that argues for an integration of first-order finite model generation and automated theorem proving includes [150], which considers a combination of Otter and FINDER. More recently, the Paradox model generator has been augmented by an automated first-order theorem prover called

Equinox [39]. We observe that theorem proving and model generation are increasingly seen as complementary activities.

Kimba [92], by Konrad, is a model generator for a linguistically motivated fragment of higher-order logic, but with a slightly special syntax and semantics in order to avoid performance issues. MONA [89] implements an automaton-based decision procedure for the weak monadic second-order theory of one successor (WS1S) that is able to provide counterexamples for unprovable formulae. MONA has been integrated with an earlier version of Isabelle as a trusted oracle [17]. In [12], an abstraction from first-order logic to monadic second order logic is proposed that enables MONA to identify false first-order conjectures. Model checkers, e.g. SPIN [72] or UPPAAL [19], can typically provide counterexamples (in the form of execution traces) for unprovable temporal logic formulae, where the model however is fixed.

Many other techniques exist to identify false conjectures. Tableau calculi are among the most popular proof procedures for first-order and modal logics [66]. They frequently allow a countermodel to be obtained immediately from a failed proof attempt. There are connections between finite model generation and testing: Kurshid and Marinov use model generation to obtain test cases for Java programs [88]. On the other hand, Berghofer has adapted QuickCheck [40], a tool for random testing of Haskell programs, for Isabelle [24]. The Isabelle *quickcheck* tool instantiates free variables in a conjecture with random integer values. If the resulting ground term evaluates to false, a counterexample is found. Support for functions has been added to *quickcheck* only recently however, and input formulae must belong to an executable fragment of higher-order logic. Brucker and Wolff use Isabelle/HOL to generate test cases for specification-based unit tests [29, 30]. Steel et al. use a method called *proof by consistency* to find counterexamples to inductive conjectures [152], with an emphasis on security protocols. [90] contains a survey of techniques for the generation of infinite models.

Perhaps most closely related to the integration of proof-producing SAT solvers is John Harrison's LCF-style integration of Stålmarck's algorithm and BDDs into HOL Light and Hol90 respectively [68, 69]. Harrison found that doing BDD operations inside HOL performed about 100 times worse (after several optimizations) than a C implementation.

Further afield, the integration of automated first-order provers with HOL provers has been explored by Joe Hurd [74, 75], Jia Meng [111], and Lawrence Paulson [112, 113]. Proofs found by the automated system are either verified by the interactive prover immediately [74], or translated into a proof script that can be executed later [112].

A more extensive overview of work related to the integration of proof-producing SAT solvers is given in Section 5.2.

1.4 Isabelle

Isabelle [134] is a popular interactive theorem prover (being developed primarily by Tobias Nipkow at Technische Universität München, and by Larry Paulson at Cambridge University), whose (meta) logic is based on the simply typed λ-calculus. Isabelle is generic, in the sense that different object logics—e.g. first-order logic, modal and linear logics, and Zermelo-Fraenkel set theory—can be defined on top of the meta logic. This thesis mostly considers Isabelle/HOL [125], an incarnation of Isabelle for higher-order logic, which is currently the best-developed object logic. Isabelle/HOL provides the user with convenient means to define

datatypes, recursive functions, axiomatic type classes, sets and records, and more. This makes Isabelle/HOL a very expressive specification language. Syntax and semantics of the logic are presented in detail in Chapter 2.

With a few notable exceptions, the Isabelle/HOL notation follows standard mathematical conventions. Application is written without parentheses, e.g. $f(x)$ is instead written $f\,x$. Application is left-associative: $f\,x\,y$ is the same as $(f\,x)\,y$. Set comprehension is written with a dot "." instead of the more common vertical dash "|": $\{x.\,P\,x\}$ denotes the set of all x that satisfy P. In this thesis, we use mathematical notation in (informal) definitions and proofs, while Isabelle notation is used only in terms and proofs that are machine-checked by the theorem prover. Basic familiarity with ZFC set theory [84] is assumed.

Isabelle is written in Standard ML [115] (SML for short) and can be executed on a number of different SML implementations/compilers. SML is a functional programming language with eager evaluation and some imperative features, e.g. references. It supports higher-order functions and an advanced module system. Historically, SML evolved from the ML programming language (ML stands for metalanguage), which was used in Robin Milner's LCF theorem prover at the University of Edinburgh in the late 1970s [61]. Nowadays the term "LCF-style" is used to designate theorem provers that allow new theorems to be generated via a fixed set of simple functions only. Each function corresponds to an inference rule of the underlying logic. Isabelle is such an LCF-style prover; its soundness therefore only depends on a relatively small *kernel*, and cannot be compromised by programming errors in advanced (and possibly complicated) proof strategies. This restriction motivates the work presented in Chapter 5.

Isabelle supports different styles of writing formal proofs. They can be given as *tactic* scripts, or they can be written in a human-readable (yet fully formal) proof language called *Isar* [171]. Variants of tactic-style proof development are currently found in most interactive theorem provers. Conjectures stated by the user become *proof goals*. Tactics are then applied interactively in order to transform, simplify, and ultimately solve the proof goal. Tactics can implement simple natural deduction rules or powerful decision procedures, e.g. for Presburger arithmetic [36]. Application of a tactic may spawn several new *subgoals*, which then need to be solved as well. Isabelle's tactics are implemented in SML, but this is irrelevant to the average Isabelle user, who usually works with the collection of provided tactics only, and does not need to implement his own.

Since the effect of powerful tactics on the proof state is often hard to predict, tactic scripts are essentially incomprehensible without the theorem prover at hand. The Isar proof language remedies this disadvantage of tactic-style proofs by providing a language that is closer to mathematical textbook reasoning. Well-written Isar proofs can be followed by a human reader, independently of the theorem prover. The focus of this thesis however is not a machine-checked formalization of some theorem in Isabelle, but an extension of the theorem prover itself, and we are concerned with disproving more than with proving. Therefore this thesis contains only some minor Isabelle proofs, and we will not present either proof style in more detail. The interested reader is referred to [123] and [125]. The former contains a tutorial introduction to Isar, while the latter is an extensive overview of Isabelle/HOL, with various examples of tactic-style proofs.

1.5 Overview

The rest of this thesis is structured into five more chapters as follows:

Chapter 2 presents a translation from higher-order logic (on top of the simply typed λ-calculus) to propositional logic, such that the resulting propositional formula is satisfiable iff the HOL formula has a model of a given finite size. A standard SAT solver can then be used to search for a satisfying assignment, and such an assignment can be transformed back into a model for the HOL formula. The algorithm has been implemented in Isabelle/HOL, where it is used to automatically generate countermodels for non-theorems.

Chapter 3 discusses how the translation to propositional logic can be augmented to cover various extensions that the actual Isabelle/HOL system offers on top of the basic HOL logic, mostly to improve usability. Among them are datatypes and recursive functions, axiomatic type classes, set types and extensible records. We also discuss how the translation can be improved to generate smaller propositional formulae.

Chapter 4 contains a presentation of three case studies. We have applied Isabelle's finite model generation techniques to obtain a correctness proof for a security protocol, counterexamples to conjectures about probabilistic programs, and a Sudoku solver.

Chapter 5 describes the integration of zChaff and MiniSat, currently two leading SAT solvers, with Isabelle/HOL. Both SAT solvers generate resolution-style proofs for (instances of) propositional tautologies. These proofs are verified by the theorem prover. The presented approach significantly improves Isabelle's performance on propositional problems.

Chapter 6 summarizes the results presented in this thesis, and gives directions for possible future work.

It is undesirable to believe a proposition when there is no ground whatsoever for supposing it is true.
Bertrand Russell, 1872–1970.

Chapter 2

Finite Model Generation

A translation from higher-order logic (on top of the simply typed λ-calculus) to propositional logic is presented, such that the resulting propositional formula is satisfiable iff the HOL formula has a model of a given finite size. A standard SAT solver can then be used to search for a satisfying assignment, and such an assignment can be transformed back into a model for the HOL formula. The algorithm has been implemented in Isabelle/HOL, where it is used to automatically generate countermodels for non-theorems.

2.1 Introduction

This chapter presents a translation from higher-order logic to propositional logic (quantifier-free Boolean formulae) such that the propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size, i.e. involving no more than a given number of individuals. A standard SAT solver can then be used to search for a satisfying assignment, and if such an assignment is found, it can easily be transformed back into a model for the HOL formula.

An algorithm that uses this translation to generate (counter-)models for HOL formulae has been implemented in Isabelle/HOL. This algorithm is not a (semi-)decision procedure: if a formula does not have a model of a given size, it may still have larger or infinite models. The algorithm's applicability is also limited by its complexity, which is non-elementary for higher-order logic. Nevertheless, formulae that occur in practice often have small models, and the usefulness of a similar approach has been confirmed e.g. in [81].

Section 2.2 introduces the syntax and semantics of the logic considered in this chapter, a version of higher-order logic on top of the simply typed λ-calculus. The translation to propositional

logic is described and proved correct in Section 2.3, while the remaining details of the model generation algorithm are explained in Section 2.4. We conclude with some final remarks in Section 2.5.

2.2 Higher-Order Logic

The translation presented in this chapter can handle the logic that is underlying the HOL [64] and Isabelle/HOL theorem provers. The logic is originally based on Church's simple theory of types [38]. In this section we present the syntax and a set-theoretic semantics of the relevant fragment. A complete account of the HOL logic, including a proof system, can be found e.g. in [63].

2.2.1 Types

We distinguish types and terms, intended to denote certain sets and elements of sets respectively. The definition of types is relative to a given type structure.

Definition 2.1 (Type Structure)**.** A *type structure* is a triple $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$, where TyVars is a set of *type variables*, TyNames is a disjoint set of *type constructors*, and $\mathrm{TyArity}\colon \mathrm{TyNames} \to \mathbb{N}$ gives each type constructor's *arity*.

We use lowercase greek letters, e.g. α, β, ..., to denote type variables.

Definition 2.2 (HOL Type)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a type structure. The set Types_Ω of *types (over Ω)* is the smallest set such that

1. $\mathrm{TyVars} \subseteq \mathrm{Types}_\Omega$, and
2. if $c \in \mathrm{TyNames}$, $\mathrm{TyArity}(c) = n$, and $\sigma_i \in \mathrm{Types}_\Omega$ for all $1 \leq i \leq n$, then $(\sigma_1, \ldots, \sigma_n)c \in \mathrm{Types}_\Omega$. In case $\mathrm{TyArity}(c) = 0$, we write c instead of $()c$.

The sets of type variables and type constructors, respectively, that occur in a type are defined by straightforward structural induction. We distinguish type constructors with different arguments.

Definition 2.3 (Type Variables in a Type)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a type structure, and let $\sigma \in \mathrm{Types}_\Omega$. The set $\mathrm{TyVars}(\sigma)$ of *type variables in σ* is defined as follows:

1. If $\sigma \in \mathrm{TyVars}$, then $\mathrm{TyVars}(\sigma) := \{\sigma\}$.
2. If $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames}$, then $\mathrm{TyVars}(\sigma) := \bigcup_{i=1}^{n} \mathrm{TyVars}(\sigma_i)$.

Definition 2.4 (Type Constructors in a Type)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a type structure, and let $\sigma \in \mathrm{Types}_\Omega$. The set $\mathrm{TyNames}(\sigma)$ of *type constructors in σ* is defined as follows:

1. If $\sigma \in \mathrm{TyVars}$, then $\mathrm{TyNames}(\sigma) := \emptyset$.

2. If $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames}$, then $\mathrm{TyNames}(\sigma) := \{\sigma\} \cup \bigcup_{i=1}^{n} \mathrm{TyNames}(\sigma_i)$.

Remark 2.5. Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a type structure, and let $\sigma \in \mathrm{Types}_\Omega$. $\mathrm{TyVars}(\sigma)$ is a finite subset of TyVars. Likewise, $\mathrm{TyNames}(\sigma)$ is a finite subset of Types_Ω.

Proof. By structural induction on σ. □

To define the semantics of types, we follow [63] and fix a set of sets $\mathcal{U}$, our *set-theoretic universe*, which is assumed to have the following properties.

Inhab Each element of $\mathcal{U}$ is a non-empty set.

Sub If $X \in \mathcal{U}$ and $\emptyset \neq Y \subseteq X$, then $Y \in \mathcal{U}$.

Prod If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then the cartesian product $X \times Y$ is in $\mathcal{U}$.

Pow If $X \in \mathcal{U}$, then $\mathcal{P}(X) = \{Y \mid Y \subseteq X\} \in \mathcal{U}$.

Infty $\mathcal{U}$ contains a distinguished infinite set I.

Choice There is a distinguished element $\mathrm{ch} \in \Pi_{X \in \mathcal{U}} X$.

One can show the existence of such a universe $\mathcal{U}$ from the axioms of Zermelo-Fraenkel set theory together with the Axiom of Choice (ZFC). Two easily provable consequences of the above requirements are important.

Lemma 2.6. *$\mathcal{U}$ contains a two-element set.*

Proof. The infinite set $I \in \mathcal{U}$ has a two-element subset, which is in $\mathcal{U}$ because of **Sub**. □

We distinguish a two-element set $\mathbb{B} = \{\top, \bot\} \in \mathcal{U}$.

Lemma 2.7. *If $X \in \mathcal{U}$ and $Y \in U$, then $X \to Y$, i.e. the set of all total functions from X to Y, is in U.*

Proof. Let $X, Y \in \mathcal{U}$. In set theory functions are identified with their graphs, which are certain sets of ordered pairs. Therefore $X \to Y$ is a subset of $\mathcal{P}(X \times Y)$, which is in $\mathcal{U}$ due to **Prod** and **Pow**. Furthermore $X \to Y$ is non-empty since Y is non-empty because of **Inhab**. Hence $X \to Y \in \mathcal{U}$ by virtue of **Sub**. □

We are now ready to define the semantics of types. Type variables denote arbitrary non-empty sets, which are given by a type environment. The meaning of type constructors is given by a type model.

Definition 2.8 (Type Environment)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a type structure. A *type environment for* Ω is a function $E \colon \mathrm{TyVars} \to \mathcal{U}$.

Definition 2.9 (Type Model)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a type structure. A *type model M of* Ω assigns to each type constructor $c \in \mathrm{TyNames}$ a function $M(c) \colon \mathcal{U}^{\mathrm{TyArity}(c)} \to \mathcal{U}$. In case $\mathrm{TyArity}(c) = 0$, we identify $M(c) \colon \mathcal{U}^0 \to \mathcal{U}$ with $M(c)() \in \mathcal{U}$.

Definition 2.10 (Semantics of Types)**.** Let $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ be a type structure. The *meaning* $[\![\sigma]\!]_{E,M}$ *of a type* $\sigma \in \text{Types}_\Omega$ *(wrt. a type environment* E *and a type model* M*)* is defined as follows:

1. If $\sigma \in \text{TyVars}$, then $[\![\sigma]\!]_{E,M} := E(\sigma)$.
2. If $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \text{TyNames}$, then $[\![\sigma]\!]_{E,M} := M(c)([\![\sigma_1]\!]_{E,M}, \ldots, [\![\sigma_n]\!]_{E,M})$.

Remark 2.11. $[\![\sigma]\!]_{E,M}$ (for σ a type) is an element of $\mathcal{U}$, i.e. $[\![\,\cdot\,]\!]_{E,M} \colon \text{Types}_\Omega \to \mathcal{U}$.

Proof. By structural induction on σ. □

The meaning of a type only depends on the meaning of those type variables and type constructors that actually occur in the type. For $f \colon X \to Y$ a function and $Z \subseteq X$, we write $f|_Z$ for the restriction of f to Z, i.e. for the function with domain Z that sends $x \in Z$ to $f(x) \in Y$.

Lemma 2.12. *Let* $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ *be a type structure, let* E, $E' \colon \text{TyVars} \to \mathcal{U}$ *be two type environments for* Ω*, and let* M, M' *be two type models of* Ω*. Let* $\sigma \in \text{Types}_\Omega$*. Suppose* $E|_{\text{TyVars}(\sigma)} = E'|_{\text{TyVars}(\sigma)}$*, and furthermore* $M(c)([\![\sigma_1]\!]_{E,M}, \ldots, [\![\sigma_n]\!]_{E,M}) = M'(c)([\![\sigma_1]\!]_{E,M}, \ldots, [\![\sigma_n]\!]_{E,M})$ *for all* $(\sigma_1, \ldots, \sigma_n)c \in \text{TyNames}(\sigma)$*. Then*

$$[\![\sigma]\!]_{E,M} = [\![\sigma]\!]_{E',M'}.$$

Proof. By structural induction on σ. For $\sigma \in \text{TyVars}$, we have

$$[\![\sigma]\!]_{E,M} \overset{2.10}{=} E(\sigma) = E'(\sigma) \overset{2.10}{=} [\![\sigma]\!]_{E',M'}$$

since $\sigma \in \text{TyVars}(\sigma)$.

If $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \text{TyNames}$ (hence $(\sigma_1, \ldots, \sigma_n)c \in \text{TyNames}(\sigma)$), then applying the induction hypothesis to $\sigma_1, \ldots, \sigma_n$ yields

$$\begin{aligned}
[\![\sigma]\!]_{E,M} &\overset{2.10}{=} M(c)([\![\sigma_1]\!]_{E,M}, \ldots, [\![\sigma_n]\!]_{E,M}) \\
&= M'(c)([\![\sigma_1]\!]_{E,M}, \ldots, [\![\sigma_n]\!]_{E,M}) \\
&\overset{\text{IH}}{=} M'(c)([\![\sigma_1]\!]_{E',M'}, \ldots, [\![\sigma_n]\!]_{E',M'}) \\
&\overset{2.10}{=} [\![\sigma]\!]_{E',M'}.
\end{aligned}$$

□

We call type structures that contain two distinguished type constructors, namely bool and $\to$, standard. We say that a type model is standard iff these type constructors are interpreted as the two-element set $\{\top, \bot\}$ and as the function space constructor, respectively.

Definition 2.13 (Standard Type Structure)**.** A type structure $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ is *standard* iff $\{\mathsf{bool}, \to\} \subseteq \text{TyNames}$, $\text{TyArity}(\mathsf{bool}) = 0$, and $\text{TyArity}(\to) = 2$.

Definition 2.14 (Standard Type Model)**.** A type model M of a standard type structure is *standard* iff $M(\mathsf{bool}) = \mathbb{B}$ and $M(\to)(X, Y) = X \to Y$ (the set of all total functions from X to Y) for all X, $Y \in \mathcal{U}$.

From now on we only consider standard type structures and standard type models, where the meaning of bool and $\to$ is fixed. We use infix notation for $\to$, i.e. we write $\sigma_1 \to \sigma_2$ instead of $(\sigma_1, \sigma_2)\to$. As usual, $\to$ associates to the right: $\sigma_1 \to \sigma_2 \to \sigma_3$ is short for $\sigma_1 \to (\sigma_2 \to \sigma_3)$.

In the literature, standard HOL type structures are sometimes required to contain another nullary type constructor, inf, whose intended interpretation is an infinite set [126]. Note that we do not require such a type constructor here, since it would immediately disallow to find finite models. We still require the set-theoretic universe $\mathcal{U}$ to contain an infinite set, so merely the (type) syntax of the logic is affected by this deviation, while model-theoretic issues are not. A possible approach to extending finite model generation to formulae with infinite types is discussed in Chapter 3.

2.2.2 Terms

Just like the definition of types is relative to a given type structure, the definition of terms is relative to a given (term) signature.

Definition 2.15 (Signature)**.** A *signature (over a type structure Ω)* is a triple $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$, where Vars is a set of *variables*, Names is a disjoint set of *constants*, and $\text{Typ}\colon \text{Names} \to \text{Types}_\Omega$ gives the *type* of each constant.

Terms are explicitly annotated with their type. A term t_σ of type σ is either an (explicitly typed) variable, a constant, an application, or a λ-abstraction. The actual type of a constant only needs to be an instance of the type given by the signature, so we need to define type instances before we can define terms.

Definition 2.16 (Type Substitution)**.** A *type substitution for a type structure* $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ is a function $\Theta\colon \text{TyVars} \to \text{Types}_\Omega$.

The *application of a type substitution* Θ *to a type* $\sigma \in \text{Types}_\Omega$, written $\sigma\,\Theta$, is defined by structural induction on σ:

1. If $\sigma \in \text{TyVars}$, then $\sigma\,\Theta := \Theta(\sigma)$.
2. If $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \text{TyNames}$, then $\sigma\,\Theta := (\sigma_1\,\Theta, \ldots, \sigma_n\Theta)c$.

Remark 2.17. For $\sigma \in \text{Types}_\Omega$ and Θ a type substitution for Ω, $\sigma\,\Theta$ is again in Types_Ω. In other words, $\cdot\,\Theta\colon \text{Types}_\Omega \to \text{Types}_\Omega$.

Proof. By structural induction on σ. □

Definition 2.18 (Type Instance)**.** Let Ω be a type structure. For $\sigma, \sigma' \in \text{Types}_\Omega$, we say that *$\sigma$ is an instance of σ'* iff $\sigma = \sigma'\,\Theta$ for some type substitution Θ.

Definition 2.19 (HOL Term)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. The set Terms_Σ of *terms over* Σ is the smallest set such that

1. if $x \in \text{Vars}$ and $\sigma \in \text{Types}_\Omega$, then $x_\sigma \in \text{Terms}_\Sigma$,
2. if $c \in \text{Names}$, $\text{Typ}(c) = \sigma$ and $\sigma' \in \text{Types}_\Omega$ is an instance of σ, then $c_{\sigma'} \in \text{Terms}_\Sigma$,

3. if $t_{\sigma'\to\sigma} \in \text{Terms}_\Sigma$ and $t'_{\sigma'} \in \text{Terms}_\Sigma$, then $(t_{\sigma'\to\sigma}\, t'_{\sigma'})_\sigma \in \text{Terms}_\Sigma$, and
4. if $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t_{\sigma_2} \in \text{Terms}_\Sigma$, then $(\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1\to\sigma_2} \in \text{Terms}_\Sigma$.

Terms of type bool are called *formulae*. We frequently omit the type annotation of terms when it can be deduced from the context.

The sets of a term's (explicitly typed) free variables and its (explicitly typed) constants, respectively, are defined as usual, by structural induction on the term.

Definition 2.20 (Free Variables in a Term)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let $t_\sigma \in \text{Terms}_\Sigma$. The set $\text{FreeVars}(t_\sigma)$ of *free variables in* t_σ is defined as follows:

1. If $t \in \text{Vars}$, then $\text{FreeVars}(t_\sigma) := \{t_\sigma\}$.
2. If $t \in \text{Names}$, then $\text{FreeVars}(t_\sigma) := \emptyset$.
3. If $t_\sigma = (t^1_{\sigma'\to\sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma'\to\sigma}$, $t^2_{\sigma'} \in \text{Terms}_\Sigma$, then $\text{FreeVars}(t_\sigma) := \text{FreeVars}(t^1_{\sigma'\to\sigma}) \cup \text{FreeVars}(t^2_{\sigma'})$.
4. If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1\to\sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, then $\text{FreeVars}(t_\sigma) := \text{FreeVars}(t'_{\sigma_2}) \setminus \{x_{\sigma_1}\}$.

Definition 2.21 (Constants in a Term)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let $t_\sigma \in \text{Terms}_\Sigma$. The set $\text{Names}(t_\sigma)$ of *constants in* t_σ is defined as follows:

1. If $t \in \text{Vars}$, then $\text{Names}(t_\sigma) := \emptyset$.
2. If $t \in \text{Names}$, then $\text{Names}(t_\sigma) := \{t_\sigma\}$.
3. If $t_\sigma = (t^1_{\sigma'\to\sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma'\to\sigma}$, $t^2_{\sigma'} \in \text{Terms}_\Sigma$, then $\text{Names}(t_\sigma) := \text{Names}(t^1_{\sigma'\to\sigma}) \cup \text{Names}(t^2_{\sigma'})$.
4. If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1\to\sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, then $\text{Names}(t_\sigma) := \text{Names}(t'_{\sigma_2})$.

Remark 2.22. Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over some standard type structure, and let $t_\sigma \in \text{Terms}_\Sigma$. $\text{FreeVars}(t_\sigma)$ and $\text{Names}(t_\sigma)$ are finite subsets of Terms_Σ.

Proof. By structural induction on t_σ. □

Having defined the syntax of terms, we now come to the definition of their semantics. The analogue of a type environment at the term level is a variable assignment, and type models correspond to term models.

Definition 2.23 (Variable Assignment)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a type structure Ω. A *variable assignment* $\mathcal{A}$ *(for* Σ*)* assigns to each variable $x \in \text{Vars}$ a function $\mathcal{A}(x) \colon \mathcal{U} \to \bigcup\mathcal{U}$ which satisfies $\mathcal{A}(x)(Y) \in Y$ for every $Y \in \mathcal{U}$.

Definition 2.24 (Term Model)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a type structure Ω. A *term model* $\mathcal{M}$ *(for* Σ*)* assigns to each constant $c \in \text{Names}$ a function $\mathcal{M}(c) \colon \mathcal{U} \to \bigcup \mathcal{U}$ which satisfies $\mathcal{M}(c)(Y) \in Y$ for every $Y \in \mathcal{U}$.

To shorten notation, we write $\mathcal{A}(x_\sigma)$ for $\mathcal{A}(x)(\llbracket \sigma \rrbracket_{E,M})$, and likewise $\mathcal{M}(c_\sigma)$ for $\mathcal{M}(c)(\llbracket \sigma \rrbracket_{E,M})$, when the type environment E and the type model M are clear from the context.

For $f \colon X \to Y$ a function, $a \in X$ and $b \in Y$, we write $f[a \mapsto b]$ for the function that sends $x \in X$ to b if $x = a$, and to $f(x)$ otherwise. We can now define the semantics of terms. The semantics of variables is given by a variable assignment, and the semantics of constants is given by a (term) model. Term application corresponds to function application, and λ-abstractions denote functions.

Definition 2.25 (Semantics of Terms)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let E be a type environment for Ω, and let M be a standard type model for Ω. Let $\mathcal{A}$ be a variable assignment and $\mathcal{M}$ be a term model for Σ. The *meaning* $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}}$ *of a term* $t_\sigma \in \text{Terms}_\Sigma$ *wrt.* $\mathcal{A}$ *and* $\mathcal{M}$ is defined as follows:

1. If $t \in \text{Vars}$, then $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}} := \mathcal{A}(t_\sigma)$.
2. If $t \in \text{Names}$, then $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}} := \mathcal{M}(t_\sigma)$.
3. If $t_\sigma = (t^1_{\sigma' \to \sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma' \to \sigma}$, $t^2_{\sigma'} \in \text{Terms}_\Sigma$, then $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}} := \llbracket t^1_{\sigma' \to \sigma} \rrbracket_{\mathcal{A},\mathcal{M}}(\llbracket t^2_{\sigma'} \rrbracket_{\mathcal{A},\mathcal{M}})$ (function application).
4. If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1 \to \sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, then $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}}$ is the function that sends each $d \in \llbracket \sigma_1 \rrbracket_{E,M}$ to $\llbracket t'_{\sigma_2} \rrbracket_{\mathcal{A}[x_{\sigma_1} \mapsto d],\mathcal{M}}$.

Remark 2.26. $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}}$ (for t_σ a term) is an element of $\bigcup \mathcal{U}$, i.e. $\llbracket \cdot \rrbracket_{\mathcal{A},\mathcal{M}} \colon \text{Terms}_\Sigma \to \bigcup \mathcal{U}$.

Proof. The claim follows from Lemma 2.27 below, where $\llbracket \sigma \rrbracket_{E,M} \in \mathcal{U}$ due to Remark 2.11. □

More specifically, the meaning of a term is an element of the meaning of the term's type.

Lemma 2.27. *Let* $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ *be a signature over a standard type structure* Ω*. Let* E *be a type environment for* Ω*, and let* M *be a standard type model for* Ω*. Let* $\mathcal{A}$ *be a variable assignment and* $\mathcal{M}$ *be a term model for* Σ*. Then* $\llbracket t_\sigma \rrbracket_{\mathcal{A},\mathcal{M}} \in \llbracket \sigma \rrbracket_{E,M}$ *for any term* $t_\sigma \in \text{Terms}_\Sigma$.

Proof. By structural induction on t_σ. For the two base cases $t \in \text{Vars}$ and $t \in \text{Names}$, the claim follows immediately from Def. 2.23 and Def. 2.24, respectively, together with Remark 2.11.

The two remaining cases are proved using the fact that M is standard, and hence interprets $\sigma_1 \to \sigma_2$ as the full function space from $\llbracket \sigma_1 \rrbracket_{E,M}$ to $\llbracket \sigma_2 \rrbracket_{E,M}$. We note that in the case of a λ-abstraction, the updated assignment $\mathcal{A}[x_{\sigma_1} \mapsto d]$ is again a variable assignment. □

The meaning of a term only depends on the meaning of its free variables and constants. This is the analogue of Lemma 2.12 (which states that the meaning of a type only depends on its type variables and type constructors) for terms.

Lemma 2.28. *Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let E be a type environment for Ω, and let M be a standard type model for Ω. Let $\mathcal{A}$, $\mathcal{A}'$ be two variable assignments and $\mathcal{M}$, $\mathcal{M}'$ be two term models for Σ. Let $t_\sigma \in \text{Terms}_\Sigma$. Suppose $\mathcal{A}|_{\text{FreeVars}(t_\sigma)} = \mathcal{A}'|_{\text{FreeVars}(t_\sigma)}$ and $\mathcal{M}|_{\text{Names}(t_\sigma)} = \mathcal{M}'|_{\text{Names}(t_\sigma)}$. Then*

$$[\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}} = [\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'}.$$

Proof. By structural induction on t_σ. For the two base cases $t \in \text{Vars}$ and $t \in \text{Names}$, the claim follows immediately from $\mathcal{A}|_{\text{FreeVars}(t_\sigma)} = \mathcal{A}'|_{\text{FreeVars}(t_\sigma)}$ and $\mathcal{M}|_{\text{Names}(t_\sigma)} = \mathcal{M}'|_{\text{Names}(t_\sigma)}$, respectively.

If $t_\sigma = (t^1_{\sigma'\to\sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma'\to\sigma}, t^2_{\sigma'} \in \text{Terms}_\Sigma$, the claim follows from the induction hypothesis, applied to $t^1_{\sigma'\to\sigma}$ and $t^2_{\sigma'}$.

If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1\to\sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, the claim follows from the induction hypothesis, applied to t'_{σ_2}. Note that for $d \in [\![\sigma_1]\!]_{E,M}$, both $\mathcal{A}[x_{\sigma_1} \mapsto d]$ and $\mathcal{A}'[x_{\sigma_1} \mapsto d]$ are again variable assignments, they agree on x_{σ_1}, and therefore (since $\text{FreeVars}(t'_{\sigma_2}) \subseteq \text{FreeVars}(t_\sigma) \cup \{x_{\sigma_1}\}$) they agree on $\text{FreeVars}(t'_{\sigma_2})$. □

We require that signatures contain two logical constants, namely $\Longrightarrow$ and $=$, which are interpreted as implication and equality, respectively.

Definition 2.29 (Standard Signature)**.** A signature $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ over a standard type structure Ω is *standard* iff $\{\Longrightarrow, =\} \subseteq \text{Names}$, $\text{Typ}(\Longrightarrow) = \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$, and $\text{Typ}(=) = \alpha \to \alpha \to \mathsf{bool}$ for some type variable α.

Definition 2.30 (Standard Term Model)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let E be a type environment for Ω, and let M be a standard type model for Ω. A term model $\mathcal{M}$ for Σ is *standard* iff

1. $\mathcal{M}(\Longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}})$ is the function that maps $\begin{cases} \top, \top & \text{to } \top \\ \top, \bot & \text{to } \bot \\ \bot, \top & \text{to } \top \\ \bot, \bot & \text{to } \top \end{cases}$, and
2. for every $\sigma \in \text{Types}_\Omega$, $\mathcal{M}(=_{\sigma\to\sigma\to\mathsf{bool}})$ is the function that maps $x, y \in [\![\sigma]\!]_{E,M}$ to $\top$ if $x = y$, and to $\bot$ otherwise.

Both $\Longrightarrow$ and $=$ are usually written in infix notation, with $\Longrightarrow$ associating to the right.

This particular choice of constants is arbitrary. Other logical constants can be defined in terms of the chosen ones, e.g. $\mathsf{True}_\mathsf{bool}$ as $(\lambda x_\mathsf{bool}.\, x_\mathsf{bool}) = (\lambda x_\mathsf{bool}.\, x_\mathsf{bool})$, universal quantification $\forall_{(\alpha\to\mathsf{bool})\to\mathsf{bool}}$ as $\lambda P_{\alpha\to\mathsf{bool}}.\, P_{\alpha\to\mathsf{bool}} = (\lambda x_\alpha.\, \mathsf{True}_\mathsf{bool})$, and $\mathsf{False}_\mathsf{bool}$ as $\forall\,(\lambda x_\mathsf{bool}.\, x_\mathsf{bool})$ [10].

2.2.3 Satisfiability

A HOL formula is satisfiable iff its meaning is $\top$ in some standard model.

Definition 2.31 (HOL Satisfiability)**.** Let Σ be a standard signature over a standard type structure Ω, and let $t_\mathsf{bool} \in \text{Terms}_\Sigma$ be a formula. For E a type environment for Ω, M a

standard type model for Ω, $\mathcal{A}$ a variable assignment and $\mathcal{M}$ a term model for Σ, we say that $\mathcal{A}, \mathcal{M}$ *satisfies* t_{bool}, written $\mathcal{A}, \mathcal{M} \models t_{\mathsf{bool}}$, iff $[\![t_{\mathsf{bool}}]\!]_{\mathcal{A},\mathcal{M}} = \top$.

For E a type environment for Ω and M a standard type model for Ω, we say that t_{bool} *is satisfiable wrt. E and M* iff there exist a variable assignment $\mathcal{A}$ and a standard term model $\mathcal{M}$ for Σ such that $\mathcal{A}, \mathcal{M} \models t_{\mathsf{bool}}$.

We say that t_{bool} *is satisfiable* iff there exist a type environment E for Ω and a standard type model M of Ω such that t_{bool} is satisfiable wrt. E and M.

It is well-known that it is not semi-decidable in general if a HOL formula is satisfiable. Even satisfiability in finite models is not semi-decidable. Consequently, the algorithm presented in Section 2.3 is not a semi-decision algorithm. It is however sound and complete in the following sense: given unbounded space and time, the algorithm will find a finite model for a HOL formula if and only if such a model exists.

2.3 Translation to Propositional Logic

The model generation for a HOL formula t_{bool} proceeds in several steps. The input formula is first translated into a propositional formula that is satisfiable iff t_{bool} has a model of a given size.

2.3.1 Propositional Logic

Let us briefly recall the basic notions of propositional logic. We fix an infinite set $\mathcal{B}$ of *Boolean variables*.

Definition 2.32 (Propositional Formula)**.** The set $\mathbb{P}$ of *propositional formulae (over $\mathcal{B}$)* is the smallest set such that

1. $\mathcal{B} \subseteq \mathbb{P}$,
2. $\mathrm{True} \in \mathbb{P}$, $\mathrm{False} \in \mathbb{P}$,
3. if $\varphi \in \mathbb{P}$, then $(\neg\varphi) \in \mathbb{P}$,
4. if φ, $\psi \in \mathbb{P}$, then $(\varphi \vee \psi) \in \mathbb{P}$ and $(\varphi \wedge \psi) \in \mathbb{P}$.

As a standard convention, $\neg$ binds stronger than $\wedge$, which in turn binds stronger than $\vee$. Using this convention, we frequently omit unnecessary parentheses. The semantics of propositional formulae is defined wrt. a truth assignment.

Definition 2.33 (Truth Assignment)**.** A *truth assignment* A is a function $A\colon \mathcal{B} \to \mathbb{B}$.

Definition 2.34 (Semantics of Propositional Formulae)**.** Let A be a truth assignment. The *meaning $[\![\varphi]\!]_A$ of a propositional formula $\varphi \in \mathbb{P}$ wrt. A* is defined as follows:

1. If $\varphi \in \mathcal{B}$, then $[\![\varphi]\!]_A := A(\varphi)$.

2. $[\![\text{True}]\!]_A := \top$, $[\![\text{False}]\!]_A := \bot$.

3. $[\![\neg\varphi]\!]_A := \begin{cases} \top & \text{if } [\![\varphi]\!]_A = \bot; \\ \bot & \text{otherwise.} \end{cases}$

4. $[\![\varphi \vee \psi]\!]_A := \begin{cases} \top & \text{if } [\![\varphi]\!]_A = \top \text{ or } [\![\psi]\!]_A = \top; \\ \bot & \text{otherwise,} \end{cases}$ and
$[\![\varphi \wedge \psi]\!]_A := \begin{cases} \top & \text{if } [\![\varphi]\!]_A = \top \text{ and } [\![\psi]\!]_A = \top; \\ \bot & \text{otherwise.} \end{cases}$

Remark 2.35. Let A be a truth assignment, and let $\varphi \in \mathbb{P}$. Then $[\![\varphi]\!]_A \in \mathbb{B}$, i.e. $[\![\cdot]\!]_A \colon \mathbb{P} \to \mathbb{B}$.

Proof. By structural induction on φ. □

Definition 2.36 (Propositional Satisfiability)**.** Let A be a truth assignment, and let $\varphi \in \mathbb{P}$ be a propositional formula. *A satisfies* φ, written $A \models \varphi$, iff $[\![\varphi]\!]_A = \top$.

We say that φ *is satisfiable* iff $A \models \varphi$ for some truth assignment A.

2.3.2 Interpretation of Types

Types in the input formula t_{bool} are interpreted as finite, non-empty, mutually disjoint sets. (Disjointness is justified because in HOL, one cannot express that different types contain equal elements: equality is only available for equal types in the first place. Therefore the type algebra can be seen as freely generated.) Let us fix a standard type structure $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ and a standard signature $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ over Ω such that $t_{\mathsf{bool}} \in \text{Terms}_\Sigma$.

We choose a type environment E that only assigns finite sets to type variables, and a standard type model M where each $M(c)$ (for c a type constructor) maps finite sets to finite sets.

Definition 2.37 (Finite Type Environment)**.** Let $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ be a type structure, and let E be a type environment for Ω. We say that E is *finite* iff $E(\alpha)$ is finite for every type variable $\alpha \in \text{TyVars}$.

Definition 2.38 (Finite Type Model)**.** Let $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ be a type structure, and let M be a type model for Ω. We say that M is *finite* iff, for every type constructor $c \in \text{TyNames}$, $M(c)(X_1, \ldots, X_n)$ is finite whenever $X_1, \ldots, X_n$ are finite (where $n = \text{TyArity}(c)$).

Remark 2.39. Let M be a standard type model. Then $M(\mathsf{bool}) = \mathbb{B}$ is finite, and $M(\to)(X, Y) = X \to Y$ is finite if both $X \in \mathcal{U}$ and $Y \in \mathcal{U}$ are finite. (More precisely, $|\mathbb{B}| = 2$, and $|X \to Y| = |Y|^{|X|}$.)

Proof. Immediate, using Def. 2.14. □

Then every type denotes a finite set wrt. E and M.

Lemma 2.40. *Let Ω be a type structure, let E be a finite type environment for Ω, and let M be a finite type model for Ω. Then for every $\sigma \in \text{Types}_\Omega$, $[\![\sigma]\!]_{E,M}$ is finite.*

Proof. By induction on σ. For σ a type variable, the claim follows directly from the fact that E is finite.

In case $\sigma = (\sigma_1, \ldots, \sigma_n)c$, where c is a type constructor, we apply the induction hypothesis to σ_1, ..., σ_n to obtain finiteness of $[\![\sigma_1]\!]_{E,M}$, ..., $[\![\sigma_n]\!]_{E,M}$. The lemma then follows since M is finite. □

Because of Lemma 2.12, it is de facto sufficient to define E and M for those—finitely many—type variables and type constructors, respectively, that occur in the typing of t_{bool}. Having fixed the meaning of relevant type variables and type constructors, we want to find a propositional formula that is (propositionally) satisfiable iff t_{bool} is (HOL-)satisfiable wrt. E and M. But before we describe the translation of HOL formulae to propositional formulae in Section 2.3.3, a few more remarks concerning the interpretation of types are in order.

Ordered Sets

Without loss of generality we require each finite set in the range of E and M to be equipped with a total order, i.e. with an antisymmetric, transitive, and total binary relation. For sets that denote type variables and type constructors other than bool and $\to$, this order is presupposed. For $\mathbb{B}$, we make an arbitrary choice.

Definition 2.41 (Order on $\mathbb{B}$)**.** The *(canonical) order on* $\mathbb{B}$, written $\leq_{\mathbb{B}}$, is given by $\top < \bot$.

Given two totally ordered finite sets X and Y, we use the lexicographic order as a total order on $X \to Y$.

Definition 2.42 (Lexicographic Order)**.** Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be totally ordered finite sets. The *(lexicographic) order on* $X \to Y$, written $\leq_{X \to Y}$, is given by

$$f \leq_{X \to Y} g \quad \text{iff} \quad \exists x \in X.\, (f(x) \leq_Y g(x) \wedge \forall x' <_X x.\, f(x') = g(x')).$$

Remark 2.43. Using finiteness of X, one verifies that $\leq_{X \to Y}$ is in fact a total order on $X \to Y$. The function space is isomorphic to the $|X|$-fold cartesian product of Y.

A totally ordered finite set $X = \{x_1, x_2, \ldots, x_n\}$, where the order on X is given by $x_1 < x_2 < \ldots < x_n$, can be identified with the list $[x_1, x_2, \ldots, x_n]$ of its elements. We say that x_1, x_2, ..., x_n is the first, second, ..., n-th element of X, respectively.

A function $f\colon X \to Y$ can be identified with its graph, i.e. with the set of ordered pairs $\{(x, f(x)) \mid x \in X\}$. A function $f\colon X \to Y$, where $X = [x_1, \ldots, x_n]$ is finite and totally ordered, can be identified with the list $[f(x_1), \ldots, f(x_n)]$ of its values. Since these identifications are crucial in the context of our work, we give a formal definition of lists.

Definition 2.44 (List)**.** Let X be a set of *list elements*. The set List_X of *lists with elements in* X is the smallest set such that

1. $[\,] \in \mathrm{List}_X$, and
2. if $l \in \mathrm{List}_X$ and $x \in X$, then $(x \# l) \in \mathrm{List}_X$.

We write $[x_1, \ldots, x_n]$ for the list $(x_1 \# (\ldots (x_n \# [\,]) \ldots))$.

The List operator is monotonic wrt. the subset relation.

Lemma 2.45. *Let $X \subseteq Y$. Then $\mathrm{List}_X \subseteq \mathrm{List}_Y$.*

Proof. Let $l \in \mathrm{List}_X$. The proof is by structural induction on l. □

Given a totally ordered finite codomain $Y = [y_1, \ldots, y_m]$ and the cardinality $|X|$ of a domain X, we can define an auxiliary function pick that computes the list representation of the function space $X \to Y$, equipped with the lexicographic order. In other words, pick enumerates all functions in $X \to Y$ (where each function is represented as the list of its values) in the order given by Def. 2.42.

Definition 2.46 (pick)**.** Let $Y = [y_1, \ldots, y_m]$, $m \geq 1$. Define

$$\mathrm{pick}(1, [y_1, \ldots, y_m]) := [[y_1], \ldots, [y_m]],$$

and for $n > 1$ define

$$\mathrm{pick}(n, [y_1, \ldots, y_m]) := [y_1 \# f_1, \ldots, y_1 \# f_{m^{n-1}}, \ \ldots, \ y_m \# f_1, \ldots, y_m \# f_{m^{n-1}}],$$

where $[f_1, \ldots, f_{m^{n-1}}] = \mathrm{pick}(n-1, [y_1, \ldots, y_m])$.

Remark 2.47. For n, $m \geq 1$, $\mathrm{pick}(n, [y_1, \ldots, y_m])$ is a list in $\mathrm{List}_{\mathrm{List}_Y}$ of length m^n, and each list element (which is again a list) has length n.

Proof. By induction on n. □

Lemma 2.48. *Let $|X| = n$, $Y = [y_1, \ldots, y_m]$, where n, $m \geq 1$. Then*

$$\mathrm{pick}(n, [y_1, \ldots, y_m]) = X \to Y,$$

where $X \to Y$ is equipped with the lexicographic order.

Proof. By induction on n, using Def. 2.42 and Def. 2.46. □

While the order used on the function space and the definition of pick are of course interdependent, using the lexicographic order was an arbitrary choice. It is merely important that we can enumerate the elements of the function space, based on enumerations for the domain and the codomain.

Isomorphic Types

Type environments and standard type models are determined uniquely up to isomorphism by the *size* of the sets that they assign; the names of individuals are irrelevant.

Definition 2.49 (Isomorphic Type Environments)**.** Let Ω be a type structure. We say that two type environments E, E' for Ω are *isomorphic* iff there exists a bijection $I : \mathcal{U} \to \mathcal{U}$ such that $I(E(\alpha)) = E'(\alpha)$ for every type variable α. I is called an *isomorphism (between E and E')*.

Definition 2.50 (Isomorphic Type Models)**.** Let Ω be a type structure. We say that two type models M, M' of Ω are *isomorphic* iff there exists a bijection $I\colon \mathcal{U} \to \mathcal{U}$ such that $I(M(c)(X_1, \ldots, X_n)) = M'(c)(I(X_1), \ldots, I(X_n))$ for every type constructor c and every X_1, $\ldots$, $X_n \in \mathcal{U}$ (where $n = \mathrm{TyArity}(c)$). I is called an *isomorphism (between M and M').*

The meaning of types wrt. isomorphic type environments and type models is given by the image of their original meaning under the isomorphism.

Lemma 2.51 (Semantics of Isomorphic Types)**.** *Let Ω be a type structure. Let E, E' be two type environments for Ω, and let M, M' be two type models of Ω. Suppose that E, E' and M, M' are isomorphic wrt. the same isomorphism $I\colon \mathcal{U} \to \mathcal{U}$. Then*

$$I([\![\sigma]\!]_{E,M}) = [\![\sigma]\!]_{E',M'}$$

for every type $\sigma \in \mathrm{Types}_\Omega$.

Proof. By structural induction on σ. For σ a type variable, we have

$$I([\![\sigma]\!]_{E,M}) \overset{2.10}{=} I(E(\sigma)) \overset{2.49}{=} E'(\sigma) \overset{2.10}{=} [\![\sigma]\!]_{E',M'}$$

since E and E' are isomorphic.

If $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames}$, we apply the induction hypothesis to σ_1, $\ldots$, σ_n to obtain

$$\begin{aligned} I([\![\sigma]\!]_{E,M}) &\overset{2.10}{=} I(M(c)([\![\sigma_1]\!]_{E,M}, \ldots, [\![\sigma_n]\!]_{E,M})) \\ &\overset{2.50}{=} M'(c)(I([\![\sigma_1]\!]_{E,M}), \ldots, I([\![\sigma_n]\!]_{E,M})) \\ &\overset{\mathrm{IH}}{=} M'(c)([\![\sigma_1]\!]_{E',M'}, \ldots, [\![\sigma_n]\!]_{E',M'}) \\ &\overset{2.10}{=} [\![\sigma]\!]_{E',M'}. \end{aligned}$$

□

This result can be "lifted" to the semantics of terms. Suppose that the isomorphism I operates on elements of sets in $\mathcal{U}$, rather than on sets in $\mathcal{U}$. Then the meaning of terms wrt. an isomorphic variable assignment and term model is given by the image of their original meaning under the isomorphism.

Definition 2.52 (Pointwise Isomorphism)**.** Let Ω be a type structure. Let E, E' be two type environments for Ω (let M, M' be two type models of Ω, respectively). We say that E, E' (M, M' respectively) are *isomorphic wrt. a bijection* $I\colon \bigcup\mathcal{U} \to \bigcup\mathcal{U}$ iff $\hat{I}(X) := \{I(x) \mid x \in X\}$ (for $X \in \mathcal{U}$) defines an isomorphism $\hat{I}\colon \mathcal{U} \to \mathcal{U}$ between E and E' (M and M', respectively). In this case I is called a *pointwise isomorphism* (*between E and E'*, or *between M and M'*).

More generally, any function $f\colon X \to Y$ can be "lifted" to a function $\hat{f}\colon \mathcal{P}(X) \to \mathcal{P}(Y)$ in the obvious way, by defining $\hat{f}(Z) := \{f(z) \mid z \in Z\}$ for $Z \subseteq X$. We simply write $f(Z)$ for $\hat{f}(Z)$ when there is no danger of confusion. Note that $\hat{f}$ is bijective if and only if f is bijective.

Lemma 2.53 (Isomorphic Variable Assignment and Term Model)**.** *Let Ω be a standard type structure, and let Σ be a standard signature over Ω. Let E, E' be two type environments for Ω, and let M, M' be two standard type models of Ω. Suppose that E, E' and M, M' are isomorphic wrt. the same pointwise isomorphism $I\colon \bigcup\mathcal{U} \to \bigcup\mathcal{U}$ with $I(\top) = \top$. Let $\mathcal{A}$ be a variable assignment for Σ, and let $\mathcal{M}$ be a standard term model for Σ.*

Then

$$\mathcal{A}'(x)(Y) := I(\mathcal{A}(x)(I^{-1}(Y)))$$

(for x a variable, $Y \in \mathcal{U}$) defines a variable assignment for Σ, and

$$\mathcal{M}'(c)(Y) := I(\mathcal{M}(c)(I^{-1}(Y)))$$

(for c a constant, $Y \in \mathcal{U}$) defines a standard term model for Σ.

Proof. Let x be a variable, let c be a constant, and let $Y \in \mathcal{U}$. Then $\mathcal{A}(x)(I^{-1}(Y)) \in I^{-1}(Y)$ (Def. 2.23), hence $\mathcal{A}'(x)(Y) = I(\mathcal{A}(x)(I^{-1}(Y))) \in Y$. Likewise, $\mathcal{M}'(c)(I^{-1}(Y)) \in I^{-1}(Y)$ (Def. 2.24), hence $\mathcal{M}'(c)(Y) = I(\mathcal{M}'(c)(I^{-1}(Y))) \in Y$. It remains to show that $\mathcal{M}'$ is standard.

Since M and M' are standard, we have $I(\mathbb{B}) = \mathbb{B}$ (hence $I(\top) = \top$ and I bijective implies $I(\bot) = \bot$) and $I(X \to Y) = I(X) \to I(Y)$ for X, $Y \in \mathcal{U}$. Due to the identification of functions with sets of ordered pairs (cf. Lemma 2.7), the latter implies $I(a, b) = (I(a), I(b))$ for a, $b \in \bigcup\mathcal{U}$, and $I(f)(I(a)) = I(f(a))$ for $f \in X \to Y \in \mathcal{U}$, $a \in X \in \mathcal{U}$. Thus

1. $\mathcal{M}'(\Longrightarrow)(\llbracket \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \rrbracket_{E',M'})$

$$\begin{aligned}
&= I(\mathcal{M}(\Longrightarrow)(I^{-1}(\llbracket \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \rrbracket_{E',M'})))\\
&\overset{2.51}{=} I(\mathcal{M}(\Longrightarrow)(I^{-1}(I(\llbracket \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \rrbracket_{E,M}))))\\
&\overset{2.52}{=} I(\mathcal{M}(\Longrightarrow)(\llbracket \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \rrbracket_{E,M}))\\
&\overset{2.30}{=} I(\{(\top, \{(\top, \top), (\bot, \bot)\}), (\bot, \{(\top, \top), (\bot, \top)\})\})\\
&= \{(\top, \{(\top, \top), (\bot, \bot)\}), (\bot, \{(\top, \top), (\bot, \top)\})\}
\end{aligned}$$

and

2. for every $\sigma \in \mathrm{Types}_\Omega$,
   $\mathcal{M}'(=)(\llbracket \sigma \to \sigma \to \mathsf{bool} \rrbracket_{E',M'})$

$$\begin{aligned}
&= I(\mathcal{M}(=)(I^{-1}(\llbracket \sigma \to \sigma \to \mathsf{bool} \rrbracket_{E',M'})))\\
&\overset{2.51}{=} I(\mathcal{M}(=)(I^{-1}(I(\llbracket \sigma \to \sigma \to \mathsf{bool} \rrbracket_{E,M}))))\\
&\overset{2.52}{=} I(\mathcal{M}(=)(\llbracket \sigma \to \sigma \to \mathsf{bool} \rrbracket_{E,M})),
\end{aligned}$$

   and since $\mathcal{M}(=)(\llbracket \sigma \to \sigma \to \mathsf{bool} \rrbracket_{E,M})$ is the function that maps x, $y \in \llbracket \sigma \rrbracket_{E,M}$ to $\top$ if $x = y$, and to $\bot$ otherwise (Def. 2.30), $I(\mathcal{M}(=)(\llbracket \sigma \to \sigma \to \mathsf{bool} \rrbracket_{E,M}))$ is the function that maps x, $y \in I(\llbracket \sigma \rrbracket_{E,M}) = \llbracket \sigma \rrbracket_{E',M'}$ to $I(\top) = \top$ if $x = y$, and to $I(\bot) = \bot$ otherwise (again using bijectivity of I),

as required by Def. 2.30. □

The above lemma merely shows that the pointwise isomorphism I can be used to define a new variable assignment $\mathcal{A}'$ and a new standard term model $\mathcal{M}'$. We now make the relation between a term's semantics wrt. the original variable assignment $\mathcal{A}$ and term model $\mathcal{M}$ on the one hand, and its semantics wrt. $\mathcal{A}'$ and $\mathcal{M}'$ on the other hand precise.

Lemma 2.54 (Semantics of Terms wrt. Isomorphic Types)**.** *Let Ω be a standard type structure, and let Σ be a standard signature over Ω. Let E, E' be two type environments for Ω, and let M, M' be two standard type models of Ω. Suppose that E, E' and M, M' are isomorphic wrt. the same pointwise isomorphism $I\colon \bigcup\mathcal{U} \to \bigcup\mathcal{U}$ with $I(\top) = \top$. Let $\mathcal{A}$ be a variable assignment for Σ, and let $\mathcal{M}$ be a standard term model for Σ. Define a variable assignment $\mathcal{A}'$ and a standard term model $\mathcal{M}'$ as in Lemma 2.53. Then for any term $t_\sigma \in \mathrm{Terms}_\Sigma$,*

$$I([\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}) = [\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'}.$$

Proof. By structural induction on t_σ. For $t \in \mathrm{Vars}$, we have

$$[\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'} \overset{2.25}{=} \mathcal{A}'(t)([\![\sigma]\!]_{E',M'}) \overset{2.51}{=} \mathcal{A}'(t)(I([\![\sigma]\!]_{E,M})) \overset{2.53}{=} I(\mathcal{A}(t)([\![\sigma]\!]_{E,M})) \overset{2.25}{=} I([\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}).$$

Likewise for $t \in \mathrm{Names}$,

$$[\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'} \overset{2.25}{=} \mathcal{M}'(t)([\![\sigma]\!]_{E',M'}) \overset{2.51}{=} \mathcal{M}'(t)(I([\![\sigma]\!]_{E,M})) \overset{2.53}{=} I(\mathcal{M}(t)([\![\sigma]\!]_{E,M})) \overset{2.25}{=} I([\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}).$$

If t is an application, i.e. $t_\sigma = (t^1_{\sigma'\to\sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma'\to\sigma}$, $t^2_{\sigma'} \in \mathrm{Terms}_\Sigma$, then

$$\begin{aligned}
[\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'} &\overset{2.25}{=} [\![t^1_{\sigma'\to\sigma}]\!]_{\mathcal{A}',\mathcal{M}'}([\![t^2_{\sigma'}]\!]_{\mathcal{A}',\mathcal{M}'}) \\
&\overset{\text{IH}}{=} I([\![t^1_{\sigma'\to\sigma}]\!]_{\mathcal{A},\mathcal{M}})(I([\![t^2_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}})) \\
&\overset{\text{Proof of } 2.53}{=} I([\![t^1_{\sigma'\to\sigma}]\!]_{\mathcal{A},\mathcal{M}}([\![t^2_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}})) \\
&\overset{2.25}{=} I([\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}).
\end{aligned}$$

If t is a λ-abstraction, i.e. $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1\to\sigma_2}$ for some variable x, some $\sigma_1 \in \mathrm{Types}_\Omega$, and some $t'_{\sigma_2} \in \mathrm{Terms}_\Sigma$, then

$$\begin{aligned}
[\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'} &\overset{2.25}{=} \{(d', [\![t'_{\sigma_2}]\!]_{\mathcal{A}'[x_{\sigma_1}\mapsto d'],\mathcal{M}'}) \mid d' \in [\![\sigma]\!]_{E',M'}\} \\
&\overset{2.51}{=} \{(I(d), [\![t'_{\sigma_2}]\!]_{\mathcal{A}'[x_{\sigma_1}\mapsto I(d)],\mathcal{M}'}) \mid d \in [\![\sigma]\!]_{E,M}\} \\
&\overset{\text{IH}}{=} \{(I(d), I([\![t'_{\sigma_2}]\!]_{\mathcal{A}[x_{\sigma_1}\mapsto d],\mathcal{M}})) \mid d \in [\![\sigma]\!]_{E,M}\} \\
&\overset{\text{Proof of } 2.53}{=} I(\{(d, [\![t'_{\sigma_2}]\!]_{\mathcal{A}[x_{\sigma_1}\mapsto d],\mathcal{M}}) \mid d \in [\![\sigma]\!]_{E,M}\}) \\
&\overset{2.25}{=} I([\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}).
\end{aligned}$$

□

The main result of this paragraph is now an easy corollary. A HOL formula is satisfiable wrt. a type environment and a standard type model iff it is satisfiable wrt. any isomorphic type environment and standard type model.

Corollary 2.55 (Satisfiability wrt. Isomorphic Types)**.** *Let Ω be a standard type structure, and let Σ be a standard signature over Ω. Let E, E' be two type environments for Ω, and let M, M' be two standard type models of Ω. Suppose that E, E' and M, M' are isomorphic wrt. the same pointwise isomorphism $I\colon \bigcup\mathcal{U} \to \bigcup\mathcal{U}$ with $I(\top) = \top$. Let $t_{\mathsf{bool}} \in \mathrm{Terms}_\Sigma$. Then t_{bool} is satisfiable wrt. E and M iff t_{bool} is satisfiable wrt. E' and M'.*

Proof. Let $\mathcal{A}$ be a variable assignment and $\mathcal{M}$ be a standard term model for Σ such that $[\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}} = \top$ (wrt. E and M). For E' and M', define a variable assignment $\mathcal{A}'$ and a standard term model $\mathcal{M}'$ as in Lemma 2.53. Then using Lemma 2.54, we have

$$[\![t_\sigma]\!]_{\mathcal{A}',\mathcal{M}'} = I([\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}) = I(\top) = \top.$$

□

Therefore satisfiability of HOL formulae needs to be tested only modulo pointwise isomorphisms. If a formula is not satisfiable wrt. one type environment and standard type model, it is not satisfiable wrt. any isomorphic type environment and model either.

2.3.3 Interpretation of Terms

Given a type environment E and a standard type model M, our task now is to find a variable assignment $\mathcal{A}$ and a term model $\mathcal{M}$ with $[\![t_{\mathsf{bool}}]\!]_{\mathcal{A},\mathcal{M}} = \top$. (To generate a countermodel instead of a model, we can either consider $\neg t_{\mathsf{bool}}$, or—equivalently—search for $\mathcal{A}$ and $\mathcal{M}$ with $[\![t_{\mathsf{bool}}]\!]_{\mathcal{A},\mathcal{M}} = \bot$.) At this point one can already view finite model generation as a generalization of satisfiability checking, where the search tree is not necessarily binary, but still finite.

In principle we could search for $\mathcal{A}$ and $\mathcal{M}$ by explicit enumeration and evaluation of t_{bool} under all possible combinations of variable assignments and term models. This however is infeasible for all but the smallest examples. We therefore translate t_{bool} into a propositional formula, leaving the search for a satisfying variable assignment and term model to a SAT solver. Our confidence that the SAT solver is more efficient than a brute force approach is justified by significant advances in the area of propositional satisfiability solving in recent years [27].

The translation $\mathcal{T}$ of terms into propositional formulae is by structural induction on the term. Although our final aim is to translate a term of type bool into a single propositional formula, a more complex intermediate data structure is needed to translate subterms, which may be of arbitrary type. We use finite trees whose leafs are labeled with lists of propositional formulae. The construction of these trees is described in detail in the remainder of this section.

Definition 2.56 (Labeled Tree)**.** Let X be a set of *labels*. The set Tree_X of *trees with labels in X* is the smallest set such that

1. if $x \in X$, then $\mathrm{Leaf}(x) \in \mathrm{Tree}_X$, and
2. if $t \in \mathrm{List}_{\mathrm{Tree}_X}$, then $\mathrm{Node}(t) \in \mathrm{Tree}_X$.

The Tree operator is monotonic wrt. the subset relation.

Lemma 2.57. *Let $X \subseteq Y$. Then* $\mathrm{Tree}_X \subseteq \mathrm{Tree}_Y$.

Proof. Let $t \in \mathrm{Tree}_X$. The proof is by structural induction on t, using Lemma 2.45. □

The translation is then a (parameterized) function from Terms_Σ to $\mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$. A leaf of length m corresponds to a term whose type is given by a type variable, or by a type constructor other than $\to$ (denoting a set of size m), while an n-ary function or predicate is given by a tree of height $n+1$. We will show how application and λ-abstraction can be "lifted" from the term level to this intermediate data structure.—Note that we could have chosen $\mathrm{Tree}_\mathbb{P}$ instead of $\mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$ for the codomain of $\mathcal{T}$, since each leaf labeled with a list can easily be encoded as a tree of height 2, with as many labeled leafs as the original list had elements. This makes no real difference, except that the current choice is more natural for our application: leafs immediately correspond to base types, and nodes correspond to function types.

Definition 2.58 (Trees for Types)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a standard type structure. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. For $\sigma \in \mathrm{Types}_\Omega$, the set $\mathrm{Trees}(\sigma)$ of *trees for σ (wrt. E and M)* is defined as follows:

1. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, then

$$\mathrm{Trees}(\sigma) := \left\{ \mathrm{Leaf}([\varphi_1, \ldots, \varphi_k]) \mid \varphi_1, \ldots, \varphi_k \in \mathbb{P},\ k = |[\![\sigma]\!]_{E,M}| \right\}.$$

2. If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then

$$\mathrm{Trees}(\sigma) := \left\{ \mathrm{Node}([t_1, \ldots, t_k]) \mid t_1, \ldots, t_k \in \mathrm{Trees}(\sigma_2),\ k = |[\![\sigma_1]\!]_{E,M}| \right\}.$$

Note that $k \geq 1$ in both cases, since types are interpreted as non-empty sets. Also note that $\mathrm{Trees}(\sigma) = \mathrm{Trees}(\sigma')$ does not imply $\sigma = \sigma'$. The condition $t \in \mathrm{Trees}(\sigma)$ merely ensures that t has the proper shape to denote an element of $[\![\sigma]\!]_{E,M}$, according to the semantics of trees given below.

Remark 2.59. Let $\sigma \in \mathrm{Types}_\Omega$. Then $\mathrm{Trees}(\sigma) \subseteq \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$.

Proof. By structural induction on σ. □

The meaning of a tree denoting an element of some type σ is defined wrt. a truth assignment that gives the meaning of propositional formulae occuring in the tree's labels. To refer to the i-th element of a type, the definition makes use of the total orders that were introduced for finite sets representing types in Section 2.3.2.

Definition 2.60 (Semantics of Trees)**.** Let $\Omega = (\mathrm{TyVars}, \mathrm{TyNames}, \mathrm{TyArity})$ be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Furthermore, let $\sigma \in \mathrm{Types}_\Omega$, and let A be a truth assignment. The *meaning* $[\![t]\!]_{\sigma,A}$ *of a tree* $t \in \mathrm{Trees}(\sigma)$ *(wrt. σ and A)* is defined as follows:

1. If $\sigma \in \text{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \text{TyNames} \setminus \{\to\}$, then $t = \text{Leaf}([\varphi_1, \ldots, \varphi_k])$ for some $\varphi_1, \ldots, \varphi_k \in \mathbb{P}$, where $k = |[\![\sigma]\!]_{E,M}|$. Let $[d_1, \ldots, d_k] = [\![\sigma]\!]_{E,M}$. In this case,

$$[\![t]\!]_{\sigma,A} := \begin{cases} d_i, & \text{if } [\![\varphi_i]\!]_A = \top \text{ and } [\![\varphi_j]\!]_A = \bot \text{ for all } j \neq i \\ & \qquad\qquad \text{(where } 1 \leq j \leq k\text{)}; \\ \text{undefined}, & \text{if no such } i \text{ (with } 1 \leq i \leq k\text{) exists.} \end{cases}$$

2. If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \text{Types}_\Omega$, then $t = \text{Node}([t_1, \ldots, t_k])$ for some $t_1, \ldots, t_k \in \text{Trees}(\sigma_2)$, where $k = |[\![\sigma_1]\!]_{E,M}|$. Let $[d_1, \ldots, d_k] = [\![\sigma_1]\!]_{E,M}$. In this case $[\![t]\!]_{\sigma,A}$ is defined as the—possibly partial—function that sends $d_i \in [\![\sigma_1]\!]_{E,M}$ (for $1 \leq i \leq k$) to $[\![t_i]\!]_{\sigma_2,A}$ (if the latter is defined).

Remark 2.61. Let $t \in \text{Trees}(\sigma)$. Then $[\![t]\!]_{\sigma,A} \in [\![\sigma]\!]_{E,M}$ iff $[\![t]\!]_{\sigma,A}$ is defined or—in case σ is a function type—$[\![t]\!]_{\sigma,A}$ is a total function, and the same holds recursively for every subtree of t (wrt. its corresponding type).

Proof. By structural induction on σ. □

Note that for $\sigma, \sigma' \in \text{Types}_\Omega$, both $[\![t]\!]_{\sigma,A}$ and $[\![t]\!]_{\sigma',A}$ may be defined (and may or may not differ) if $\text{Trees}(\sigma) = \text{Trees}(\sigma')$.

As one can see from Def. 2.60, Boolean variables are used in a unary, rather than in a binary fashion. This means that we need n variables to represent an element of a base type of size n, rather than $\lceil \log_2 n \rceil$ variables. However, at most one of these variables may later be set to true (which keeps the search space for the SAT solver small due to unit propagation [178]), and our encoding—which is inspired by [81]—allows for a relatively simple translation of first-order application: only a single Boolean variable needs to be considered when we want to know if a function's argument (of base type) denotes a particular value. On the other hand, the representation of functions by trees still yields an encoding that is linear in the size of a function's domain.

The previous paragraph already hints at the two somewhat independent choices that must be made when one defines the translation from terms to propositional formulae. First, terms of base type are encoded as lists of Boolean variables, which can be used either in a unary or in a binary fashion. Second, functions can be encoded as trees, supporting the view that a function corresponds to a table of its values. Alternatively, functions could be encoded as lists just like terms of base type—making trees completely unnecessary—, merely by noticing that the function space is finite (and by forgetting its "internal" structure).

Both choices affect how the application of a function to its argument must be encoded. The translation of application is in any case based on the idea of an explicit case distinction over the function's domain: if we know which value in the domain is denoted by the argument, we also know which entry in the table of function values gives the result of the application. From this point of view, it is best to encode the function as a tree (from which we can immediately read off a function value by looking at the corresponding subtree), and to use the linear list encoding for the argument (since a case distinction over the domain then depends on the truth value of single Boolean variables). Any other combination of encodings would result in a fair amount of arithmetic having to be used in the encoding of the application's result.

In higher-order logic however, functions can be arguments themselves. We still encode functions as trees, but when a function occurs as an argument to another (higher-order) function, we will need an additional translation step to turn the tree encoding of the argument function into its linear list encoding. The formal details are given later in this section, when the translation $\mathcal{T}$ is defined for applications.

Well-formed Truth Assignments

The alert reader will notice that Def. 2.60 introduces two kinds of partiality: a leaf has an undefined meaning if none of the label's elements evaluate to $\top$, or if more than one label element evaluates to $\top$. The first kind of undefinedness will actually turn out to be useful later, when we consider datatypes and recursive functions (see Section 3.6). Nevertheless for the time being we want to rule out this kind of undefinedness (to simplify the correctness proof given in this chapter), as well as the second kind, which could be interpreted as "a leaf denotes two (or more) of the type's elements at the same time". To this end well-formedness formulae are introduced that impose restrictions on the truth assignment.

Definition 2.62 (Well-formed Truth Assignment)**.** Let $t \in \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$. A truth assignment $A\colon \mathcal{B} \to \mathbb{B}$ is *well-formed wrt. t* iff every label $[x_1, \ldots, x_n]$ of t contains exactly one propositional formula x_i with $[\![x_i]\!]_A = \top$.

Let $T \subseteq \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$ be a set of trees. A truth assignment $A\colon \mathcal{B} \to \mathbb{B}$ is *well-formed wrt. T* iff A is well-formed wrt. each $t \in T$.

Lemma 2.63. *Let $t_1, \ldots, t_n \in \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$. A truth assignment $A\colon \mathcal{B} \to \mathbb{B}$ is well-formed wrt.* $\mathrm{Node}([t_1, \ldots, t_n])$ *iff A is well-formed wrt. $\{t_1, \ldots, t_n\}$.*

Proof. $l \in \mathrm{List}_\mathbb{P}$ is a label of $\mathrm{Node}([t_1, \ldots, t_n])$ iff l is a label of (at least) one of the trees t_1, $\ldots$, t_n. □

Definition 2.64 (Well-formedness Formula)**.** Let $l = [x_1, \ldots, x_n] \in \mathrm{List}_\mathbb{P}$. The *well-formedness formula for l*, written $\mathrm{wf}(l)$, is defined as

$$\Big(\bigvee_{i=1}^{n} x_i\Big) \wedge \bigwedge_{\substack{i,j=1 \\ i \neq j}}^{n} (\neg x_i \vee \neg x_j).$$

Let $t \in \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$. The set of well-formedness formulae for t, $\mathrm{wf}(t)$, is defined as the set of all well-formedness formulae $\mathrm{wf}(l)$ such that l is a label of t.

Let $T \subseteq \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$ be a set of trees. The set of well-formedness formulae for T, $\mathrm{wf}(T)$, is defined as $\bigcup_{t \in T} \mathrm{wf}(t)$.

Lemma 2.65. *Let $T \subseteq \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$ be a set of trees. A truth assignment $A\colon \mathcal{B} \to \mathbb{B}$ is well-formed wrt. T iff $A \models \varphi$ for every $\varphi \in \mathrm{wf}(T)$.*

Proof. Let $T \subseteq \mathrm{Tree}_{\mathrm{List}_\mathbb{P}}$ be a set of trees, and let $A\colon \mathcal{B} \to \mathbb{B}$ be a truth assignment.

Suppose A is well-formed wrt. T. Let $\varphi = \big(\bigvee_{i=1}^{n} x_i\big) \wedge \bigwedge_{i,j=1;i\neq j}^{n} (\neg x_i \vee \neg x_j)$ be in $\mathrm{wf}(T)$. Then $[x_1, \ldots, x_n]$ is a label of some tree in T. Hence well-formedness of A implies that there is exactly one formula x_i with $[\![x_i]\!]_A = \top$. Therefore also $[\![\varphi]\!]_A = \top$, i.e. $A \models \varphi$.

For the other direction of the equivalence, suppose $[\![\varphi]\!]_A = \top$ for every $\varphi \in \mathrm{wf}(T)$. Let $l = [x_1, \ldots, x_n]$ be a label of a tree in T. Then in particular $\mathrm{wf}(l) \in \mathrm{wf}(T)$. Hence $[\![\mathrm{wf}(l)]\!]_A = \top$, and therefore $[\![x_i]\!]_A = \top$ for exactly one propositional formula x_i. □

The truth assignment being well-formed is a necessary and sufficient condition for the meaning of a tree to be an element of the tree's corresponding type.

Lemma 2.66. *Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Furthermore, let $\sigma \in \mathrm{Types}_\Omega$, and let A be a truth assignment. Let $t \in \mathrm{Trees}(\sigma)$. Then $[\![t]\!]_{\sigma,A} \in [\![\sigma]\!]_{E,M}$ iff A is well-formed wrt. t.*

Proof. From Def. 2.60, using Remark 2.61. □

Terms are variables and constants, λ-abstractions, or applications. We will now consider each of these cases, before we put everything together to define a single translation function near the end of the section. Aside from describing how terms are translated to trees in each case, we will also prepare the correctness proof by explaining why the translation works as intended. The general proof structure is as follows. For the base case of variables and constants, we show that (under certain assumptions) the existence of a variable assignment and term model that assign certain meanings is equivalent to the existence of a well-formed truth assignment that assigns the same meanings to the trees that result from translating the variables and constants. Next we show that the translation of λ-abstraction and application preserves the meaning of terms, i.e. a λ-abstraction is translated as a tree which denotes a certain function, and an application term is translated as function application. (In particular, if the trees for the immediate subterms of a term have a defined meaning, then so does the tree for the whole term. In other words, a truth assignment which is well-formed wrt. the trees for a term's free variables and constants is also well-formed wrt. the tree which results from translating the entire term.) Together these properties imply that a HOL formula is satisfiable iff its translation, under some well-formed truth assignment, denotes $\top$.

Variables and Constants

We define tree assignments and tree models as analogues of variable assignments and term models. Tree assignments (tree models) map each explicitly typed variable (constant) to a tree of the proper shape. They allow us to establish a connection between the interpretation of terms via variable assignments and term models on the one hand, and the interpretation of Boolean variables via truth assignments on the other hand.

Definition 2.67 (Tree Assignment)**.** Let $\Sigma = (\mathrm{Vars}, \mathrm{Names}, \mathrm{Typ})$ be a signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. A *tree assignment $\bar{T}$ for Σ (wrt. E and M)* assigns to each explicitly typed variable $x_\sigma \in \mathrm{Terms}_\Sigma$ (where $x \in \mathrm{Vars}$, $\sigma \in \mathrm{Types}_\Omega$) a tree $\bar{T}(x_\sigma) \in \mathrm{Trees}(\sigma)$.

Definition 2.68 (Tree Model)**.** Let $\Sigma = (\mathrm{Vars}, \mathrm{Names}, \mathrm{Typ})$ be a signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. A *tree model $\bar{M}$ for Σ (wrt. E and M)* assigns to each explicitly typed constant $c_\sigma \in \mathrm{Terms}_\Sigma$ (where $c \in \mathrm{Names}$, $\sigma \in \mathrm{Types}_\Omega$) a tree $\bar{M}(c_\sigma) \in \mathrm{Trees}(\sigma)$.

A term's variables and constants are interpreted independently of each other by a variable assignment and term model, respectively, with no restrictions other than those imposed on standard term models in Def. 2.30. A tree assignment and tree model on the other hand could impose restrictions on the interpretation of variables and constants by using propositional formulae, rather than just Boolean variables, as label elements. Also, the same Boolean variable could be used in more than one label. We rule out such unwanted restrictions and interdependencies with the following definition of *standard* tree assignments and tree models.

Definition 2.69 (Standard Tree Assignment/Tree Model)**.** Let $\Sigma = (\mathrm{Vars}, \mathrm{Names}, \mathrm{Typ})$ be a standard signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). We say that $\bar{T}$ and $\bar{M}$ are *standard* (wrt. each other) iff

1. $\bar{T}(x_\sigma) \in \mathrm{Tree}_{\mathrm{List}_\mathcal{B}}$, for every $x_\sigma \in \mathrm{Terms}_\Sigma$ (where $x \in \mathrm{Vars}$, $\sigma \in \mathrm{Types}_\Omega$); and
2. $\bar{M}(c_\sigma) \in \mathrm{Tree}_{\mathrm{List}_\mathcal{B}}$, for every $c_\sigma \in \mathrm{Terms}_\Sigma$ (where $c \in \mathrm{Names}$, $\sigma \in \mathrm{Types}_\Omega$), provided c_σ is not equal to $\Longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$, and c_σ is not equal to $=_{\sigma'\to\sigma'\to\mathsf{bool}}$ for any $\sigma' \in \mathrm{Types}_\Omega$; and
3. $\bar{M}(\Longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}) = \mathrm{Node}([\mathrm{Node}([\mathrm{Top}, \mathrm{Bot}]), \mathrm{Node}([\mathrm{Top}, \mathrm{Top}])])$, where Top and Bot abbreviate $\mathrm{Leaf}([\mathrm{True}, \mathrm{False}])$ and $\mathrm{Leaf}([\mathrm{False}, \mathrm{True}])$, respectively; and
4. for every $\sigma \in \mathrm{Types}_\Omega$, $\bar{M}(=_{\sigma\to\sigma\to\mathsf{bool}}) = \mathrm{Node}([\mathrm{Node}(\mathrm{UV}_1^k), \ldots, \mathrm{Node}(\mathrm{UV}_k^k)])$, where $k = |[\![\sigma]\!]_{E,M}|$ and UV_n^k is defined as the list $[t_1, \ldots, t_k]$ that is given by

$$t_i := \begin{cases} \mathrm{Top} & \text{if } i = n; \\ \mathrm{Bot} & \text{otherwise;} \end{cases}$$

   and
5. each Boolean variable occurs at most once as the element of a label in the range of $\bar{T}$ and $\bar{M}$, i.e. no label contains the same Boolean variable more than once, and no two labels contain the same Boolean variable.

The first condition states that trees for variables may only use Boolean variables in labels, but no other propositional formulae. The second condition imposes the same restriction on trees for constants (other than implication and equality, whose meanings are fixed, and hence the corresponding trees are fixed by the third and fourth condition, respectively). The last condition allows us to interpret different terms—and moreover a function's values for different arguments—independently of each other.

Remark 2.70. Without loss of generality we may assume that standard tree assignments and tree models for our fixed standard signature Σ (over the fixed standard type structure Ω) exist.

Proof. Regarding conditions 1 and 2, note that any function $f\colon \mathbb{P} \to \mathcal{B}$, if applied to every label element of a tree in $\mathrm{Trees}(\sigma)$, will yield a tree in $\mathrm{Tree}_{\mathrm{List}_\mathcal{B}} \cap \mathrm{Trees}(\sigma)$. In particular, $\mathrm{Tree}_{\mathrm{List}_\mathcal{B}} \cap \mathrm{Trees}(\sigma) \neq \emptyset$ for any type $\sigma \in \mathrm{Types}_\Omega$.

Conditions 3 and 4 are trivially satisfiable: $\mathrm{Node}([\mathrm{Node}([\mathrm{Top},\mathrm{Bot}]),\mathrm{Node}([\mathrm{Top},\mathrm{Top}])]) \in \mathrm{Trees}(\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool})$, and $\mathrm{Node}([\mathrm{Node}(\mathrm{UV}_1^k),\ldots,\mathrm{Node}(\mathrm{UV}_k^k)]) \in \mathrm{Trees}(\sigma \to \sigma \to \mathsf{bool})$ for any type $\sigma \in \mathrm{Types}_\Omega$ (where $k = |[\![\sigma]\!]_{E,M}|$).

For condition 5, recall that our only requirement on $\Omega = (\mathrm{TyVars},\mathrm{TyNames},\mathrm{TyArity})$ and $\Sigma = (\mathrm{Vars},\mathrm{Names},\mathrm{Typ})$ so far, aside from being standard, was that the input formula t_{bool} is in Terms_Σ. Therefore we can choose sufficiently small sets for TyVars, TyNames, Vars and Names to allow the Boolean variables used as label elements (which are drawn from the infinite set $\mathcal{B}$) to be distinct. □

The rationale behind Def. 2.69 is the following. For any tree assignment and tree model that are standard wrt. each other, we want the existence of a variable assignment and standard term model that assign certain meanings to variables and constants, respectively, to be equivalent to the existence of a well-formed truth assignment that assigns the same meanings to the trees that correspond to these variables and constants. This property indeed holds, as shown by the next lemma.

Lemma 2.71. *Let $\Sigma = (\mathrm{Vars},\mathrm{Names},\mathrm{Typ})$ be a standard signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). Assume that $\bar{T}$ and $\bar{M}$ are standard (wrt. each other). Let $t_\sigma \in \mathrm{Terms}_\Sigma$.*

For every variable assignment $\mathcal{A}$ and standard term model $\mathcal{M}$ (for Σ) there exists a truth assignment A that is well-formed wrt. $\bar{T}(\mathrm{FreeVars}(t_\sigma)) \cup \bar{M}(\mathrm{Names}(t_\sigma))$ such that $[\![x_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}} = [\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A}$ for every $x_{\sigma'} \in \mathrm{FreeVars}(t_\sigma)$, and $[\![c_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}} = [\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A}$ for every $c_{\sigma'} \in \mathrm{Names}(t_\sigma)$.

Also, for every truth assignment A that is well-formed wrt. $\bar{T}(\mathrm{FreeVars}(t_\sigma)) \cup \bar{M}(\mathrm{Names}(t_\sigma))$, there exist a variable assignment $\mathcal{A}$ and standard term model $\mathcal{M}$ (for Σ) such that $[\![x_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}} = [\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A}$ for every $x_{\sigma'} \in \mathrm{FreeVars}(t_\sigma)$, and $[\![c_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}} = [\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A}$ for every $c_{\sigma'} \in \mathrm{Names}(t_\sigma)$.

We prove an auxiliary lemma first, namely that trees of Boolean variables satisfying the distinctness condition of Def. 2.69 can denote any particular element of their corresponding type's meaning if a suitable truth assignment is chosen. In addition, we may assume that this truth assignment is well-formed.

Lemma 2.72. *Let $\Omega = (\mathrm{TyVars},\mathrm{TyNames},\mathrm{TyArity})$ be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Furthermore, let $\sigma \in \mathrm{Types}_\Omega$, and let $t \in \mathrm{Trees}(\sigma) \cap \mathrm{Tree}_{\mathrm{List}_\mathcal{B}}$ such that t satisfies condition 5 of Def. 2.69.*

Then for any $d \in [\![\sigma]\!]_{E,M}$ there exists a truth assignment $A\colon \mathcal{B} \to \mathbb{B}$ that is well-formed wrt. t such that $[\![t]\!]_{\sigma,A} = d$.

Proof. By structural induction on σ. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1,\ldots,\sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, then $t = \mathrm{Leaf}([x_1,\ldots,x_k])$ for some $x_1, \ldots, x_k \in \mathcal{B}$, where $k = |[\![\sigma]\!]_{E,M}|$. Let $[\![\sigma]\!]_{E,M} = [d_1,\ldots,d_k]$, and assume $d = d_i$ (for some $1 \le i \le k$). In this case, define $A\colon \mathcal{B} \to \mathbb{B}$ by

$$A(x) := \begin{cases} \top, & \text{if } x = x_i; \\ \bot, & \text{otherwise.} \end{cases}$$

Note that $x_1, \ldots, x_k$ are distinct due to Def. 2.69. In particular, $x_j = x_i$ (for $1 \leq j \leq k$) if and only if $j = i$. Using this, one easily verifies that A is well-formed wrt. t (Def. 2.62), and that $[\![t]\!]_{\sigma,A} = d_i$ as required (Def. 2.60).

If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then $t = \mathrm{Node}([t_1, \ldots, t_k])$ for some $t_1, \ldots, t_k \in \mathrm{Trees}(\sigma_2)$, where $k = |[\![\sigma_1]\!]_{E,M}|$. Let $[d_1, \ldots, d_k] = [\![\sigma_1]\!]_{E,M}$. Then d, as a function from $[\![\sigma_1]\!]_{E,M}$ to $[\![\sigma_2]\!]_{E,M}$, is given by $d = \{(d_1, d(d_1)), \ldots, (d_k, d(d_k))\}$, where $d(d_1), \ldots, d(d_k) \in [\![\sigma_2]\!]_{E,M}$. In this case the induction hypothesis yields truth assignments $A_1, \ldots, A_k \colon \mathcal{B} \to \mathbb{B}$ such that each A_i (for $1 \leq i \leq k$) is well-formed wrt. t_i, $[\![t_i]\!]_{\sigma,A_i} = d(d_i)$, and without loss of generality $A_i(x) = \top$ only for variables $x \in \mathcal{B}$ that occur as label elements of t_i. Now define $A \colon \mathcal{B} \to \mathbb{B}$ by

$$A(x) := \bigvee_{i=1}^{k} A_i(x).$$

A is well-formed wrt. each t_i because Def. 2.69 implies that the label elements of each tree t_j (for $1 \leq j \leq k$, $j \neq i$) are disjoint from those of t_i; hence $A_j(x) = \bot$ for each $j \neq i$, where x is a label element of t_i. Therefore A is well-formed wrt. t (Lemma 2.63). Moreover, $[\![t_i]\!]_{\sigma,A} = d(d_i)$ for the same reason. This immediately implies $[\![t]\!]_{\sigma,A} = d$ (Def. 2.60). □

The proof of Lemma 2.71 follows.

Proof. Assume that $\mathcal{A}$ is a variable assignment and $\mathcal{M}$ is a standard term model for Σ. Since $\bar{T}$ and $\bar{M}$ are standard, we can use Lemma 2.72 to obtain well-formed truth assignments $A_{x_{\sigma'}}$ (for every $x_{\sigma'} \in \mathrm{FreeVars}(t_\sigma)$) and $A_{c_{\sigma'}}$ (for every $c_{\sigma'} \in \mathrm{Names}(t_\sigma)$, provided $c_{\sigma'}$ is not equal to $\Longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$, and $c_{\sigma'}$ is not equal to $=_{\tau\to\tau\to\mathsf{bool}}$ for any $\tau \in \mathrm{Types}_\Omega$) such that $[\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A_{x_{\sigma'}}} = [\![x_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}}$ and $[\![\bar{M}(c_{\sigma'})]\!]_{\sigma,A_{c_{\sigma'}}} = [\![c_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}}$. Now define $A \colon \mathcal{B} \to \mathbb{B}$ by

$$A(v) := \begin{cases} A_{x_{\sigma'}}(v), & \text{if } v \text{ is a label element of } \bar{T}(x_{\sigma'}); \\ A_{c_{\sigma'}}(v), & \text{if } v \text{ is a label element of } \bar{M}(c_{\sigma'}); \\ \bot, & \text{otherwise.} \end{cases}$$

Condition 5 of Def. 2.69 ensures that A is well-defined. A is well-formed wrt. $\bar{T}(\mathrm{FreeVars}(t_\sigma)) \cup \bar{M}(\mathrm{Names}(t_\sigma))$ because each truth assignment $A_{x_{\sigma'}}$ and $A_{c_{\sigma'}}$ is well-formed wrt. $\bar{T}(x_{\sigma'})$ and $\bar{M}(c_{\sigma'})$, respectively. Furthermore, $[\![x_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}} = [\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A}$ for every $x_{\sigma'} \in \mathrm{FreeVars}(t_\sigma)$, and $[\![c_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}} = [\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A}$ for every $c_{\sigma'} \in \mathrm{Names}(t_\sigma)$ (where conditions 3 and 4 of Def. 2.69 are needed for implication and equality terms, respectively). This proves the first part of the lemma.

Next, assume $A \colon \mathcal{B} \to \mathbb{B}$ is a truth assignment that is well-formed wrt. $\bar{T}(\mathrm{FreeVars}(t_\sigma)) \cup \bar{M}(\mathrm{Names}(t_\sigma))$. Then $[\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A} \in [\![\sigma']\!]_{E,M}$ for every $x_{\sigma'} \in \mathrm{FreeVars}(t_\sigma)$, and $[\![\bar{M}(c_{\sigma'})]\!]_{\sigma,A} \in [\![\sigma']\!]_{E,M}$ for every $c_{\sigma'} \in \mathrm{Names}(t_\sigma)$ by Lemma 2.66. Thus we can simply define the variable assignment $\mathcal{A}$ and the term model $\mathcal{M}$ by

$$\mathcal{A}(x_{\sigma'}) := [\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A}$$

for $x_{\sigma'} \in \mathrm{FreeVars}(t_\sigma)$, and

$$\mathcal{M}(c_{\sigma'}) := [\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A}$$

for $c_{\sigma'} \in \text{Names}(t_\sigma) \cup \{ \Longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}\} \cup \{=_{\tau\to\tau\to\mathsf{bool}} \mid \tau \in \text{Types}_\Omega\}$ (where we extend A and M to other variables and constants in an arbitrary fashion). Conditions 3 and 4 of Def. 2.69 imply that $\mathcal{M}$ is standard. This concludes the proof of the lemma's second part. □

Lemma 2.71 establishes the close connection between the existence of HOL models and propositional models that we need to show satisfiability equivalence of our translation for the base cases of variables and constants.

λ-Abstraction

To simplify notation, we introduce an auxiliary operator map (well-established in functional programming) which applies an argument function to every element of a list.

Definition 2.73 (map)**.** Let X, Y be sets. For $f\colon X \to Y$ and $[x_1, \ldots, x_n] \in \text{List}_X$, define

$$\text{map}(f, [x_1, \ldots, x_n]) := [f(x_1), \ldots, f(x_n)].$$

Remark 2.74. For $f\colon X \to Y$ a function and $l \in \text{List}_X$, we have $\text{map}(f, l) \in \text{List}_Y$, i.e. $\text{map}(f, \cdot)\colon \text{List}_X \to \text{List}_Y$. Furthermore, l and $\text{map}(f, l)$ have the same length.

Proof. By structural induction on l. □

Lemma 2.75. *Let $f\colon X \to Y$, $g\colon Y \to Z$ be two functions, and let $l \in \text{List}_X$. Then* $\text{map}(g, \text{map}(f, l)) = \text{map}(g \circ f, l)$.

Proof. By structural induction on l. □

We now define trees whose leafs are labeled with lists of propositional constants (i.e. True and False) only, and where exactly one element in each label is True, while all others are False. Independently of the truth assignment, these trees denote specific (i.e. the first, second, ...) elements of their corresponding type. Moreover, we can define a function consts which returns trees corresponding to a type's elements in the correct order. This function will be used in the translation of λ-abstractions, whose body needs to be evaluated separately for each possible value of the bound variable.

Definition 2.76 (Propositional Unit Vector)**.** For $1 \leq n \leq k$, uv_n^k, the *n-th propositional unit vector of length k*, is defined as the list $[\varphi_1, \ldots, \varphi_k] \in \text{List}_{\{\text{True,False}\}}$ that is given by

$$\varphi_i := \begin{cases} \text{True} & \text{if } i = n; \\ \text{False} & \text{otherwise.} \end{cases}$$

Definition 2.77 (Constant Trees)**.** Let $\Omega = (\text{TyVars}, \text{TyNames}, \text{TyArity})$ be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. For $\sigma \in \text{Types}_\Omega$, the *constant trees for σ (wrt. E and M)*, written $\text{consts}(\sigma)$, are defined as follows:

1. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, then
$$\mathrm{consts}(\sigma) := [\mathrm{Leaf}(\mathrm{uv}_1^k), \ldots, \mathrm{Leaf}(\mathrm{uv}_k^k)],$$
where $k = |[\![\sigma]\!]_{E,M}|$.
2. If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then
$$\mathrm{consts}(\sigma) := \mathrm{map}(\mathrm{Node}, \mathrm{pick}(k, \mathrm{consts}(\sigma_2))),$$
where $k = |[\![\sigma_1]\!]_{E,M}|$.

Example 2.78. As an example, consider the constant trees for bool (wrt. an arbitrary finite type environment and finite standard type model). Using the canonical order on $\mathbb{B}$ (Def. 2.41), the tree for $\top$ is given by $\mathrm{Leaf}([\mathrm{True}, \mathrm{False}])$, while the tree for $\bot$ is given by $\mathrm{Leaf}([\mathrm{False}, \mathrm{True}])$.

Remark 2.79. Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. For $\sigma \in \mathrm{Types}_\Omega$, $\mathrm{consts}(\sigma)$ is a list of length $|[\![\sigma]\!]_{E,M}|$, where each list element is in $\mathrm{Trees}(\sigma) \cap \mathrm{Tree}_{\mathrm{List}_{\{\mathrm{True},\mathrm{False}\}}}$.

Proof. By structural induction on σ. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, the claim follows immediately from Def. 2.76 and Def. 2.58.

If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then $\mathrm{consts}(\sigma_2)$ has length $|[\![\sigma_2]\!]_{E,M}|$ by the induction hypothesis. Hence $\mathrm{consts}(\sigma)$ has length $|[\![\sigma_2]\!]_{E,M}|^k$ (Remark 2.74 and Remark 2.47), which is equal to $|[\![\sigma]\!]_{E,M}| = |[\![\sigma_1 \to \sigma_2]\!]_{E,M}| = |[\![\sigma_2]\!]_{E,M}|^{|[\![\sigma_1]\!]_{E,M}|}$ since M is standard (Remark 2.39). That each list element is in $\mathrm{Trees}(\sigma) \cap \mathrm{Tree}_{\mathrm{List}_{\{\mathrm{True},\mathrm{False}\}}}$ also follows from the induction hypothesis, together with Remark 2.74 and Remark 2.47 (and of course Def. 2.58). □

The key property of $\mathrm{consts}(\sigma)$ is stated and proved below.

Lemma 2.80. *Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Furthermore, let $\sigma \in \mathrm{Types}_\Omega$, and let A be an arbitrary truth assignment. Then*
$$\mathrm{map}([\![\,\cdot\,]\!]_{\sigma,A}, \mathrm{consts}(\sigma)) = [\![\sigma]\!]_{E,M}.$$

Proof. By structural induction on σ. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, the claim follows immediately by unfolding the relevant definitions (i.e. Defs. 2.77, 2.76, 2.73, and 2.60). Note that the semantics of a tree in $\mathrm{Tree}_{\mathrm{List}_{\{\mathrm{True},\mathrm{False}\}}}$ is independent of the truth assignment A.

If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then
$$\begin{aligned}
\mathrm{map}([\![\,\cdot\,]\!]_{\sigma,A}, \mathrm{consts}(\sigma)) &\overset{2.77}{=} \mathrm{map}([\![\,\cdot\,]\!]_{\sigma,A}, \mathrm{map}(\mathrm{Node}, \mathrm{pick}(|[\![\sigma_1]\!]_{E,M}|, \mathrm{consts}(\sigma_2))))\\
&\overset{2.75}{=} \mathrm{map}([\![\,\cdot\,]\!]_{\sigma,A} \circ \mathrm{Node}, \mathrm{pick}(|[\![\sigma_1]\!]_{E,M}|, \mathrm{consts}(\sigma_2)))\\
&\overset{2.48}{=} \mathrm{map}([\![\,\cdot\,]\!]_{\sigma,A} \circ \mathrm{Node}, [\![\sigma_1]\!]_{E,M} \to \mathrm{consts}(\sigma_2))\\
&\overset{2.60}{=} [\![\sigma_1]\!]_{E,M} \to \mathrm{map}([\![\,\cdot\,]\!]_{\sigma_2,A}, \mathrm{consts}(\sigma_2))\\
&\overset{\mathrm{IH}}{=} [\![\sigma_1]\!]_{E,M} \to [\![\sigma_2]\!]_{E,M}\\
&\overset{2.14}{=} [\![\sigma]\!]_{E,M},
\end{aligned}$$

where we again identify functions over the (finite and totally ordered) domain $[\![\sigma_1]\!]_{E,M}$ with a list of their values. □

Lemma 2.80 shows that consts(σ) enumerates trees corresponding to the elements of $[\![\sigma]\!]_{E,M}$ in the correct order. Moreover, since a constant tree only carries propositional unit vectors as leaf labels, any truth assignment is well-formed wrt. the tree.

Lemma 2.81. *Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Furthermore, let $\sigma \in \text{Types}_\Omega$, let t be a constant tree for σ, and let A be an arbitrary truth assignment. Then A is well-formed wrt. t.*

Proof. This is an immediate consequence of Lemma 2.80 with Lemma 2.66. □

Alternatively, it can easily be seen directly also, by structural induction on σ.

Application

Functions of arity n are represented by trees of height $n + 1$. Intuitively, when a function is applied to the i-th element of its domain, the result is given by the i-th subtree of the tree representing the function. As mentioned earlier, the function may be higher-order, i.e. its argument may be a function again. In this case the argument is itself represented by a tree of height > 1.

We define a function enum that tells us if a tree denotes the i-th element of its corresponding type. More precisely, enum(t) returns a list of propositional formulae, where the i-th formula of the list evaluates to $\top$ iff the tree t denotes the i-th element of its corresponding type.

If t is a tree representing a function, we want to employ pick (Def. 2.46) to define enum. A minor complication in this case is caused by the fact that pick(n, Y) returns (an enumeration of) Y^n, while for the definition of enum, we need a more general function pick$'$ with pick$'([Y_1, \ldots, Y_n]) = Y_1 \times \cdots \times Y_n$ (where $Y_1 \times \cdots \times Y_n$ is again equipped with the lexicographic order obtained from the individual orders on Y_1, …, Y_n). We define pick$'$ first.

Definition 2.82 (pick$'$)**.** Let $n \geq 1$, and let Y_i (for $1 \leq i \leq n$) be finite, non-empty and totally ordered, with $Y_1 = [y_1, \ldots, y_m]$ (for some $m \geq 1$). Define

$$\text{pick}'([Y_1]) := [[y_1], \ldots, [y_m]],$$

and for $n > 1$ define

$$\text{pick}'([Y_1, Y_2, \ldots, Y_n]) := [y_1 \# f_1, \ldots, y_1 \# f_k, \ \ldots, \ y_m \# f_1, \ldots, y_m \# f_k],$$

where $[f_1, \ldots, f_k] = \text{pick}'([Y_2, \ldots, Y_n])$.

Remark 2.83. For $n \geq 1$ and Y_i (for $1 \leq i \leq n$) finite, non-empty and totally ordered, pick$'([Y_1, \ldots, Y_n])$ is a list in $\text{List}_{\text{List}_Y}$ (where $Y := \bigcup_{i=1}^n Y_i$) of length $\prod_{i=1}^n |Y_i|$, and each list element (which is again a list) has length n.

Proof. By induction on n. □

Lemma 2.84. *Let $n \geq 1$, and let Y_i (for $1 \leq i \leq n$) finite, non-empty and totally ordered. Then*

$$\mathrm{pick}'([Y_1, \ldots, Y_n]) = Y_1 \times \cdots \times Y_m,$$

where $Y_1 \times \cdots \times Y_m$ is equipped with the lexicographic order.

Proof. By induction on n, using Def. 2.42 (adapted for cartesian products), and Def. 2.82 of course. □

Another auxiliary function $\bigwedge$, which returns the conjunction of a (non-empty) list of formulae, is needed as well.

Definition 2.85 ($\bigwedge$)**.** For $\varphi_1, \ldots, \varphi_n \in \mathbb{P}$, $n \geq 1$, define

$$\bigwedge([\varphi_1, \ldots, \varphi_n]) := \varphi_1 \wedge \cdots \wedge \varphi_n.$$

The definition of enum follows.

Definition 2.86 (enum)**.** Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\sigma \in \mathrm{Types}_\Omega$, and let $t \in \mathrm{Trees}(\sigma)$. Then $\mathrm{enum}(t)$ is defined as follows:

1. If $t = \mathrm{Leaf}([\varphi_1, \ldots, \varphi_k])$ for some $\varphi_1, \ldots, \varphi_k \in \mathbb{P}$, then
$$\mathrm{enum}(t) := [\varphi_1, \ldots, \varphi_k].$$
2. If $t = \mathrm{Node}([t_1, \ldots, t_k])$ for some $t_1, \ldots, t_k \in \mathrm{Trees}(\sigma')$ (where $\sigma' \in \mathrm{Types}_\Omega$), then
$$\mathrm{enum}(t) := \mathrm{map}\big(\textstyle\bigwedge, \mathrm{pick}'(\mathrm{map}(\mathrm{enum}, [t_1, \ldots, t_k]))\big).$$

Remark 2.87. Let $t \in \mathrm{Trees}(\sigma)$. Then $\mathrm{enum}(t) \in \mathrm{List}_\mathbb{P}$ is a list of length $|[\![\sigma]\!]_{E,M}|$.

Proof. By structural induction on σ. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, the claim follows immediately from Def. 2.58.

If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then by the induction hypothesis $\mathrm{enum}(t_i) \in \mathrm{List}_\mathbb{P}$ (for $1 \leq i \leq k$, where $k = |[\![\sigma_1]\!]_{E,M}|$ by Def. 2.58) is a list of length $|[\![\sigma_2]\!]_{E,M}|$. Hence $\mathrm{map}(\mathrm{enum}, [t_1, \ldots, t_k]) \in \mathrm{List}_{\mathrm{List}_\mathbb{P}}$ is a list of length k by Remark 2.74. Now Remark 2.83 implies that $\mathrm{pick}'(\mathrm{map}(\mathrm{enum}, [t_1, \ldots, t_k])) \in \mathrm{List}_{\mathrm{List}_\mathbb{P}}$ is a list of length $|[\![\sigma_2]\!]_{E,M}|^k$, and from this $\mathrm{enum}(t) \in \mathrm{List}_\mathbb{P}$ follows with Remark 2.74, while Remark 2.39 shows that the length is equal to $|[\![\sigma]\!]_{E,M}|$. □

The next lemma shows that enum indeed builds the desired list of formulae, where the i-th formula is true iff the corresponding tree denotes the i-th element of its type.

Lemma 2.88. *Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\sigma \in \mathrm{Types}_\Omega$, and let $t \in \mathrm{Trees}(\sigma)$. Assume $\mathrm{enum}(t) = [\varphi_1, \ldots, \varphi_k]$, and $[\![\sigma]\!]_{E,M} = [d_1, \ldots, d_k]$ (where $k = |[\![\sigma]\!]_{E,M}|$). Let A be a truth assignment that is well-formed wrt. t. Then, for $1 \leq i \leq k$,*

$$[\![\varphi_i]\!]_A = \top \quad \textit{iff} \quad [\![t]\!]_{\sigma,A} = d_i.$$

Proof. By structural induction on σ. If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, the "$\Leftarrow$" direction of the equivalence follows immediately from Def. 2.60, while the "$\Rightarrow$" direction uses the well-formedness of A (Def. 2.62).

If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then $t = \mathrm{Node}([t_1, \ldots, t_k])$ for some $t_1, \ldots, t_k \in \mathrm{Trees}(\sigma_2)$, where $k = |[\![\sigma_1]\!]_{E,M}|$. For $1 \le i \le k$, let $\mathrm{enum}(t_i) = [\varphi_1^i, \ldots, \varphi_l^i]$ for some $\varphi_1^i, \ldots, \varphi_l^i \in \mathbb{P}$ (where $l = |[\![\sigma_2]\!]_{E,M}|$). In this case,

$$\begin{aligned}
\mathrm{enum}(t) &\overset{2.86}{=} \mathrm{map}\big(\textstyle\bigwedge, \mathrm{pick}'(\mathrm{map}(\mathrm{enum}, [t_1, \ldots, t_k]))\big) \\
&\overset{2.73}{=} \mathrm{map}\big(\textstyle\bigwedge, \mathrm{pick}'([\mathrm{enum}(t_1), \ldots, \mathrm{enum}(t_k)])\big) \\
&\overset{2.84}{=} \mathrm{map}\big(\textstyle\bigwedge, \mathrm{enum}(t_1) \times \cdots \times \mathrm{enum}(t_k)\big) \\
&\overset{2.42}{=} \mathrm{map}\big(\textstyle\bigwedge, [[\varphi_1^1, \ldots, \varphi_1^k], \ldots, [\varphi_l^1, \ldots, \varphi_l^k]]\big) \\
&\overset{2.73}{=} [\textstyle\bigwedge([\varphi_1^1, \ldots, \varphi_1^k]), \ldots, \bigwedge([\varphi_l^1, \ldots, \varphi_l^k])] \\
&\overset{2.85}{=} [\varphi_1^1 \wedge \cdots \wedge \varphi_1^k, \ldots, \varphi_l^1 \wedge \cdots \wedge \varphi_l^k],
\end{aligned}$$

hence the lemma follows with Def. 2.42 and the induction hypothesis. □

To define the application of one tree to another, we need further auxiliary functions. An analogue of the map function for trees, $\mathrm{treemap}(f, t)$, returns the tree that results from application of f to every element of every leaf of t. $\mathrm{merge}(g, t_1, t_2)$ applies a binary function g to corresponding leaf elements in two trees t_1 and t_2 of the same shape. (Note that we will need treemap and merge only for trees that are labeled with lists. The following definitions are therefore adapted to this special case. They can easily be generalized to trees with arbitrary labels, but this would require the use of $\mathrm{treemap}(\mathrm{map}(f, \cdot), t)$ in place of $\mathrm{treemap}(f, t)$—likewise for merge.)

Definition 2.89 (treemap)**.** Let X, Y be sets. For $f\colon X \to Y$ and $t \in \mathrm{Tree}_{\mathrm{List}_X}$, define $\mathrm{treemap}(f, t)$ as follows:

1. If $t = \mathrm{Leaf}([x_1, \ldots, x_n])$ for some $x_1, \ldots, x_n \in X$, then
$$\mathrm{treemap}(f, t) := \mathrm{Leaf}([f(x_1), \ldots, f(x_n)]).$$
2. If $t = \mathrm{Node}([t_1, \ldots, t_n])$ for some $t_1, \ldots, t_n \in \mathrm{Tree}_{\mathrm{List}_X}$, then
$$\mathrm{treemap}(f, t) := \mathrm{Node}([\mathrm{treemap}(f, t_1), \ldots, \mathrm{treemap}(f, t_n)]).$$

Remark 2.90. Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\sigma \in \mathrm{Types}_\Omega$. Let $f\colon \mathbb{P} \to \mathbb{P}$ and $t \in \mathrm{Trees}(\sigma)$. Then $\mathrm{treemap}(f, t) \in \mathrm{Trees}(\sigma)$.

Proof. By structural induction on σ. □

Definition 2.91 (merge)**.** Let X, Y be sets. For $f\colon X \times X \to Y$ and $t_1, t_2 \in \mathrm{Tree}_{\mathrm{List}_X}$, define $\mathrm{merge}(f, t_1, t_2)$ as follows:

1. If $t_1 = \text{Leaf}([a_1, \ldots, a_n])$ and $t_2 = \text{Leaf}([b_1, \ldots, b_n])$ for some $a_1, \ldots, a_n, b_1, \ldots, b_n \in X$, then
$$\text{merge}(f, t_1, t_2) := \text{Leaf}([f(a_1, b_1), \ldots, f(a_n, b_n)]).$$
2. If $t_1 = \text{Node}([u_1, \ldots, u_n])$ and $t_2 = \text{Node}([v_1, \ldots, v_n])$ for some $u_1, \ldots, u_n, v_1, \ldots, v_n \in \text{Tree}_{\text{List}_X}$, then
$$\text{merge}(f, t_1, t_2) := \text{Node}([\text{merge}(f, u_1, v_1), \ldots, \text{merge}(f, u_n, v_n)]).$$
3. Otherwise, $\text{merge}(f, t_1, t_2)$ is undefined.

We extend the definition of merge to any non-empty list $[t_1, \ldots, t_n]$ of trees in $\text{Trees}(\sigma)$ by defining $\text{merge}(f, [t_1]) := t_1$, and $\text{merge}(f, [t_1, t_2, \ldots, t_n]) := f(t_1, \text{merge}(f, [t_2, \ldots, t_n]))$ (provided f has type $X \times X \to X$, for some set X).

Remark 2.92. Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\sigma \in \text{Types}_\Omega$. For $f \colon \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ and $t_1, t_2 \in \text{Trees}(\sigma)$, $\text{merge}(f, t_1, t_2)$ is (defined and) in $\text{Trees}(\sigma)$. Furthermore, for $t_1, \ldots, t_n \in \text{Trees}(\sigma)$, $n \geq 1$, $\text{merge}(f, [t_1, \ldots, t_n])$ is (defined and) in $\text{Trees}(\sigma)$.

Proof. The first claim follows by structural induction on σ, while the second claim follows by structural induction on the (non-empty) list $[t_1, \ldots, t_n]$. □

Finally we can define a translation function apply which corresponds to function application. $\text{apply}(t, u)$, where t is a tree representing a function, and u is a tree for the function's argument, is a tree that encodes the value of the function when applied to this specific argument. The tree's leafs are labeled with propositional formulae that simulate selection of the correct subtree of t, based on the value denoted by the argument tree u.

Definition 2.93 (apply)**.** Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\sigma', \sigma \in \text{Types}_\Omega$. Let $t \in \text{Trees}(\sigma' \to \sigma)$ and $u \in \text{Trees}(\sigma')$. Assume $t = \text{Node}([t_1, \ldots, t_k])$ with $t_1, \ldots, t_k \in \text{Trees}(\sigma)$, and $\text{enum}(u) = [\varphi_1, \ldots, \varphi_k]$ (where $k = |[\![\sigma']\!]_{E,M}|$). Define $\text{apply}(t, u)$ as follows:
$$\text{apply}(t, u) := \text{merge}(\vee, [\text{treemap}((\varphi_1 \wedge \cdot), t_1), \ldots, \text{treemap}((\varphi_k \wedge \cdot), t_k)]).$$

Remark 2.94. Let $t \in \text{Trees}(\sigma' \to \sigma)$ and $u \in \text{Trees}(\sigma')$. Then $\text{apply}(t, u) \in \text{Trees}(\sigma)$.

Proof. Immediate, using Remark 2.92 and Remark 2.90. □

The following lemma shows that the meaning of apply is indeed that of function application.

Lemma 2.95. *Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\sigma', \sigma \in \text{Types}_\Omega$. Let $t \in \text{Trees}(\sigma' \to \sigma)$ and $u \in \text{Trees}(\sigma')$. Let A be a truth assignment that is well-formed wrt. t and u. Then*
$$[\![\text{apply}(t, u)]\!]_{\sigma,A} = [\![t]\!]_{\sigma' \to \sigma, A}([\![u]\!]_{\sigma', A}).$$

Proof. Assume $t = \mathrm{Node}([t_1, \ldots, t_k])$ with $t_1, \ldots, t_k \in \mathrm{Trees}(\sigma)$, and $\mathrm{enum}(u) = [\varphi_1, \ldots, \varphi_k]$ (where $k = |[\![\sigma']\!]_{E,M}|$). Furthermore, assume $[\![\sigma']\!]_{E,M} = [d_1, \ldots, d_k]$, and $[\![u]\!]_{\sigma',A} = d_j$ (for some $1 \leq j \leq k$). Then $[\![\varphi_j]\!]_A = \top$, and for each $1 \leq i \leq k$ with $i \neq j$, $[\![\varphi_i]\!]_A = \bot$ (by Lemma 2.88). The proof is by structural induction on σ.

If $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$, then (for each $1 \leq i \leq k$) $t_i = \mathrm{Leaf}([x_1^i, \ldots, x_l^i])$ for some $x_1^i, \ldots, x_l^i \in \mathbb{P}$, where $l = |[\![\sigma]\!]_{E,M}|$. In this case,

$$
\begin{aligned}
[\![\mathrm{apply}(t,u)]\!]_{\sigma,A} &\overset{2.93}{=} [\![\mathrm{merge}(\vee, [\mathrm{treemap}((\varphi_1 \wedge \cdot), t_1), \ldots, \mathrm{treemap}((\varphi_k \wedge \cdot), t_k)])]\!]_{\sigma,A} \\
&\overset{2.89}{=} [\![\mathrm{merge}(\vee, \\
&\qquad [\mathrm{Leaf}([\varphi_1 \wedge x_1^1, \ldots, \varphi_1 \wedge x_l^1]), \ldots, \mathrm{Leaf}([\varphi_k \wedge x_1^k, \ldots, \varphi_k \wedge x_l^k])])]\!]_{\sigma,A} \\
&\overset{2.91}{=} [\![\mathrm{Leaf}([\bigvee_{i=1}^{k} \varphi_i \wedge x_1^i, \ldots, \bigvee_{i=1}^{k} \varphi_i \wedge x_l^i])]\!]_{\sigma,A} \\
&\overset{2.60}{=} [\![\mathrm{Leaf}([x_1^j, \ldots, x_l^j])]\!]_{\sigma,A} \\
&= [\![t_j]\!]_{\sigma,A} \\
&\overset{2.60}{=} [\![\mathrm{Node}([t_1, \ldots, t_k])]\!]_{\sigma' \to \sigma,A}(d_j) \\
&= [\![t]\!]_{\sigma' \to \sigma,A}([\![u]\!]_{\sigma',A}).
\end{aligned}
$$

If $\sigma = \sigma_1 \to \sigma_2$ with $\sigma_1, \sigma_2 \in \mathrm{Types}_\Omega$, then (for each $1 \leq i \leq k$) $t_i = \mathrm{Node}([t_1^i, \ldots, t_l^i])$ for some $t_1^i, \ldots, t_l^i \in \mathrm{Trees}(\sigma_2)$, where $l = |[\![\sigma_1]\!]_{E,M}|$. In this case,

$$
\begin{aligned}
[\![\mathrm{apply}(t,u)]\!]_{\sigma,A} &\overset{2.93}{=} [\![\mathrm{merge}(\vee, [\mathrm{treemap}((\varphi_1 \wedge \cdot), t_1), \ldots, \mathrm{treemap}((\varphi_k \wedge \cdot), t_k)])]\!]_{\sigma,A} \\
&\overset{2.89}{=} [\![\mathrm{merge}(\vee, [\mathrm{Node}([\mathrm{treemap}((\varphi_1 \wedge \cdot), t_1^1), \ldots, \mathrm{treemap}((\varphi_1 \wedge \cdot), t_l^1)]), \ldots, \\
&\qquad \mathrm{Node}([\mathrm{treemap}((\varphi_k \wedge \cdot), t_1^k), \ldots, \mathrm{treemap}((\varphi_k \wedge \cdot), t_l^k)])])]\!]_{\sigma,A} \\
&\overset{2.91}{=} [\![\mathrm{Node}([\mathrm{merge}(\vee, [\mathrm{treemap}((\varphi_1 \wedge \cdot), t_1^1), \ldots, \mathrm{treemap}((\varphi_k \wedge \cdot), t_1^k)]), \ldots, \\
&\qquad \mathrm{merge}(\vee, [\mathrm{treemap}((\varphi_1 \wedge \cdot), t_l^1), \ldots, \mathrm{treemap}((\varphi_k \wedge \cdot), t_l^k)])])]\!]_{\sigma,A} \\
&\overset{2.93}{=} [\![\mathrm{Node}([\mathrm{apply}(\mathrm{Node}([t_1^1, \ldots, t_1^k]), u), \ldots, \\
&\qquad \mathrm{apply}(\mathrm{Node}([t_l^1, \ldots, t_l^k]), u)])]\!]_{\sigma,A} \\
&\overset{\mathrm{IH}}{=} [\![\mathrm{Node}([t_1^j, \ldots, t_l^j])]\!]_{\sigma,A} \\
&= [\![t_j]\!]_{\sigma,A} \\
&\overset{2.60}{=} [\![\mathrm{Node}([t_1, \ldots, t_k])]\!]_{\sigma' \to \sigma,A}(d_j) \\
&= [\![t]\!]_{\sigma' \to \sigma,A}([\![u]\!]_{\sigma',A}).
\end{aligned}
$$

□

In particular, $[\![\mathrm{apply}(t,u)]\!]_{\sigma,A} \in [\![\sigma]\!]_{E,M}$. Thus every truth assignment that is well-formed wrt. t and u is also well-formed wrt. $\mathrm{apply}(t,u)$.

Lemma 2.96. *Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let σ', $\sigma \in \mathrm{Types}_\Omega$. Let $t \in \mathrm{Trees}(\sigma' \to \sigma)$*

and $u \in \text{Trees}(\sigma')$*. Let* A *be a truth assignment that is well-formed wrt.* t *and* u*. Then* A *is well-formed wrt.* $\text{apply}(t, u)$*.*

Proof. This is an immediate consequence of Lemma 2.95 and Lemma 2.66. □

Alternatively, and similar to Lemma 2.81, it can be proved directly also, by structural induction on σ.

Translation to Trees

Having considered variables, constants, λ-abstraction and application, we are now ready to define the translation $\mathcal{T}$ from terms to trees of propositional formulae. The translation is parameterized by a tree assignment $\bar{T}$, which is updated when the translation descends into the body of a λ-abstraction to give the tree for the bound variable. (Isabelle internally uses de Bruijn indices [47] to represent bound variables, so in the actual implementation of our translation, a mapping from indices—rather than variables—to trees is extended every time a λ is encountered. Terms with de Bruijn indices however are not particularly easy to read for humans, and for the sake of clarity, we have instead chosen to use terms with variable names in this presentation.)

Definition 2.97 (Translation from Terms to Trees)**.** Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). The *translation* $\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)$ *of a term* $t_\sigma \in \text{Terms}_\Sigma$ *wrt.* $\bar{T}$ *and* $\bar{M}$ is defined as follows:

1. If $t \in \text{Vars}$, then $\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) := \bar{T}(t_\sigma)$.
2. If $t \in \text{Names}$, then $\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) := \bar{M}(t_\sigma)$.
3. If $t_\sigma = (t^1_{\sigma'\to\sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma'\to\sigma}$, $t^2_{\sigma'} \in \text{Terms}_\Sigma$, then
$$\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) := \text{apply}(\mathcal{T}_{\bar{T},\bar{M}}(t^1_{\sigma'\to\sigma}), \mathcal{T}_{\bar{T},\bar{M}}(t^2_{\sigma'})).$$
4. If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1\to\sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, then
$$\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) := \text{Node}([t_1, \ldots, t_n]),$$
where $n = |[\![\sigma_1]\!]_{E,M}|$, $\text{consts}(\sigma_1) = [c_1, \ldots, c_n]$, and (for $1 \le i \le n$)
$$t_i := \mathcal{T}_{\bar{T}[x_{\sigma_1}\mapsto c_i],\bar{M}}(t'_{\sigma_2}).$$

We first prove that the translation of a term t_σ is an element of $\text{Trees}(\sigma)$.

Remark 2.98. Let $t_\sigma \in \text{Terms}_\Sigma$. Then $\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) \in \text{Trees}(\sigma)$.

Proof. The proof is by structural induction on t_σ. If $t \in \text{Vars}$ or $t \in \text{Names}$, the claim follows immediately from Def. 2.67 and Def. 2.68, respectively.

If $t_\sigma = (t^1_{\sigma' \to \sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma' \to \sigma}$, $t^2_{\sigma'} \in \text{Terms}_\Sigma$, then $\mathcal{T}_{\bar{T},\bar{M}}(t^1_{\sigma' \to \sigma}) \in \text{Trees}(\sigma' \to \sigma)$ and $\mathcal{T}_{\bar{T},\bar{M}}(t^2_{\sigma'}) \in \text{Trees}(\sigma')$ by the induction hypothesis. Hence $\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) \in \text{Trees}(\sigma)$ by Remark 2.94.

If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1 \to \sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, then $\bar{T}[x_{\sigma_1} \mapsto c_i]$ (for $1 \leq i \leq n$, where $n = |[\![\sigma_1]\!]_{E,M}|$ and $\text{consts}(\sigma_i) = [c_1, \ldots, c_n]$) is a tree assignment for Σ (wrt. E and M) due to Remark 2.79. Hence each $\mathcal{T}_{\bar{T}[x_{\sigma_1} \mapsto c_i],\bar{M}}(t'_{\sigma_2})$ (for $1 \leq i \leq n$) is in $\text{Trees}(\sigma_2)$ by the induction hypothesis. Thus $\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma) \in \text{Trees}(\sigma_1 \to \sigma_2)$ (Def. 2.58). □

Next we show that the translation preserves the meaning of terms (wrt. their HOL semantics, while the meaning of the translation result is given by its tree semantics).

Theorem 2.99. *Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\mathcal{A}$ be a variable assignment and $\mathcal{M}$ be a term model for Σ. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). Let $t_\sigma \in \text{Terms}_\Sigma$. Suppose A is a truth assignment such that $[\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A} = \mathcal{A}(x_{\sigma'})$ for every $x_{\sigma'} \in \text{FreeVars}(t_\sigma)$, and $[\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A} = \mathcal{M}(c_{\sigma'})$ for every $c_{\sigma'} \in \text{Names}(t_\sigma)$. Then*

$$[\![\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)]\!]_{\sigma,A} = [\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}}.$$

Proof. The proof is by structural induction on t_σ. If $t = x \in \text{Vars}$ or $t = c \in \text{Names}$, the claim follows immediately from $[\![\bar{T}(x_\sigma)]\!]_{\sigma,A} = \mathcal{A}(x_\sigma)$ or $[\![\bar{M}(c_\sigma)]\!]_{\sigma,A} = \mathcal{M}(c_\sigma)$, respectively.

If $t_\sigma = (t^1_{\sigma' \to \sigma}\, t^2_{\sigma'})_\sigma$ for some $t^1_{\sigma' \to \sigma}$, $t^2_{\sigma'} \in \text{Terms}_\Sigma$, then $[\![\mathcal{T}_{\bar{T},\bar{M}}(t^1_{\sigma' \to \sigma})]\!]_{\sigma' \to \sigma,A} = [\![t^1_{\sigma' \to \sigma}]\!]_{\mathcal{A},\mathcal{M}}$ and $[\![\mathcal{T}_{\bar{T},\bar{M}}(t^2_{\sigma'})]\!]_{\sigma',A} = [\![t^2_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}}$ by the induction hypothesis. Hence

$$[\![\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)]\!]_{\sigma,A} = [\![t^1_{\sigma' \to \sigma}]\!]_{\mathcal{A},\mathcal{M}}([\![t^2_{\sigma'}]\!]_{\mathcal{A},\mathcal{M}})$$

by Lemma 2.95.

If $t_\sigma = (\lambda x_{\sigma_1}.\, t'_{\sigma_2})_{\sigma_1 \to \sigma_2}$ for some $x \in \text{Vars}$, $\sigma_1 \in \text{Types}_\Omega$ and $t'_{\sigma_2} \in \text{Terms}_\Sigma$, then $[\![\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)]\!]_{\sigma,A}$ is defined as the function that sends each $d_i \in [\![\sigma_1]\!]_{E,M}$ (for $1 \leq i \leq n$, where $n = |[\![\sigma_1]\!]_{E,M}|$, $[\![\sigma_1]\!]_{E,M} = [d_1, \ldots, d_n]$, and $\text{consts}(\sigma_i) = [c_1, \ldots, c_n]$) to $[\![\mathcal{T}_{\bar{T}[x_{\sigma_1} \mapsto c_i],\bar{M}}(t'_{\sigma_2})]\!]_{\sigma_2,A}$, which is equal to $[\![t'_{\sigma_2}]\!]_{\mathcal{A}[x_{\sigma_1} \mapsto d_i],\mathcal{M}}$ by the induction hypothesis and Lemma 2.80. □

We now state the main result of this section: there is a well-formed truth assignment under which the tree that results from translating t_σ denotes $d \in [\![\sigma]\!]_{E,M}$ if and only if there exist a variable assignment $\mathcal{A}$ and a standard term model $\mathcal{M}$ such that $[\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}} = d$.

Theorem 2.100. *Let $\Sigma = (\text{Vars}, \text{Names}, \text{Typ})$ be a standard signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). Assume that $\bar{T}$ and $\bar{M}$ are standard (wrt. each other). Let $t_\sigma \in \text{Terms}_\Sigma$, and let $d \in [\![\sigma]\!]_{E,M}$.*

There exist a variable assignment $\mathcal{A}$ and a standard term model $\mathcal{M}$ (for Σ) such that

$$[\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}} = d$$

iff there exists a truth assignment A that is well-formed wrt. $\bar{T}(\text{FreeVars}(t_\sigma)) \cup \bar{M}(\text{Names}(t_\sigma))$ such that

$$[\![\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)]\!]_{\sigma,A} = d.$$

Proof. Assume that $\mathcal{A}$ is a variable assignment and $\mathcal{M}$ is a standard term model for Σ such that $[\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}} = d$. Then by Lemma 2.71, there exists a truth assignment A that is well-formed wrt. $\bar{T}(\text{FreeVars}(t_\sigma)) \cup \bar{M}(\text{Names}(t_\sigma))$ such that $[\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A} = \mathcal{A}(x_{\sigma'})$ for every $x_{\sigma'} \in \text{FreeVars}(t_\sigma)$, and $[\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A} = \mathcal{M}(c_{\sigma'})$ for every $c_{\sigma'} \in \text{Names}(t_\sigma)$. Hence $[\![\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)]\!]_{\sigma,A} = d$ by Theorem 2.99.

For the other direction of the equivalence, assume that A is a truth assignment that is well-formed wrt. $\bar{T}(\text{FreeVars}(t_\sigma)) \cup \bar{M}(\text{Names}(t_\sigma))$ such that $[\![\mathcal{T}_{\bar{T},\bar{M}}(t_\sigma)]\!]_{\sigma,A} = d$. Then—again by Lemma 2.71—there exist a variable assignment $\mathcal{A}$ and a standard term model $\mathcal{M}$ (for Σ) such that $[\![\bar{T}(x_{\sigma'})]\!]_{\sigma',A} = \mathcal{A}(x_{\sigma'})$ for every $x_{\sigma'} \in \text{FreeVars}(t_\sigma)$, and $[\![\bar{M}(c_{\sigma'})]\!]_{\sigma',A} = \mathcal{M}(c_{\sigma'})$ for every $c_{\sigma'} \in \text{Names}(t_\sigma)$. Hence $[\![t_\sigma]\!]_{\mathcal{A},\mathcal{M}} = d$ by Theorem 2.99. □

As an immediate corollary, a HOL formula is satisfiable iff its translation denotes $\top$ under some well-formed truth assignment.

Corollary 2.101. *Let Σ be a standard signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). Assume that $\bar{T}$ and $\bar{M}$ are standard (wrt. each other). Let $t_{\mathsf{bool}} \in \text{Terms}_\Sigma$.*

Then t_{bool} is satisfiable wrt. E and M iff there exists a truth assignment A that is well-formed wrt. $\bar{T}(\text{FreeVars}(t_\sigma)) \cup \bar{M}(\text{Names}(t_\sigma))$ such that $[\![\mathcal{T}_{\bar{T},\bar{M}}(t_{\mathsf{bool}})]\!]_{\mathsf{bool},A} = \top$.

Proof. Recall that a formula is satisfiable wrt. E and M iff its meaning (wrt. some variable assignment and standard term model for Σ) is $\top$. Choose $\sigma = \mathsf{bool}$ and $d = \top$ in Theorem 2.100. □

Translation to Propositional Logic

$\mathcal{T}_{\bar{T},\bar{M}}(t_{\mathsf{bool}})$ is still not a propositional formula, but a tree in $\text{Trees}(\mathsf{bool})$, i.e. of the form $\text{Leaf}([\varphi_1, \varphi_2])$ for some formulae φ_1, $\varphi_2 \in \mathbb{P}$. Obtaining a single propositional formula however is a rather small final step now. By Def. 2.60 and Def. 2.41,

$$[\![\text{Leaf}([\varphi_1, \varphi_2])]\!]_{\mathsf{bool},A} = \top \quad \text{iff} \quad [\![\varphi_1]\!]_A = \top \text{ and } [\![\varphi_2]\!]_A = \bot.$$

This motivates the following definition.

Definition 2.102 (Translation from Terms to Propositional Formulae)**.** Let Ω be a standard type structure, let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let Σ be a signature over Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). The *propositional formula $\phi_{\bar{T},\bar{M}}(t_{\mathsf{bool}})$ for a term $t_{\mathsf{bool}} \in \text{Terms}_\Sigma$ wrt. $\bar{T}$ and $\bar{M}$* is defined as follows:

$$\phi_{\bar{T},\bar{M}}(t_{\mathsf{bool}}) := \varphi_1 \wedge \neg\varphi_2 \wedge \bigwedge \text{wf}(\bar{T}(\text{FreeVars}(t_{\mathsf{bool}}))) \wedge \bigwedge \text{wf}(\bar{M}(\text{Names}(t_{\mathsf{bool}})))$$

where $\mathcal{T}_{\bar{T},\bar{M}}(t_{\mathsf{bool}}) = \text{Leaf}([\varphi_1, \varphi_2])$.

A HOL formula t_{bool} is satisfiable (wrt. a fixed finite type environment and model) iff its corresponding propositional formula is satisfiable.

Corollary 2.103. *Let Σ be a signature over a standard type structure Ω. Let E be a finite type environment for Ω, and let M be a finite standard type model for Ω. Let $\bar{T}$ and $\bar{M}$ be a tree assignment and a tree model, respectively, for Σ (wrt. E and M). Assume that $\bar{T}$ and $\bar{M}$ are standard (wrt. each other). Let $t_{\mathsf{bool}} \in \mathrm{Terms}_\Sigma$.*

Then t_{bool} is satisfiable wrt. E and M iff $\phi_{\bar{T},\bar{M}}(t_{\mathsf{bool}})$ is (propositionally) satisfiable.

Proof. This is a direct consequence of Corollary 2.101 and Lemma 2.65. □

We have thus defined (and proved correct) a satisfiability-equivalent translation from HOL to propositional logic, which is exactly what we had set out to do at the beginning of this section.

2.3.4 Examples

As an example, consider the formula

$$t_{\mathsf{bool}} := ((\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha} =_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to\mathsf{bool}} y_{\alpha\to\alpha})_{\mathsf{bool}}$$

(with $=$ written in infix notation). Its only type variable is α, and its only free variable is $y_{\alpha\to\alpha}$. Assume E is a type environment with $E(\alpha) = [a_1, a_2]$, and let M be an arbitrary standard type model (hence $|[\![\alpha\to\alpha]\!]_{E,M}| = 2^2 = 4$). A possible standard tree assignment $\bar{T}$ is given by

$$\bar{T}(y_{\alpha\to\alpha}) := \mathrm{Node}([\mathrm{Leaf}([y_1, y_2]), \mathrm{Leaf}([y_3, y_4])]),$$

where y_1, y_2, y_3, y_4 are four distinct Boolean variables. Furthermore, let $\bar{M}$ be an arbitrary standard tree model.

The subterms of t_{bool} are then translated into the following trees:

$$\begin{aligned}
\mathcal{T}_{\bar{T},\bar{M}}((\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha}) &= \mathrm{Node}([\mathrm{Leaf}([\mathrm{True}, \mathrm{False}]), \mathrm{Leaf}([\mathrm{False}, \mathrm{True}])]),\\
\mathcal{T}_{\bar{T},\bar{M}}(=_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to\mathsf{bool}}) &= \mathrm{Node}([\mathrm{Node}(\mathrm{UV}_1^4), \ldots, \mathrm{Node}(\mathrm{UV}_4^4)]),\\
\mathcal{T}_{\bar{T},\bar{M}}(y_{\alpha\to\alpha}) &= \mathrm{Node}([\mathrm{Leaf}([y_1, y_2]), \mathrm{Leaf}([y_3, y_4])]).
\end{aligned}$$

Using the translation rule for application (and simplifying the resulting formulae), we thus have

$$\mathcal{T}_{\bar{T},\bar{M}}(=_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to\mathsf{bool}}\ (\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha}) = \mathrm{Node}([\mathrm{Bot}, \mathrm{Top}, \mathrm{Bot}, \mathrm{Bot}])$$

(where the position of Top reflects that the identity function, due to the use of the lexicographic order, is the second element of the function space $[\![\alpha\to\alpha]\!]_{E,M}$) and

$$\mathcal{T}_{\bar{T},\bar{M}}(t_{\mathsf{bool}}) = \mathrm{Leaf}([y_1 \wedge y_4, (y_1 \wedge y_3) \vee (y_2 \wedge y_3) \vee (y_2 \wedge y_4)]).$$

Additionally two well-formedness formulae are constructed for $\bar{T}(y_{\alpha\to\alpha})$, namely

$$\mathrm{wf}([y_1, y_2]) = \neg y_1 \vee \neg y_2$$

Property/Formula	Countermodel
"Every function that is onto is invertible."	$E(\alpha) = \{a_1, a_2\}, E(\beta) = \{b_1\}$
$(\forall y.\, \exists x.\, f\, x = y) \implies (\exists g.\, \forall x.\, g\, (f\, x) = x)$	$f = \{(a_1, b_1), (a_2, b_1)\}$
"There exists a unique choice function."	$E(\alpha) = \{a_1\}, E(\beta) = \{b_1, b_2\}$
$(\forall x.\, \exists y.\, P\, x\, y) \implies (\exists! f.\, \forall x.\, P\, x\, (f\, x))$	$P = \{(a_1, \{(b_1, \top), (b_2, \top)\})\}$
"The transitive closure of $A \cap B$ is equal to the intersection of the transitive closures of A and B."	$E(\alpha) = \{a_1, a_2\}$ $A = \{(a_1, a_2), (a_2, a_1), (a_2, a_2)\}$ $B = \{(a_1, a_1), (a_2, a_1), (a_2, a_2)\}$

Table 2.1: Refutable HOL formulae (examples)

and

$$\mathrm{wf}([y_3, y_4]) = \neg y_3 \vee \neg y_4.$$

Hence the only satisfying assignment for $\phi_{\bar{T},\bar{M}}(t_{\mathsf{bool}})$ is given by $A(y_1) := A(y_4) := \top$, $A(y_2) := A(y_3) := \bot$. This assignment corresponds to an interpretation of $y_{\alpha\to\alpha}$ as the function that maps a_1 to a_1 and a_2 to a_2 (i.e. as the identity function on $\{a_1, a_2\}$, which is of course just what the original formula states: namely that $y_{\alpha\to\alpha}$ is equal to the identity function). On the other hand, there are three well-formed falsifying assignments; e.g. $A'(y_1) := A'(y_3) := \top$, $A'(y_2) := A'(y_4) := \bot$. This particular assignment corresponds to an interpretation of $y_{\alpha\to\alpha}$ as the function that maps both a_1 and a_2 to a_1.

Table 2.1 shows a few examples of formulae for which our algorithm can automatically find a countermodel. Type annotations are suppressed, and functions in the countermodel are given by their graphs. "$\exists!$" denotes unique existence, defined as usual:

$$(\exists! x.\, P\, x) := (\exists x.\, P\, x \wedge (\forall y.\, P\, y \implies y = x)).$$

The countermodels are rather small, and were all found within a few milliseconds on a current personal computer. The main purpose of these examples is to illustrate the expressive power of the underlying logic. Some larger case studies are discussed in Chapter 4.

2.4 Model Generation

Translating the HOL input formula to propositional logic, while crucial, is only the first part of the task that the model generation algorithm must accomplish. Next, a satisfying assignment for the resulting propositional formula must be found, and this assignment must be translated back into a HOL model, which is then displayed to the user of the Isabelle system. If no satisfying assignment can be found, the translation is repeated for larger types.

2.4.1 Finding a Satisfying Assignment

Satisfiability of the resulting propositional formula can be tested with an off-the-shelf SAT solver. To this end translations into DIMACS SAT and DIMACS CNF format [50] have been implemented. The translation into SAT format is trivial, whereas CNF format (supported by zChaff [119], BerkMin [59] and other state-of-the-art solvers) requires the Boolean formula to

be in conjunctive normal form. We translate into definitional CNF [161] to avoid an exponential blowup at this stage, introducing auxiliary Boolean variables where necessary. A more sophisticated CNF conversion might further enhance the performance of our approach [83].

Isabelle/HOL runs on a number of different platforms, and installation should be as simple as possible. Therefore we have also implemented a naive DPLL-based [45, 178] SAT solver in Isabelle. This solver is not meant to replace the external solver for serious applications, but it has proved to be efficient enough for small examples. Hence it allows users to experiment with the countermodel generation without them having to worry about the installation and configuration of an additional tool that is external to Isabelle/HOL.

If the SAT solver cannot find a satisfying assignment, the translation is repeated for a larger type environment and standard type model. Details of this loop are explained in the following paragraphs.

2.4.2 Type Environments and Type Models

The translation to propositional logic defined in Section 2.3 requires that we fix a finite type environment E and a finite standard type model M, at least for those—finitely many—type variables and type constructors, respectively, that occur in the typing of t_{bool}. Remember that types denote non-empty sets. Initially, we fix E and M such that each type variable and each type constructor (other than bool and $\to$) is mapped to a singleton set. If translating t_{bool} wrt. this type environment and model yields an unsatisfiable propositional formula, we proceed by employing a function that incrementally assigns larger sets to types. After the initial assignment of singleton sets, we try every assignment which maps but one type to a singleton set, and the remaining type to a two-element set. Next, every assignment is tried which maps either two types to two-element sets, or one type to a three-element set (and all remaining types to singleton sets). In this way we can enumerate all possible assignments of (finite, non-isomorphic) sets to the types that occur in t_{bool}: if there are $k \geq 1$ types in t_{bool}, and the total number of individuals (i.e. the sum of the sizes of sets assigned to these types) is $n \geq k$, then after assigning one individual to each type, there are $n - k$ individuals which can be assigned to types freely; hence there there are

$$\binom{n-1}{n-k} = \binom{n-1}{k-1} = \frac{(n-1)!}{(k-1)!(n-k)!}$$

assignments to consider. If $k = 0$, i.e. t_{bool} contains no types other than bool and $\to$, whose interpretation is fixed, then translating t_{bool} once will determine its satisfiability.

Note that it would clearly not be sufficient to assign the same size to every type. Consider the formula

$$(\exists x_\alpha\, y_\alpha.\, x_\alpha \neq y_\alpha) \wedge (\forall x_\beta\, y_\beta.\, x_\beta = y_\beta)$$

for example, which states that type α has size at least 2, while type β has size 1. This formula is obviously satisfiable wrt. a finite model, but using equinumerous sets to interpret α and β would not find this satisfying model.

On the other hand, *any* enumeration of all assignments (of positive sizes to types) would do for our purposes. Using the enumeration described above, which minimizes the sum of the sizes of all sets that are assigned to types, was merely a design choice, driven by our desire to obtain small models.

```
fun find_model (t : term) : type_environment * type_model *
  tree_assignment * tree_model * truth_assignment =
let
    V = ty_vars t
    T = ty_names t
    F = free_vars t
    N = names t
    E := singleton_type_environment V
    M := singleton_type_model T
    T̄ := tree_assignment (F, !E, !M)
    M̄ := tree_model (N, !E, !M)
    sat := sat_solver (φ_{T̄,M̄} t)
in
    while (!sat = Unsatisfiable) do
    (
        (E, M) := next_type_environment_and_model (!E, !M)
        T̄ := tree_assignment(F, !E, !M);
        M̄ := tree_model (N, !E, !M);
        sat := sat_solver (φ_{T̄,M̄} t)
    );
    (!E, !M, !T̄, !M̄, !sat)
end
```

Figure 2.1: Model generation algorithm

There also is a practical need however to consider small models first, namely performance. Translation time and memory requirements to a large extend depend on the sizes of the types involved, and assigning large sets to types may make a translation of the input formula practically infeasible. Therefore assigning the same size to every type may not be a good idea even for those input formulae for which we can (e.g. due to syntactical restrictions) guarantee that this will in theory not miss all satisfying models: it is not unlikely that the smallest satisfying model that meets the same-size property is beyond the practically feasible search space, while at the same time a practically feasible satisfying model which assigns different sizes to types may exist (but would not be found). More sophisticated analyses (e.g. from [138]) could be used to obtain bounds on the necessary size of types, but this hasn't been implemented yet.

2.4.3 The Algorithm

Figure 2.1 depicts a simplified version of the overall model generation loop in SML-style pseudo code. We know from Corollary 2.103 that this algorithm is sound and complete, i.e. that find_model t will return with a type model and satisfying truth assignment if and only if the input formula t is satisfiable wrt. a finite model, provided

- functions tree_assignment and tree_model return a standard tree assignment and tree model, respectively; and

- function next_type_environment_and_model implements an enumeration of all possible size assignments to types, as discussed in Section 2.4.2; and
- the underlying SAT solver is sound (i.e. will not claim an unsatisfiable formula to be satisfiable) and complete (i.e. will find a satisfying assignment for the input formula if the input formula is satisfiable); and
- the function is given unbounded space and time.

If t is not satisfiable wrt. any finite model, then find_model t will loop forever.

In practice, neither unbounded memory nor unbounded time are available. In fact, Isabelle is an interactive system, and the average user hardly wants to wait more than a few seconds for feedback from the (counter-)model search. Therefore several termination conditions can be specified: a minimal and maximal size for types, a limit on the number of Boolean variables to be used, and a runtime limit. As soon as either limit is exceeded, the loop terminates. In case t only contains types bool and $\rightarrow$, the SAT solver is called at most once.

These are rather simple termination conditions, and implementing them was mostly straightforward. Bounded time execution however posed a bit of a technical challenge, and therefore deserves a more detailed discussion. It is achieved via a function timeLimit of SML type time $\rightarrow$ ('a $\rightarrow$ 'b) $\rightarrow$ 'a $\rightarrow$ 'b, whose actual implementation is compiler-specific. Under SML/NJ [56], the TimeLimit structure provides the needed functionality. Under Poly/ML [99], no such structure is available. Therefore we had to implement the functionality ourselves.

There are two essentially different approaches to implementing bounded time execution: first, the function to be executed can repeatedly check in an inner loop whether it should terminate prematurely. Second, the function to be executed is executed in parallel with a monitor function, whose only purpose is to terminate the former when the specified amount of time has elapsed. Since the first (cooperative) approach has the disadvantage of relying on the worker function to be modified in a proper way to support bounded time execution, we decided to implement the second (preemptive) approach, which provides functionality similar to that of SML/NJ's TimeLimit structure.

Getting concurrent applications right is notoriously hard however, and despite our final implementation of the timeLimit function consisting of a few lines of code only, our experience is no different. First, we tried installing a handler function for the Posix.Signal.alrm signal via Signal.signal and calling Posix.Process.alarm with the desired timeout value. Several attempts to obtain a stable implementation this way failed due to bugs related to signal handling in the Poly/ML runtime implementation. Under Poly/ML 5.0, we now use the Process structure (whose process model is based on Milner's CCS [114]) to create two console processes; one which sleeps for the specified amount of time before sending a timeout flag to the result channel, and one which performs the computation of $f(x)$, sending the result to the same channel when it is available. A call to Process.receive in the main process blocks until either the timeout flag or the actual function value has been sent. Function f should not manipulate the timer used by Posix.Process.sleep.

Various bugs in the Poly/ML runtime implementation were uncovered and subsequently fixed in the course of this work. Particularly unpleasant was a race condition between process creation/termination and garbage collection, which would infrequently cause our code to produce a segmentation fault. Due to its sporadic nature, this bug required extensive testing before

it could be reproduced and tracked down. It has been fixed by David Matthews in Poly/ML Version 5.0 [98].

Still, minor problems remained: Poly/ML's scheduling algorithm did not always assign enough CPU time to the timer process, which then could not indicate a timeout when (or soon after) the specified time had elapsed. While we could use further communication between the timer and the worker process to ensure that the timer process receives *some* time from the scheduler, there was no way to guarantee that the timer process was scheduled with sufficient priority to complete its task. The latest version of Poly/ML, Version 5.1, therefore abandons the CCS-based process model in favor of a Thread structure [100] that implements the POSIX Threads standard [78]. This structure provides new primitives that allow to implement the timeLimit function without the issues mentioned above.

2.4.4 Building the HOL Model

The satisfying truth assignment returned by the SAT solver (and then by the find_model function) assigns truth values to Boolean variables that were introduced only as intermediate artifacts by the translation from HOL to propositional logic. These variables have no meaning by themselves. Hence there is little point in displaying the satisfying truth assignment to the user directly. Instead, from the truth assignment, the type environment/model, and the tree assignment/model, we should build the corresponding HOL variable assignment and term model, which then need to be rendered in a human-readable form.

This is quite straighforward, and essentially just involves computing the meaning (cf. Def. 2.60) of those trees that were assigned to the input term's variables and constants by the given tree assignment and tree model, wrt. to the truth assignment returned by the SAT solver. There is a twist to this however. Instead of mapping terms to (string representations of) semantic values, we map terms to terms again. Suppose type α is given by the set $\{a_1, \ldots, a_n\}$. We then introduce constants $a_\alpha^1, \ldots, a_\alpha^n$ as actual Isabelle terms, and map x_α to the term a_α^i (for some $1 \leq i \leq n$). Likewise, variables of type bool are mapped to either $\mathsf{True}_{\mathsf{bool}}$ or $\mathsf{False}_{\mathsf{bool}}$, rather than to semantic values $\top$ and $\bot$. Finally functions are mapped to a set of (argument, value) pairs, where the argument ranges over all (constants for) elements of the function's domain, and the corresponding values are given by the meaning of the function's tree.

Interpreting terms as terms in particular allows us to use Isabelle's pretty-printing facilities for terms to display the model. This may appear to be a small advantage at the moment, as we could easily have implemented pretty-printing for ground types, bool, and functions ourselves. It will turn out to be very useful however when we extend the translation to cover datatypes (see Section 3.6), elements of which can then be printed with any user-defined syntax that may exist for them in the Isabelle system: e.g. lists as "$[a, b, c]$" instead of "$\mathsf{Cons}\, a\, (\mathsf{Cons}\, b\, (\mathsf{Cons}\, c\, \mathsf{Nil}))$", or pairs as "$(a, b)$" instead of "$\mathsf{Pair}\, a\, b$".

2.5 Conclusion

We have presented a translation from higher-order logic to propositional formulae, such that the resulting propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size. A correctness proof for the translation was given in this chapter. A working

implementation, consisting of roughly 3,500 lines of code written in Standard ML [115], is available in the Isabelle/HOL theorem prover. A standard SAT solver can be used to search for a satisfying assignment for the propositional formula, and if such an assignment is found, it can be transformed into a model for the HOL formula. This allows for the automatic generation of finite countermodels for non-theorems in Isabelle/HOL. A similar translation has been discussed before [81]; our main contributions are its extension to higher-order logic, a proof of its correctness, and the seamless integration with a popular interactive theorem prover.

The applicability of the algorithm is limited by its non-elementary complexity. We believe that the algorithm can still be useful for practical purposes, since many formulae have small models (and small order). To substantiate this claim, some case studies are carried out in Chapter 4. First however, a number of extensions to the basic algorithm that was presented here are discussed in the following Chapter 3.

You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.
Winston Churchill, 1874–1965.

Chapter 3

Extensions and Optimizations

The actual Isabelle/HOL system offers various extensions on top of the basic HOL logic, mostly to improve usability. Among them are datatypes and recursive functions, axiomatic type classes, set types and extensible records. In this chapter we discuss how the translation to propositional logic can be augmented to cover these extensions, and also how it can be improved to generate smaller propositional formulae.

3.1 Introduction

The basic HOL logic described in Chapter 2, as implemented in the Isabelle/HOL system, has been augmented with several extensions. There are packages that provide the user with convenient means to define datatypes, recursive functions, axiomatic type classes, sets and records, and more. Figure 3.1 shows a dependency graph for some of these packages.

Their basic purpose, at a conceptual level, is similar: to let the user define objects in a concise, accessible and safe way, and handle the necessary translation from this description into the logic. The translation itself can be done in various ways. We distinguish the *axiomatic*, the *definitional* and the *internal* approach. The axiomatic approach simply asserts the properties that the user has stated about an object as new axioms of the theory under consideration. This is the easiest approach (especially from a package implementor's point of view), but also the most dangerous one. No logical means ensure that an object with the desired properties in fact exists, which makes introducing inconsistencies all too easy. Therefore the definitional approach is usually the approach of choice in the Isabelle system, even though it puts a much greater burden on the package. This approach requires new objects to be defined in terms of existing ones, and their properties derived (and proved), rather than just asserted. The

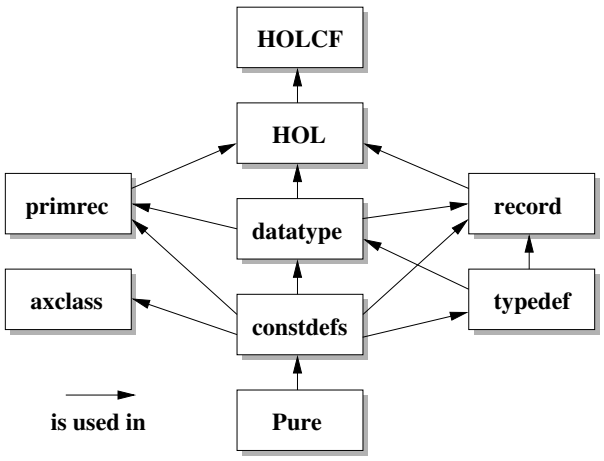


Figure 3.1: HOL package structure

current datatype package for example uses this approach [25]. For some concepts however, neither approach is feasible, because the underlying logic does not allow to define them. In this case the logic itself needs to be extended, and new objects are encoded internally, i.e. in the extended logic, rather than through added axioms or definitions. Axiomatic type classes [170] are an example of this.

The translation to propositional logic must be extended before it can properly deal with the various features present in the Isabelle/HOL implementation of higher-order logic. This also can be done in several ways, which roughly correspond to the axiomatic, definitional, and internal approaches described above. When a formula containing certain constants or types is translated, it is often sufficient to translate a set of *relevant* axioms, e.g. a constant's definition, along with the given formula to sufficiently restrict the possible models that the SAT solver may find. This straightforward solution may not always work however, perhaps because those relevant axioms mention infinite types (which would prevent us from finding any finite models at all), or perhaps because translating those axioms might pose a performance issue. Therefore some features require the translation to be extended in a more direct (and usually more involved) way, to interpret HOL concepts in propositional logic such that their relevant properties are implicitly preserved. Also a combination of both approaches can be used, where some properties are preserved by the translation itself, while others are guaranteed by adjoining relevant axioms. Finally extensions to the logic must be reflected in corresponding extensions to the translation. In the following sections of this chapter, we will consider individual features of Isabelle/HOL, and describe how the translation is extended to accommodate them. First however, we will discuss some modifications to the translation which can reduce the size of the generated propositional formula.

3.2 Optimizations

We describe some optimizations in the implementation of the translation $\phi_{\bar{T},\bar{M}}(\cdot)$. None of them affect soundness or completeness of the algorithm that was proved correct in Chapter 2.

Well-formedness. In Chapter 2, we had defined $\phi_{\bar{T},\bar{M}}(t_{\mathsf{bool}})$ as

$$\varphi_1 \wedge \neg\varphi_2 \wedge \bigwedge \mathrm{wf}(\bar{T}(\mathrm{FreeVars}(t_{\mathsf{bool}}))) \wedge \bigwedge \mathrm{wf}(\bar{M}(\mathrm{Names}(t_{\mathsf{bool}})))$$

(Def. 2.102), where $\mathcal{T}_{\bar{T},\bar{M}}(t_{\mathsf{bool}}) = \mathrm{Leaf}([\varphi_1, \varphi_2])$. We only consider well-formed truth assignments however, i.e. truth assignments which make exactly one formula φ_i in each (sub-)tree of the form $\mathrm{Leaf}([\varphi_1, \ldots, \varphi_n])$ true. One can easily show (and in fact, we have already done so: see Lemmas 2.81 and 2.96) that this property propagates from trees for variables and constants to trees for λ-abstractions and applications. Thus exactly one of the two formulae φ_1 and φ_2 is true (provided the truth assignment is well-formed), and it is sufficient to require

$$\varphi_1 \wedge \bigwedge \mathrm{wf}(\bar{T}(\mathrm{FreeVars}(t_{\mathsf{bool}}))) \wedge \bigwedge \mathrm{wf}(\bar{M}(\mathrm{Names}(t_{\mathsf{bool}}))).$$

The SAT solver never needs to consider $\neg\varphi_2$.

Undefined values. We can relax the notion of well-formedness to require truth assignments to make *at most* one formula φ_i in each (sub-)tree of the form $\mathrm{Leaf}([\varphi_1, \ldots, \varphi_n])$ true, rather than exactly one formula. This amounts to allowing undefined values and partial functions in our models, which will be put to good use when we consider recursive datatypes (see Section 3.6). Application of one tree to another, $\mathrm{apply}(t, u)$, yields a tree with a defined meaning if and only if u has a defined meaning, and t (which now denotes a partial function) is defined for the meaning of u. Thus undefinedness propagates from arguments to application terms.

The correctness proof given in the previous chapter can be modified to cover this different notion of well-formedness as well. The necessary modifications are significant however (because well-formedness is no longer equivalent to a defined meaning for trees), and we do not spell out the details. We merely note that the property of at most one label element being true, just like the property of exactly one being true, propagates from trees for variables and constants to trees for λ-abstractions and applications. Thus the optimization described in the previous paragraph on well-formedness remains valid.

An immediate consequence of this change in the notion of well-formedness is that our well-formedness formulae become simpler: let $l = [x_1, \ldots, x_n] \in \mathrm{List}_{\mathbb{P}}$. Instead of

$$wf(l) = \Big(\bigvee_{i=1}^{n} x_i\Big) \wedge \bigwedge_{\substack{i,j=1 \\ i \neq j}}^{n} (\neg x_i \vee \neg x_j)$$

(Def. 2.64), it is now sufficient to define

$$wf(l) := \bigwedge_{\substack{i,j=1 \\ i \neq j}}^{n} (\neg x_i \vee \neg x_j),$$

i.e. the first clause can be omitted. The effect of this change on overall performance is not totally clear however, due to the complexity of today's SAT solvers. While either definition ultimately requires the SAT solver to find a truth assignment that corresponds to a model where the HOL formula t_{bool} is true (thus in particular defined), the former definition may help to reduce the search space.

Application. The translation of application is based on the idea of an explicit case distinction over the argument's possible values: $t\,u$ (i.e. t applied to u) is equal to d if u is equal to u_1 and t maps u_1 to d, or if u is equal to u_2 and t maps u_2 to d, or .... Thus apply(t, u) yields a tree with labels which contain disjunctions of conjunctions.

Converting a disjunction of conjunctions to CNF (conjunctive normal form, i.e. a conjunction of disjunctions), the standard input format of most SAT solvers, is rather expensive and causes an increase in the size of the formula or (in case a definitional CNF transformation is used) in the number of Boolean variables. The problem is aggravated by nested applications: e.g. $t\,(u\,v)$ first requires $u\,v$ to be translated into a tree whose labels contain disjunctions of conjunctions, while applying t then yields a tree whose labels contain disjunctions of conjunctions of (nested) disjunctions of conjunctions. Thus nested applications lead to an unfavorable nesting of disjunctions and conjunctions at the propositional level. Furthermore, nested applications cause a duplication of sub-formulae: $t\,(u\,v)$ is equal to d_1 if $u\,v$ is equal to u_1 and t maps u_1 to d_1; $t\,(u\,v)$ is equal to d_2 if $u\,v$ is equal to u_1 and t maps u_1 to d_2; .... Thus the (possibly complex) formula which describes that $u\,v$ is equal to u_1 is duplicated once for every element in the codomain of t, and likewise for every formula which is used as a label element to describe that $u\,v$ is equal to u_i, where u_i is in the domain of t.

We can tackle these problems at the HOL level already (rather than at the propositional level) by introducing new variables as abbreviations for more complex subterms. For example, instead of translating $t\,(u\,v)$, we can consider $t\,s$ (where s is a fresh variable of the appropriate type), while adding $s = (u\,v)$ as an additional premise. This can greatly reduce the alternation depth of nested disjunctions and conjunctions, and instead of complex formulae, only single Boolean variables (which occur as labels in the tree for s) still need to be duplicated.

Propositional simplification. The translation of implication, equality, and bound variables introduces propositional constants True and False as label elements, which may then be combined with other label elements to produce more complex propositional formulae. The resulting formulae can immediately be simplified, using the following basic algebraic laws of $\neg$, $\vee$, $\wedge$, True, and False:

$$\begin{array}{lcl} \neg\text{True} & \equiv & \text{False} \\ \neg\text{False} & \equiv & \text{True} \end{array} \qquad \begin{array}{lcl} \text{True} \vee \varphi & \equiv & \text{True} \\ \varphi \vee \text{True} & \equiv & \text{True} \\ \text{False} \vee \varphi & \equiv & \varphi \\ \varphi \vee \text{False} & \equiv & \varphi \end{array} \qquad \begin{array}{lcl} \text{True} \wedge \varphi & \equiv & \varphi \\ \varphi \wedge \text{True} & \equiv & \varphi \\ \text{False} \wedge \varphi & \equiv & \text{False} \\ \varphi \wedge \text{False} & \equiv & \text{False} \end{array}$$

Doing this consequently results in closed HOL formulae without constants (other than implication and equality) being translated simply to Leaf([True, False]) or Leaf([False, True]). The SAT solver is used only to search for an interpretation of *free* variables.

Stripping outermost quantifiers. In contrast to what we just said, outermost universal quantifiers are stripped before a formula is translated when we are searching for a countermodel, e.g. $\forall x, y.\, P\,x\,y$ is instead translated as $P\,x\,y$. (Likewise, outermost existential quantifiers can be stripped when we are searching for a model.) The advantage of this is two-fold. First, we avoid translating the body several times: universal/existential quantification is in essence translated as a finite conjunction/disjunction over all possible values, which can lead to a combinatorial explosion in the presence of nested quantifiers. Second, we use the SAT solver

to search for an interpretation of the now free variables; if a model is found, it contains actual instantiations for these variables, which can be displayed to the user. (Models of course don't contain instantiations for bound variables.)

Small types. Variables of a type with size 1 can be represented by Leaf([True]), using no Boolean variable at all (instead of one Boolean variable x together with a well-formedness formula x). While this has little effect at the SAT solver level due to unit propagation, it allows a more extensive simplification (cf. the above paragraph on propositional simplification) of the resulting Boolean formulae. Also variables of a type with size 2, including variables of type bool, can be represented by a tree of the form Leaf($[x, \neg x]$), rather than by a tree Leaf($[x_0, x_1]$) and a corresponding well-formedness formula $(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$.

Unfolding and specialization. More importantly, we avoid unfolding the definition of logical constants (i.e. $\mathsf{True}_{\mathsf{bool}}$, $\mathsf{False}_{\mathsf{bool}}$, $\neg_{\mathsf{bool}\to\mathsf{bool}}$, $\wedge_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$, $\vee_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$, and the quantifiers $\forall_{(\sigma\to\mathsf{bool})\to\mathsf{bool}}$, $\exists_{(\sigma\to\mathsf{bool})\to\mathsf{bool}}$) as λ-terms as far as possible. Instead these constants are replaced directly by their counterparts in propositional logic. Since every type is finite, quantifiers of arbitrary order can be replaced by a finite conjunction or disjunction.

The latter leads to a more general optimization technique, applicable also to other functions and predicates (including e.g. equality): namely specialization of the rule for function application to particular functions. While any given function can be represented by a tree, it is often more efficient to implement a particular function's action on its arguments, assuming these arguments are given as trees already, than to translate the function into a tree to which the general translation rule for application needs to be applied. For $=_{\sigma\to\sigma\to\mathsf{bool}}$ this avoids creating a tree whose size is proportional to $|[\![\sigma]\!]_{E,M}|^2$, and instead uses a function that operates on two trees representing elements of $[\![\sigma]\!]_{E,M}$ to produce a tree for a Boolean value.

Three-valued logic. We apply the same specialization technique to the logical constants to achieve a translation that corresponds to a three-valued logic, where a logical constant applied to possibly undefined arguments yields a tree with an undefined meaning if and only if the meaning of the entire expression depends on the truth value of an undefined argument. More precisely, let $*$ denote arguments of type bool whose tree has an undefined meaning. Then the special rules implemented are

$$\begin{array}{lcllcl} \mathsf{True} \vee * & \equiv & \mathsf{True} & \mathsf{False} \wedge * & \equiv & \mathsf{False} \\ * \vee \mathsf{True} & \equiv & \mathsf{True} & * \wedge \mathsf{False} & \equiv & \mathsf{False}, \end{array}$$

and quantifiers are again treated as finite conjunctions or disjunctions. In all other cases, our usual definition of the translation of application will cause undefined argument values to propagate. In effect, this implements Kleene's three-valued logic [57]. The SAT solver is thus relieved from assigning a defined meaning to irrelevant parts of a formula.

Equality can be extended to this three-valued logic as well: trees are considered equal if they both denote the same total function, and not equal if they denote (possibly partial) functions that disagree for at least one argument. It is undefined (i.e. unknown, neither true nor false) however whether a tree whose meaning is a partial function is equal to another tree whose meaning is an extension of this partial function. This definition is applied recursively to curried functions, which yield values of function type.

3.3 Isabelle's Meta-Logic

Aside from HOL, a variety of other logics can be (and have been) defined in Isabelle, e.g. first-order logic [130], modal and linear logics [18, 87], and Zermelo-Fraenkel set theory [131, 132, 133]. These logics are formulated within Isabelle's meta logic [129], *Isabelle/Pure*. Isabelle/Pure offers a 2-element type Prop of propositions, and three logical constants: (meta) implication, (meta) equality, and (meta) universal quantification.

These constants are translated just like their Isabelle/HOL counterparts. No distinction is made between types Prop and bool, and the constant Trueprop, which converts a Boolean value into a proposition, is treated as the identity function. Despite the different names, Isabelle/Pure really is just an implementation of what we defined as higher-order logic in Chapter 2, while Isabelle/HOL extends this logic substantially, as mentioned before and described in detail in the following sections of this chapter.

The fact that the translation can handle Isabelle's meta logic allows it to be applied to other logics defined on top of Isabelle/Pure, aside from Isabelle/HOL. The (counter-)model finder could easily be turned into a *generic* tool that is not restricted to a single object logic.

3.4 Type and Constant Definitions, Overloading

When we search for a model of a HOL formula t_{bool}, it is clear that this model should not only satisfy t_{bool}, but also every axiom of the theory under consideration. The axioms of the basic HOL theory are already respected by the hard-wired translation of the logical constants presented earlier, but these axioms can be augmented with arbitrary user-supplied axioms. Therefore not only t_{bool}, but also all axioms of the current theory would have to be translated to propositional logic and passed to the SAT solver.

As there can be hundreds or even thousands of axioms in a theory, this is usually infeasible. Luckily, and because it is all too easy to introduce inconsistencies with the *axiomatic approach*, the Isabelle system provides more controlled means of asserting axioms to define new types and constants. Users are encouraged to develop their theories via the *definitional approach* (described below), which has been shown to be consistency-preserving and *meta-safe* [170], in the sense that additional axioms merely define new names as abbreviations for pre-existing syntactic objects. For such theories it is sufficient to consider a (usually small) set of axioms that are *relevant* wrt. the given term t_{bool}, while all irrelevant axioms can safely be ignored.

3.4.1 Type Definitions

A type definition introduces an axiom stating that $(\alpha_1, \ldots, \alpha_n)T$ is isomorphic to A, where T is a type constructor with arity n, and A is a term representing some set. There are several side conditions: T must be new and not occur in A, A must be closed, $\mathrm{TyVars}(A) \subseteq \{\alpha_1, \ldots, \alpha_n\}$, and non-emptiness of the set A must be derivable. The type definition then introduces three new constants Rep_T, Abs_T and T (where the constant T is just defined to abbreviate the term A), and the isomorphism axiom $\mathrm{type\_definition}_T$ is stated as follows:

$$(\forall x.\, \mathsf{Rep}_T\, x \in T) \wedge (\forall x.\, \mathsf{Abs}_T\, (\mathsf{Rep}_T\, x) = x) \wedge (\forall y.\, y \in T \implies \mathsf{Rep}_T\, (\mathsf{Abs}_T\, y) = y).$$

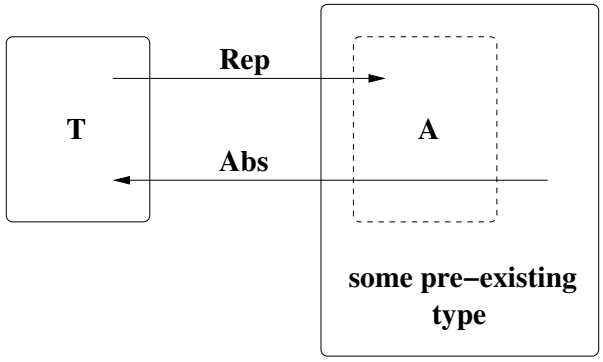


Figure 3.2: HOL type definition

In other words, the range of Rep_T is contained in A, Abs_T is a left inverse of Rep_T, and Rep_T is a left inverse of Abs_T when the latter is restricted to the set A. Figure 3.2 provides a graphical illustration.

The isomorphism axiom type_definition$_T$ is considered relevant for a given term t_σ if and only if the type $(\sigma_1, \ldots, \sigma_n)T$ occurs in (a subterm of) t_σ. (Note that this is automatically the case if Rep_T or Abs_T occur as constants, i.e. if $\{\mathsf{Rep}_T, \mathsf{Abs}_T\} \cap \mathrm{Names}(t_\sigma) \neq \emptyset$.) In this case, the axiom—with all type variables α_1, ..., α_n replaced by the actual type parameters σ_1, ..., σ_n—is conjoined to the HOL formula under consideration, and translated to propositional logic as well.

Ignoring the type_definition$_T$ axiom otherwise is justified precisely because type definitions are safe. Note that this in turn requires the interpretation of types from a subset-closed universe $\mathcal{U}$ (property **Sub** in Section 2.2). A counterexample which shows that type definitions become unsafe if we drop the **Sub** requirement on $\mathcal{U}$ is sketched in [170].

3.4.2 Constant Definitions and Overloading

An overloaded constant definition introduces a finite set of equations of the form $c_{\tau^i} = t^i$, provided that c is a new constant of type τ, τ^i is an instance of τ (for every i), every t^i is closed, and $\mathrm{TyVars}(t^i) = \mathrm{TyVars}(\tau^i)$. Furthermore no two different c_{τ^1}, c_{τ^2} may have a common instance, and recursive occurrences of $c_{\tau'}$ in some t^i require that τ' is strictly simpler than τ^i in a well-founded sense. (Structural containment of τ' in τ^i is certainly sufficient, but not necessary. See [127] for a detailed discussion.)

An equation $c_{\tau^i} = t^i$ is considered relevant for a given term t_σ if and only if $c_{\tau'}$ occurs in t_σ, and τ' is an instance of τ^i. (No other equation $c_{\tau^j} = t^j$ with $i \neq j$ can be relevant in this case, since τ^i and τ^j may not have a common instance.) In this case, the relevant equation—again with all type variables replaced by their image under the type substitution which shows that τ' is an instance of τ^i—could be conjoined to the HOL formula under consideration, similar to how we treated type definitions above.

However, conjoining a definition and translating it to propositional logic can be rather inefficient, especially in the case of function definitions with multiple (curried) arguments, where the translation needs to build a function tree by iterating over all possible values for the ar-

guments. This quickly leads to a combinatorial explosion. Our default strategy thefore is to unfold the relevant definition, i.e. to replace the constant $c_{\tau'}$ in the input formula by the right-hand side t^i of the defining equation (with type variables in t^i replaced as described above). Then β-reduction is performed if possible, i.e. if the right-hand side is a λ-abstraction that is applied to one or more arguments in t_σ. Theoretically this can cause a non-elementary blowup in the length of the input term, but since most definitions are well-behaved in practice, the unfolding approach has so far proved to be superior to conjoining relevant definitions.

The Isabelle/HOL implementation for convenience allows function definitions to have the form $c_{\tau^i}\, x^1 \,\ldots\, x^n = t^i$, where each x^k is a variable, and $\mathrm{FreeVars}(t^i) \subseteq \{x^1, \ldots, x^n\}$. Since there may be less than n actual parameters for $c_{\tau'}$ in the input formula, we first normalize such an equation to the equivalent form $c_{\tau^i} = \lambda x^1, \ldots, x^n.\, t^i$, before substituting the new right-hand side for $c_{\tau'}$ and possibly performing β-reduction as described above.

Constant definitions that adhere to the format and restrictions described here are safe [170]. Thus irrelevant constant definitions (just like irrelevant type definitions) can be ignored when the axioms of a theory are translated to propositional logic. In practice, this eliminates the largest deal of all axioms: Isabelle/HOL, at the time of writing, contains 3721 axioms, out of which 3672 are constant definitions. The remaining axioms are mostly type definitions, class axioms (see Section 3.5), or defining the relation between basic logical constants in Isabelle/HOL and Isabelle/Pure.

3.4.3 Definite Description and Hilbert's Choice

Hilbert's choice operator, ϵ, is a polymorphic constant of type $(\sigma \to \mathsf{bool}) \to \sigma$, satisfying the axiom

$$\text{someI}: \quad (\exists x.\, P\, x) \implies P\,(\epsilon\, P).$$

Similarly, The, also a constant of type $(\sigma \to \mathsf{bool}) \to \sigma$, satisfies

$$\text{the\_eq\_trivial}: \quad (\mathsf{The}\, x.\, x = a) = a,$$

and $\mathsf{arbitrary}$ is a completely unspecified polymorphic constant. (Of course one can nevertheless prove certain theorems that mention $\mathsf{arbitrary}$, e.g. $\mathsf{arbitrary}_\alpha = \mathsf{arbitrary}_\alpha$ by reflexivity.) For the purpose of our translation $\mathcal{T}$, we can treat these logical constants just like any other constant, and introduce trees labeled with Boolean variables that determine their interpretation. For ϵ and The, we then translate the conjunction of the input formula t_{bool} with the relevant axiom (i.e. someI or the_eq_trivial, or both axioms if both ϵ and The occur in t_{bool}). As usual, the type variable in someI (or in the_eq_trivial) is instantiated to match the type of ϵ (or the type of The, respectively) in t_{bool}.

Note that we have to add multiple copies of the relevant axiom(s), instantiated to different types, when there are multiple occurrences of ϵ (or The) in t_{bool} which differ in type. This is similar for usual constant definitions, and also for type definitions when a type constructor (with arity at least 1) is applied to different argument types.

3.5 Axiomatic Type Classes

Axiomatic type classes extend the first-order type system of HOL introduced in Section 2.2 with ordered type classes that qualify types. An axiomatic type class is the class of all types that satisfy certain properties, the *class axioms*. As an example, consider the HOL formula $\forall x_\alpha, y_\alpha.\, x_\alpha = y_\alpha$, which has one free type variable α. As a class axiom, it describes the class of singleton types, i.e. types containing only one element. Type classes were introduced for Isabelle in [122], and a more recent description is found in [170].

Type classes are encoded in HOL by adding a new type constructor itself with arity 1 to the type structure, and a new polymorphic constant TYPE of type $\alpha\,\mathsf{itself}$ to the signature. Furthermore, a polymorphic constant C of type $\alpha\,\mathsf{itself} \to \mathsf{bool}$ is introduced for every type class C. Now the term $C_{\sigma\,\mathsf{itself}\to\mathsf{bool}}\,\mathsf{TYPE}_{\sigma\,\mathsf{itself}}$ is intended to encode that type σ belongs to type class C. To achieve this, the meaning of itself is chosen to be the function that maps $A \in \mathcal{U}$ to the singleton set $\{A\}$ (which is assumed to be in $\mathcal{U}$), and consequently the meaning of $\mathsf{TYPE}_{\sigma\,\mathsf{itself}}$ must be the meaning of σ.

A type class definition for a class C with class axioms $\phi_1, \ldots, \phi_n$, aside from introducing the constant C mentioned above, also asserts an axiom

$$\text{C\_class\_def}: \quad C\,\mathsf{TYPE}_{\alpha\,\mathsf{itself}} = \phi_1 \wedge \cdots \wedge \phi_n,$$

provided that $\text{FreeVars}(\phi_i) = \emptyset$ (which we can always achieve by taking the universal closure over all free variables otherwise) and $\text{TyVars}(\phi_i) \subseteq \{\alpha\}$ for $1 \le i \le n$.

Isabelle/HOL encourages the definition of type classes which have one or more *superclasses* $C_1, \ldots, C_k$. Superclasses allow to establish an inclusion relation on type classes, which is a necessary prerequisite for an order-sorted type system [122]. Their logical significance is that every type in C satisfies not only the class axioms of C, but also those of C's superclasses $C_1, \ldots, C_k$ (and in turn the class axioms of their superclasses, if they have any). The axiom C_class_def in this case has the following form:

$$C\,\mathsf{TYPE}_{\alpha\,\mathsf{itself}} = C_1\,\mathsf{TYPE}_{\alpha\,\mathsf{itself}} \wedge \cdots \wedge C_k\,\mathsf{TYPE}_{\alpha\,\mathsf{itself}} \wedge \phi_1 \wedge \cdots \wedge \phi_n.$$

Class axioms become relevant for a term t_σ in two (not necessarily related) cases: when t_σ contains the constant C, and also when t_σ contains a type variable which is explicitly restricted to the class C. (Isabelle annotates type variables with *sorts*, which are finite sets of type classes. A sort is understood as an intersection, i.e. a type variable that belongs to a sort S belongs to each type class $C \in S$. We write $\alpha :: S$ for a type variable α that is annotated with sort S, and we usually omit the empty sort: α is short for $\alpha :: \emptyset$.)

In the first case, the entire axiom C_class_def is relevant, and conjoined with the input term as usual, after the type of C in the class axiom has been instantiated to match the type of the actual occurrence of C in t_σ. More needs to be done however. Isabelle's inference system has been enhanced to support sort annotations on type variables, and contains the axiom scheme

$$\text{class\_triv}: \quad C\,\mathsf{TYPE}_{\alpha::\{C\}\,\mathsf{itself}}$$

(where C is an arbitrary type class) as a basic rule. It is this axiom scheme that defines the logical meaning of sort annotations. The axiom scheme, with C instantiated to the actual class

constant in t_σ, is therefore conjoined as another relevant term. Note that type instantiation will fail if the domain of class constant C in t_σ is not restricted to be in the type class C, i.e. if the actual domain type is not annotated with a sort S that contains class C. No instance of the class_triv axiom scheme is considered relevant then. Of course the C_class_def axiom must still be satisfied though.

In the second case, t_σ merely contains a type variable $\alpha :: S$ with $C \in S$. We could consider the same relevant axioms as in the first case (i.e. C_class_def and class_triv), but this would unnecessarily introduce the constants C and TYPE in relevant axioms, which can be avoided. Instead, only the class axioms $\phi_1, \ldots, \phi_n$ (with their single type variable replaced by $\alpha :: S$) are considered relevant. In addition, we now need to keep track of superclass relations ourselves: also all class axioms of (direct or indirect) superclasses of C are considered relevant.

3.6 Datatypes and Recursive Functions

Isabelle/HOL has packages that ease the definition of inductive datatypes (e.g. lists, trees) and recursive functions over datatypes. The current version of the datatype package [22, 25] supports a number of advanced features, including *mutual* and *indirect recursion*, and *arbitrary branching* over existing types. A general datatype specification looks as follows:

$$\begin{aligned}&\textbf{datatype}\ (\alpha_1, \ldots, \alpha_h)t_1 = C_1^1\ \sigma_{1,1}^1\ \ldots\ \sigma_{1,m_1^1}^1 \mid \ldots \mid C_{k_1}^1\ \sigma_{k_1,1}^1\ \ldots\ \sigma_{k_1,m_{k_1}^1}^1\\ &\quad\textbf{and}\ \ldots\\ &\quad\textbf{and}\ (\alpha_1, \ldots, \alpha_h)t_n = C_1^n\ \sigma_{1,1}^n\ \ldots\ \sigma_{1,m_1^n}^n \mid \ldots \mid C_{k_n}^n\ \sigma_{k_n,1}^n\ \ldots\ \sigma_{k_n,m_{k_n}^n}^n\end{aligned}$$

Here $\alpha_1, \ldots, \alpha_h$ are type variables, constructors C_i^j are distinct, they are annotated with the types $\sigma_{i,1}^j, \ldots, \sigma_{i,m_i^j}^j$ of their arguments (where $m_i^j \geq 0$), and each argument type $\sigma_{i,i'}^j$ must be an *admissible* type containing at most the type variables $\alpha_1, \ldots, \alpha_h$. Admissibility is defined in [25]; it is required to restrict recursive occurrences of types in a way that ensures the existence of a set-theoretic model for the datatype. For example,

$$\textbf{datatype}\ t = C\,(t \to \mathsf{bool})$$

would not be a valid datatype specification, because a model for this datatype would require an injection $C\colon (\llbracket t \rrbracket_{E,M} \to \mathbb{B}) \to \llbracket t \rrbracket_{E,M}$, violating Cantor's theorem [32]. Admissibility currently also rules out *heterogeneous* datatypes like the type of *powerlists* [1], which could otherwise be defined as

$$\textbf{datatype}\ (\alpha)\mathsf{PList} = \mathsf{Zero}\,\alpha \mid \mathsf{Succ}\,(\alpha \times \alpha)\mathsf{PList}.$$

Furthermore, each datatype must be non-empty, since HOL does not admit empty types. This is guaranteed iff each datatype $(\alpha_1, \ldots, \alpha_h)t_j$ (for $1 \leq j \leq n$) has a constructor C_i^j such that each argument type $\sigma_{i,i'}^j$ (for $1 \leq i' \leq m_i^j$) which is an instance of a simultaneously defined datatype $(\alpha_1, \ldots, \alpha_h)t_{j'}$ (for some $1 \leq j' \leq n$) is non-empty.

Internally, the datatype package follows the definitional approach. Instead of just asserting the properties of a datatype, the new datatype and its constructors are defined in terms of

existing concepts (using HOL's type and constant definitions, respectively). The actual definition (which is hidden from the user, who can always work with the more convenient notions provided by the package) is fairly elaborate. Starting from a type $(\alpha, \beta)\mathsf{dtree}$ of trees, the representing set for a datatype is cut out inductively as the least set (using the Knaster-Tarski theorem [157], which justifies inductive definitions) that contains representing trees for all of the datatype's elements. More specifically, the type $(\alpha, \beta)\mathsf{dtree}$ is defined as $(\alpha, \beta)\mathsf{node\,set}$, where $(\alpha, \beta)\mathsf{node}$ is defined as $(\mathsf{nat} \to (\beta + \mathsf{nat})) \times (\alpha + \mathsf{bool})$. For this type, certain injective operations can be defined (namely Leaf of type $\alpha \to (\alpha, \beta)\mathsf{dtree}$, $\mathsf{In0}$, $\mathsf{In1}$, both of type $(\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree}$, Pair of type $(\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree}$, and Lim of type $(\beta \to (\alpha, \beta)\mathsf{dtree}) \to (\alpha, \beta)\mathsf{dtree}$), which allow to embed non-recursive occurrences of types in a datatype specification (Leaf), to model distinct constructors ($\mathsf{In0}$, $\mathsf{In1}$), to model constructors with multiple arguments (Pair), and to embed functions types (Lim). More details can be found in [25].

Using a datatype's internal definition is not an option when we translate a HOL formula to propositional logic. The type $(\alpha, \beta)\mathsf{dtree}$, regardless of α and β, is infinite. This would prohibit finding finite models even for datatypes that only have a finite number of elements. We could alleviate the problem by considering some finite subset of dtrees only. However, the correspondence between dtrees and a datatype's elements is not very intuitive, and disallowing certain dtrees may lead to unexpected models at the datatype level. Furthermore, expanding the internal definitions—in particular the definition of a datatype's representing set as a least fixed point—would be rather inefficient at the propositional level. Therefore we have instead extended the translation to be able to deal with datatypes directly. Inductive types are fully determined by freeness of their constructors (which guarantees that the datatype is "big enough") and structural induction (which guarantees that there are not "too many" elements). These are the two important properties that must be preserved by the translation.

3.6.1 Non-Recursive Datatypes

We distinguish between recursive and non-recursive datatypes. The reason is that we search for finite models, and thus we can accommodate only finite datatypes offhand. A non-empty datatype $(\alpha_1, \ldots, \alpha_h)t_j$ is finite iff it has no recursive or mutually recursive constructor, and every argument type of each of its constructors is finite. The latter is ensured by the fact that we have fixed a finite type environment and model before translation (cf. Lemma 2.40). Therefore the finite datatypes coincide with the non-recursive ones. (As a simple example, consider the datatype $(\alpha)\mathsf{option} = \mathsf{None} \mid \mathsf{Some}\,\alpha$. This type is finite since $[\![\alpha]\!]_{E,M}$ is finite in our setting; its size is $1 + |[\![\alpha]\!]_{E,M}|$.) Freeness and structural induction imply that in this case the datatype's size, i.e. the number of elements of the datatype, is given by

$$\sum_{i=1}^{k_j} \prod_{i'=1}^{m_i^j} |[\![\sigma_{i,i'}^j]\!]_{E,M}|,$$

i.e. by the sum over the datatype's constructors of the product over the constructor's arguments of their respective type's size.

This immediately leads us to the representation of a datatype's elements as trees of propositional formulae. With regard to this representation, a datatype is treated just like a ground

type of the same size. A datatype's element is given by a leaf, its length equal to the datatype's size. The tree-like structure that can be imposed on constructor terms denoting elements of a datatype is *not* reflected in the tree structure of the corresponding propositional tree, which remains to be used for the encoding of function types only. It is clear then that each element of a datatype (just like each element of a ground type) has exactly one representing constant propositional tree.

Datatype constructors are functions, mapping their arguments to an element of the datatype. A nullary constructor is a certain element of the datatype, and hence translated as a constant leaf. Constructors which take n arguments are translated as constant propositional trees of height $n+1$, representing an n-ary function. To translate a datatype constructor, it is necessary to have a fixed order for the elements of a datatype (and vice versa: a precise definition of the translation for datatype constructors implies a fixed order for the datatype elements). For non-recursive datatypes, this order—and hence the corresponding translation of constructors—is relatively simple. The (user-supplied) datatype specification already imposes an order on the datatype's constructors, namely the order in which they are given (i.e. $C^j_{i_1} < C^j_{i_2}$ iff $i_1 < i_2$). Assuming orders for all their argument types, we can "lift" this order to the datatype elements.

Definition 3.1 (Order on Non-Recursive Datatypes)**.** Let $(\alpha_1, \ldots, \alpha_h)t_j$ be a non-recursive datatype, given by the general datatype specification stated at the beginning of Section 3.6. The *(datatype) order* $<_{t_j}$ *on* $[\![(\alpha_1, \ldots, \alpha_h)t_j]\!]_{E,M}$ is given by

$$[\![C^j_{i_1}]\!]_{\mathcal{A},\mathcal{M}}\, x_1 \ldots x_{m^j_{i_1}} <_{t_j} [\![C^j_{i_2}]\!]_{\mathcal{A},\mathcal{M}}\, y_1 \ldots y_{m^j_{i_2}} \quad \text{iff}$$
$$i_1 < i_2 \vee \left[i_1 = i_2 \wedge (x_1, \ldots, x_{m^j_{i_1}}) < (y_1, \ldots, y_{m^j_{i_2}}) \right],$$

where the argument tuples are compared wrt. the lexicographic order on $[\![\sigma^j_{i,1}]\!]_{E,M} \times \cdots \times [\![\sigma^j_{i,m^j_{i_1}}]\!]_{E,M}$ that is induced by the individual (presupposed) orders on $[\![\sigma^j_{i,1}]\!]_{E,M}, \ldots, [\![\sigma^j_{i,m^j_{i_1}}]\!]_{E,M}$.

We merely write $<$ instead of $<_{t_j}$ when the datatype is clear from context.—As an example, consider the type $(\alpha)\mathsf{option}$ again, with its above specification. Suppose that $[\![\alpha]\!]_{E,M} = [a_1, \ldots, a_n]$, i.e. $a_1 < \cdots < a_n$. Because None comes before Some in the datatype specification, the order on $(\alpha)\mathsf{option}$ (which is a type of size $n+1$ in this case) is then given by

$$[\![\mathsf{None}]\!]_{\mathcal{A},\mathcal{M}} < [\![\mathsf{Some}]\!]_{\mathcal{A},\mathcal{M}}\, a_1 < \cdots < [\![\mathsf{Some}]\!]_{\mathcal{A},\mathcal{M}}\, a_n.$$

The translation of None is the tree $\mathrm{Leaf}(\mathrm{uv}^{n+1}_1)$ (recall Def. 2.76), and Some, which is a function of type $\alpha \to (\alpha)\mathsf{option}$, is translated as the following tree of height 2 and width n: $\mathrm{Node}([\mathrm{Leaf}(\mathrm{uv}^{n+1}_2), \ldots, \mathrm{Leaf}(\mathrm{uv}^{n+1}_{n+1})])$.

More generally, the order of elements for a non-recursive datatype with constructors C_1, …, C_k, where C_1 takes arguments of type σ_1, …, σ_m (and $[\![\sigma_i]\!]_{E,M} = [x^i_1, \ldots, x^i_{n_i}]$, for $1 \le i \le m$) is shown in Figure 3.3. Def. 3.1 is generalized to instances $(\tau_1, \ldots, \tau_h)t_j$ of a non-recursive datatype $(\alpha_1, \ldots, \alpha_h)t_j$ in the obvious way.

3.6.2 Recursive Datatypes

Recursive datatypes like nat, the type of natural numbers (with its constructors 0_{nat} and $\mathsf{Suc}_{\mathsf{nat}\to\mathsf{nat}}$), or $(\alpha)\mathsf{list}$, the type of lists with elements of type α (with constructors $\mathsf{Nil}_{(\alpha)\mathsf{list}}$ and

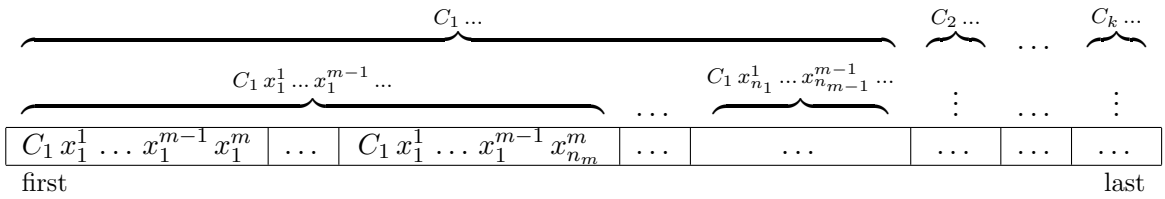


Figure 3.3: Element order for non-recursive datatypes

$\mathsf{Cons}_{\alpha\to(\alpha)\mathsf{list}\to(\alpha)\mathsf{list}}$), as well as more complex examples involving mutual or nested recursion, require an infinite model. Hence they cannot be treated in full generality in a finite model generation framework.

To be able to treat them at all, we consider *finite approximations* of such datatypes. When translating a HOL formula t_{bool} that involves recursive datatypes, we extend the type model (which gives us the semantics of type constructors in t_{bool} as finite sets) to provide finite sets for all recursive datatypes in t_{bool} as well. Currently these sets correspond to *initial fragments* of a datatype, i.e. to all elements of the datatype whose canonical representation contains at most a certain number of constructor applications. (The elements of type nat that can be written with at most 3 constructor applications, for example, are 0_{nat}, $\mathsf{Suc}\,0$, and $\mathsf{Suc}\,(\mathsf{Suc}\,0)$.) When the search for a model fails for a given initial fragment, the size of the fragment is increased, similar to how the size of sets corresponding to other types is increased (cf. Section 2.4.2). The only difference is that we increment the allowed number of constructor applications, rather than the cardinality of the finite approximation. The following definition already covers the general case of (instances of) mutually recursive datatypes.

Definition 3.2 (Initial Datatype Fragment)**.** Consider the general datatype specification stated at the beginning of Section 3.6. Let $r = (r_1, \ldots, r_n) \in \mathbb{N}^n$. The *$r$-th initial (datatype) fragment of a type σ*, written σ^r, is defined as follows:

1. If $\sigma = (\tau_1, \ldots, \tau_h)t_j$ is an instance of $(\alpha_1, \ldots, \alpha_h)t_j$ for some $1 \le j \le n$, i.e. $(\tau_1, \ldots, \tau_h)t_j = (\alpha_1, \ldots, \alpha_h)t_j\,\Theta$ for some type substitution Θ, then

$$\sigma^r := \begin{cases} \emptyset & \text{if } r_j = 0; \\ \left\{ [\![C_i^j]\!]_{\mathcal{A},\mathcal{M}}\, x_1\, \ldots\, x_{m_i^j} \;\middle|\; 1 \le i \le k_j \wedge \right. & \\ \qquad \left. x_{i'} \in (\sigma_{i,i'}^j\,\Theta)^{(r_1,\ldots,r_j-1,\ldots,r_n)} \text{ for } 1 \le i' \le m_i^j \right\} & \text{otherwise.} \end{cases}$$

2. $\sigma^r := [\![\sigma]\!]_{E,M}$ otherwise.

Lemma 3.3. *Let $r = (r_1, \ldots, r_n) \in \mathbb{N}^n$. Then σ^r is a finite subset of $[\![\sigma]\!]_{E,M}$.*

Proof. By induction on $\sum_{j=1}^n r_j$. If σ is not an instance of a datatype, then $\sigma^r = [\![\sigma]\!]_{E,M}$. Finiteness in this case follows from Lemma 2.40.

If $\sigma = (\alpha_1, \ldots, \alpha_h)t_j\,\Theta$, then $\sigma^r = \emptyset$ if $r_j = 0$. In this case the claim is trivial. Otherwise, i.e. if $r_j > 0$, the claim follows from the induction hypothesis, applied to $(\sigma_{i,i'}^j\,\Theta)^{(r_1,\ldots,r_j-1,\ldots,r_n)}$ (for each $1 \le i \le k_j$, $1 \le i' \le m_i^j$). □

Note that the proof assumes that *all* infinite datatypes were declared within the same general datatype specification. The actual Isabelle/HOL system does not impose this restriction. Therefore if a datatype specification mentions instances of previously declared infinite datatypes, the vector r has to be augmented accordingly, to provide bounds for all relevant infinite datatypes.

Lemma 3.4. *Let $r = (r_1, \ldots, r_n) \in \mathbb{N}^n$. Let $1 \leq j' \leq n$. Then $\sigma^r \subseteq \sigma^{(r_1,\ldots,r_{j'}+1,\ldots,r_n)}$.*

Proof. By induction on $\sum_{j=1}^{n} r_j$. If σ is not an instance of a datatype, then $\sigma^r = [\![\sigma]\!]_{E,M} = \sigma^{(r_1,\ldots,r_{j'}+1,\ldots,r_n)}$.

If $\sigma = (\alpha_1, \ldots, \alpha_h)t_j\,\Theta$, then $\sigma^r = \emptyset$ if $r_j = 0$. In this case the claim is trivial. Otherwise, i.e. if $r_j > 0$, the claim follows from the induction hypothesis, applied to $(\sigma^j_{i,i'}\,\Theta)^{(r_1,\ldots,r_j-1,\ldots,r_n)}$ (for each $1 \leq i \leq k_j$, $1 \leq i' \leq m^j_i$). □

Lemma 3.5. *Let $r \in (\mathbb{N} \setminus \{0\})^n$. Then σ^r is non-empty.*

Proof. If σ is not an instance of a datatype, then clearly $[\![\sigma]\!]_{E,M}$ is non-empty because HOL does not permit empty types (see Remark 2.11 and property **Inhab**, Section 2.2).

If $\sigma = (\alpha_1, \ldots, \alpha_h)t_j\,\Theta$ is a datatype instance, non-emptiness of σ^r follows from the restriction imposed on datatype specifications (to enforce non-emptiness of datatypes), namely that $(\alpha_1, \ldots, \alpha_h)t_j$ must have at least one constructor C^j_i such that each argument type $\sigma^j_{i,i'}$ which is an instance of a simultaneously defined datatype $(\alpha_1, \ldots, \alpha_h)t_{j'}$ is non-empty. □

We can in fact easily compute the exact size of an initial datatype fragment, by generalizing the formula (given earlier in this section) for the size of non-recursive datatypes.

Lemma 3.6. *Consider the general datatype specification stated at the beginning of Section 3.6. Let $1 \leq j \leq n$. Let $r = (r_1, \ldots, r_n) \in \mathbb{N}^n$ such that $r_j > 0$. Let Θ be a type substitution. Then*

$$|((\alpha_1, \ldots, \alpha_h)t_j\,\Theta)^r| = \sum_{i=1}^{k_j} \prod_{i'=1}^{m^j_i} |(\sigma^j_{i,i'}\,\Theta)^{(r_1,\ldots,r_j-1,\ldots,r_n)}|.$$

Proof. By induction on $\sum_{j=1}^{n} r_j$, using Def. 3.2. Freeness of constructors justifies the above sum, while the fact that each constructor is an injective function explains the nested product. □

We have thus eliminated the need for infinite models for recursive datatypes, and instead replaced them by finite approximations, based on a limit on the number of nested constructor occurrences.

Treating infinite datatypes in this way comes at a price: the resulting algorithm is not sound anymore; spurious (counter-)models may be returned. For example, consider the formula $\forall n_{\mathsf{nat}}.\, n = 0_{\mathsf{nat}}$. This formula is clearly false, but it becomes true when we only consider those elements of type nat that can be written with at most 1 constructor application (the only such element in fact being 0_{nat} itself). For this simple example, the spurious model could be ruled out by considering nat^2 instead of nat^1, but that is not always the case: e.g. the formula

$\forall n_{\mathsf{nat}}.\, n \leq m$ is satisfiable for any nat^r (where $r \geq 1$: take m to be $r-1$), but false for the infinite type nat.

What *can* we say about the algorithm then, if it may not find (infinite) models for satisfiable formulae, and if it may return spurious models for unsatisfiable formulae? At least that it remains sound if recursive datatypes occur only *positively*. Formulae like $0_{\mathsf{nat}} < n_{\mathsf{nat}}$ or $f\, x_\alpha = 0_{\mathsf{nat}}$ are satisfiable in nat^2 and nat^1, respectively. Therefore they are also satisfiable in nat simply because of the syntactic restriction on the datatype's occurrences. After all, nat^r is a subset of nat (for any r). Therefore any element of nat^r also is an element of nat, and any function to nat^r is (or rather, can be seen as) a function to nat as well.

The fact that we simply "cut off" a datatype at a certain number of nested constructor applications also raises another question: how should we translate terms that cannot be interpreted in the initial fragment of the datatype that we consider? What does $\mathsf{Suc}\,(\mathsf{Suc}\,0)$ mean, for example, when we are working with nat^2 (or nat^1) instead of nat? We propose to treat such terms as undefined. As explained earlier in Section 3.2, this undefinedness propagates when functions or predicates are applied to these terms. Recursive datatype constructors become *partial* functions then. Before we can focus on their translation in detail, we need to fix the order of datatype elements. Of course we have quite a bit of freedom here; any order will do as long as we later translate datatype constructors and recursion operators accordingly. To keep their translation as simple as possible, we chose to use a straightforward generalization of the order on non-recursive datatypes. Definition 3.1 is extended to initial fragments of recursive datatypes as follows.

Definition 3.7 (Order on Initial Datatype Fragments)**.** Let $(\alpha_1, \ldots, \alpha_h)t_j$ be a datatype, given by the general datatype specification stated at the beginning of Section 3.6. Let $r = (r_1, \ldots, r_n) \in \mathbb{N}^n$ such that $r_j > 0$. Let $r' = (r_1, \ldots, r_j - 1, \ldots, r_n)$. The *(datatype) order* $<^r_{t_j}$ *on* ${(\alpha_1, \ldots, \alpha_h)t_j}^r$ is given by

$$[\![C^j_{i_1}]\!]_{\mathcal{A},\mathcal{M}}\, x_1 \ldots x_{m^j_{i_1}} <^r_{t_j} [\![C^j_{i_2}]\!]_{\mathcal{A},\mathcal{M}}\, y_1 \ldots y_{m^j_{i_2}} \quad \text{iff}$$
$$i_1 < i_2 \vee \left[i_1 = i_2 \wedge (x_1, \ldots, x_{m^j_{i_1}}) < (y_1, \ldots, y_{m^j_{i_2}}) \right],$$

where the argument tuples are compared wrt. the lexicographic order on $\left(\sigma^j_{i,1}\right)^{r'} \times \cdots \times \left(\sigma^j_{i,m^j_{i_1}}\right)^{r'}$ that is induced by the individual (inductively defined) orders $<^{r'}_{\ldots}$ on $\left(\sigma^j_{i,1}\right)^{r'}$, …, $\left(\sigma^j_{i,m^j_{i_1}}\right)^{r'}$. For σ not a datatype, we define $<^{r'}_\sigma$ to mean the usual (presupposed) order on $[\![\sigma]\!]_{E,M}$.

Induction on $\sum_{j=1}^n r_j$ shows that $<^r_{t_j}$ is in fact well-defined. We write $<^r$ or just $<$ instead of $<^r_{t_j}$ when t_j and r are clear from the context. This definition is again generalized to instances $(\tau_1, \ldots, \tau_h)t_j$ of a datatype $(\alpha_1, \ldots, \alpha_h)t_j$ in the obvious way.

As a simple example, consider the datatype $(\alpha)\mathsf{list}$, given by the specification

datatype $(\alpha)\mathsf{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha)\mathsf{list}$.

Suppose that $[\![\alpha]\!]_{E,M} = [a_1, \ldots, a_n]$, i.e. $a_1 < \cdots < a_n$. Then

$$
\begin{aligned}
(\alpha)\mathsf{list}^1 &= \big[[\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}\big], \\
(\alpha)\mathsf{list}^2 &= \big[[\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}, [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_1\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}, \ldots, [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_n\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}\big], \\
(\alpha)\mathsf{list}^3 &= \big[[\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}, \\
&\qquad [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_1\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}, [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_1\, ([\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_1\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}), \ldots, \\
&\qquad\quad [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_1\, ([\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_n\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}), \\
&\qquad \ldots, \\
&\qquad [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_n\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}, [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_n\, ([\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_1\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}), \ldots, \\
&\qquad\quad [\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_n\, ([\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_n\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}})\big],
\end{aligned}
$$

and for arbitrary $r \in \mathbb{N}$, $|(\alpha)\mathsf{list}^r| = \sum_{k=0}^{r-1} |[\![\alpha]\!]_{E,M}|^k$, in accordance with Lemma 3.6. We can see that $[\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}$ is always the first element of $(\alpha)\mathsf{list}^r$. Thus the translation of Nil, which is a non-recursive constructor (without any arguments at all), is straightforward: it is translated as $\mathrm{Leaf}(\mathrm{uv}_1^{|(\alpha)\mathsf{list}^r|})$. The translation of Cons however is more complicated. We already mentioned that recursive constructors become partial functions: e.g. if we restrict ourselves to the initial fragment $(\alpha)\mathsf{list}^1$, then $\mathsf{Cons}\, x_\alpha$ is interpreted as a function with domain and codomain $(\alpha)\mathsf{list}^1$, but $[\![\mathsf{Cons}\, x_\alpha\, \mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}$ clearly cannot be equal to $[\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}$ (which happens to be the only element of $(\alpha)\mathsf{list}^1$), as this would violate the freeness assumption on different constructors. Our best option is to leave $[\![\mathsf{Cons}\, x_\alpha\, \mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}$ undefined. We translate an undefined value (of a datatype or ground type of size k) as a tree $\mathrm{Leaf}([\mathrm{False}, \ldots, \mathrm{False}])$ of length k.

Definition 3.8 (Undefined Leaf)**.** The *undefined leaf of length* k, written undef^k, is defined as the tree $\mathrm{Leaf}([\mathrm{False}, \ldots, \mathrm{False}])$, where $[\mathrm{False}, \ldots, \mathrm{False}] \in \mathrm{List}_{\{\mathrm{False}\}}$ is a list of length k.

Remark 3.9. Let $\sigma \in \mathrm{TyVars}$ or $\sigma = (\sigma_1, \ldots, \sigma_n)c$ with $c \in \mathrm{TyNames} \setminus \{\to\}$. Let $k = |[\![\sigma]\!]_{E,M}|$. Then $\mathrm{undef}^k \in \mathrm{Trees}(\sigma)$.

Proof. Immediate, using Def. 2.58. □

Remark 3.10. Let $\mathrm{undef}^k \in \mathrm{Trees}(\sigma)$, and let A be a truth assignment. Then $[\![\mathrm{undef}^k]\!]_{\sigma,A}$ is undefined.

Proof. Immediate, using Def. 2.60. □

Thus Cons, if we restrict ourselves to the initial fragment $(\alpha)\mathsf{list}^1$, is translated as the tree $\mathrm{Node}([\mathrm{Node}([\mathrm{undef}^1]), \ldots, \mathrm{Node}([\mathrm{undef}^1])])$, where the root node has n children (one for each element of $[\![\alpha]\!]_{E,M}$), and each child node in turn has one child leaf of length 1 (simply because $|(\alpha)\mathsf{list}^1| = 1$). If we consider $(\alpha)\mathsf{list}^2$ instead, Cons is again translated as a tree of height 3 and width n at the root, but now each child node has $|(\alpha)\mathsf{list}^2| = n + 1$ child leafs, each of length $n + 1$. All but the first child leaf of each node (which corresponds to $[\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}\, a_i\, [\![\mathsf{Nil}]\!]_{\mathcal{A},\mathcal{M}}$, for the respective $1 \le i \le n$) is the undefined leaf. Thus Cons in this case is translated as the tree

$$
\begin{aligned}
&\mathrm{Node}([\mathrm{Node}([\mathrm{Leaf}(\mathrm{uv}_2^{n+1}), \mathrm{undef}^{n+1}, \ldots, \mathrm{undef}^{n+1}]), \\
&\qquad \ldots, \\
&\qquad \mathrm{Node}([\mathrm{Leaf}(\mathrm{uv}_{n+1}^{n+1}), \mathrm{undef}^{n+1}, \ldots, \mathrm{undef}^{n+1}])]).
\end{aligned}
$$

The case $(\alpha)\mathsf{list}^3$ is still more complicated, since Cons now yields defined values also when applied to lists of length 1, which are themselves in the range of $[\![\mathsf{Cons}]\!]_{\mathcal{A},\mathcal{M}}$. Again Cons is translated as a tree of height 3 and width n at the root. Each of the n child nodes now has $|(\alpha)\mathsf{list}^3| = n^2 + n + 1$ child leafs, each of length $n^2 + n + 1$. The first child leaf in each node corresponds to the Nil argument (which was mapped to a defined value by Cons already when we considered $(\alpha)\mathsf{list}^2$), and leafs with index $2 + k(n + 1)$, for $0 \leq k \leq n - 1$, correspond to argument lists of length 1, while all other leafs correspond to argument lists of length 2 (and are thus mapped to undefined values). More precisely, let $N = n^2 + n + 1$. Then Cons is translated as the tree

$$\begin{aligned}
&\mathrm{Node}([\mathrm{Node}([\mathrm{Leaf}(\mathrm{uv}_2^N), \mathrm{Leaf}(\mathrm{uv}_3^N), \mathrm{undef}^N, \ldots, \mathrm{undef}^N,\ \ldots,\\
&\qquad\qquad\qquad\qquad \mathrm{Leaf}(\mathrm{uv}_{n+2}^N), \mathrm{undef}^N, \ldots, \mathrm{undef}^N]),\\
&\quad \ldots,\\
&\quad \mathrm{Node}([\mathrm{Leaf}(\mathrm{uv}_{n^2+1}^N), \mathrm{Leaf}(\mathrm{uv}_{n^2+2}^N), \mathrm{undef}^N, \ldots, \mathrm{undef}^N,\ \ldots,\\
&\qquad\qquad\qquad\qquad \mathrm{Leaf}(\mathrm{uv}_N^N), \mathrm{undef}^N, \ldots, \mathrm{undef}^N])]).
\end{aligned}$$

Another, perhaps even more instructive, example is given by two mutually recursive datatypes

$$\begin{aligned}
\textbf{datatype}\ X &= A \mid B\,X \mid C\,Y\\
\textbf{and}\ Y &= D\,X \mid E\,Y \mid F.
\end{aligned}$$

Here from Def. 3.7 it follows that $X^{(0,r)} = Y^{(r,0)} = \emptyset$ for any $r \in \mathbb{N}$, and

$$\begin{aligned}
X^{(1,0)} &= [[\![A]\!]_{\mathcal{A},\mathcal{M}}],\\
Y^{(0,1)} &= [[\![F]\!]_{\mathcal{A},\mathcal{M}}],\\
X^{(2,0)} &= [[\![A]\!]_{\mathcal{A},\mathcal{M}}, [\![B\,A]\!]_{\mathcal{A},\mathcal{M}}],\\
Y^{(0,2)} &= [[\![E\,F]\!]_{\mathcal{A},\mathcal{M}}, [\![F]\!]_{\mathcal{A},\mathcal{M}}],\\
X^{(1,1)} &= [[\![A]\!]_{\mathcal{A},\mathcal{M}}, [\![C\,F]\!]_{\mathcal{A},\mathcal{M}}],\\
Y^{(1,1)} &= [[\![D\,A]\!]_{\mathcal{A},\mathcal{M}}, [\![F]\!]_{\mathcal{A},\mathcal{M}}],\\
X^{(2,1)} &= [[\![A]\!]_{\mathcal{A},\mathcal{M}}, [\![B\,A]\!]_{\mathcal{A},\mathcal{M}}, [\![B\,(C\,F)]\!]_{\mathcal{A},\mathcal{M}}, [\![C\,(D\,A)]\!]_{\mathcal{A},\mathcal{M}}, [\![C\,F]\!]_{\mathcal{A},\mathcal{M}}],\\
Y^{(2,1)} &= [[\![D\,A]\!]_{\mathcal{A},\mathcal{M}}, [\![D\,(B\,A)]\!]_{\mathcal{A},\mathcal{M}}, [\![F]\!]_{\mathcal{A},\mathcal{M}}],\\
X^{(1,2)} &= [[\![A]\!]_{\mathcal{A},\mathcal{M}}, [\![C\,(E\,F)]\!]_{\mathcal{A},\mathcal{M}}, [\![C\,F]\!]_{\mathcal{A},\mathcal{M}}],\\
Y^{(1,2)} &= [[\![D\,A]\!]_{\mathcal{A},\mathcal{M}}, [\![D\,(C\,F)]\!]_{\mathcal{A},\mathcal{M}}, [\![E\,(D\,A)]\!]_{\mathcal{A},\mathcal{M}}, [\![E\,F]\!]_{\mathcal{A},\mathcal{M}}, [\![F]\!]_{\mathcal{A},\mathcal{M}}],\\
&\ldots.
\end{aligned}$$

The datatype constructors A_X, $B_{X\to X}$, $C_{Y\to X}$, $D_{X\to Y}$, $E_{Y\to Y}$, and F_Y must be translated to trees accordingly.

Our algorithm to achieve this for an arbitrary constructor C_i^j with (possibly recursive) argument types $\sigma_{i,1}^j, \ldots, \sigma_{i,m_i^j}^j$ proceeds as follows. We first compute the number of elements of the datatype that are generated by constructors $C_{i'}^j$, with $i' < i$, using Lemma 3.6. This gives the index of the first element generated by constructor C_i^j. Next we compute a list of all elements

of $\left(\sigma_{i,1}^j\right)^r$, and also of $\left(\sigma_{i,1}^j\right)^{(r_1,\ldots,r_j-1,\ldots,r_n)}$, in their canonical representation (similar to the lists shown in the above example). Note that we do not need to know the translation of datatype constructors to compute these lists (which would lead to infinite recursion), but we are merely working with HOL terms. Nevertheless, care must be taken to obtain an implementation that always terminates. To compute the list of canonical representations of elements of a datatype, we consider the datatype's constructors (together with their argument types), and form their application to all possible combinations of argument terms (in the order given by Def. 3.7). The argument terms are obtained by a recursive application of the enumeration function, where the size of the initial fragment of the datatype under consideration has been reduced by 1. (The 0-th initial fragment of each datatype is empty.) The constructor C_i^j now maps those elements of $\left(\sigma_{i,1}^j\right)^r$ that are already present in $\left(\sigma_{i,1}^j\right)^{(r_1,\ldots,r_j-1,\ldots,r_n)}$ to subtrees which may contain defined values. For these elements, we proceed recursively over the remaining argument types $\sigma_{i,2}^j$, …, $\sigma_{i,m_i^j}^j$. Elements of $\left(\sigma_{i,1}^j\right)^r$ that are not present in $\left(\sigma_{i,1}^j\right)^{(r_1,\ldots,r_j-1,\ldots,r_n)}$ are mapped to subtrees which contain undefined values only.

When the SAT solver returns a Boolean model, we need to print elements of datatypes that are given by their index (according to Def. 3.7) in a user-readable form, i.e. in their canonical representation involving (possibly nested) constructor applications. Of course the internals of our enumeration of datatype elements, i.e. the order that we use on datatype fragments, should not matter to the user. To print an element of a datatype $(\alpha_1, \ldots, \alpha_h)t_j$, we translate the datatype's constructors C_1^j, …, $C_{k_j}^j$ to trees, and search for a tree that yields the element with the given index. Since every element of the datatype is generated by some constructor, we will find such a tree. The position of the element within the tree determines indices for the constructor's arguments (in their respective types). To print arguments whose type is again given by a datatype, this algorithm is applied recursively. Termination is guaranteed because every datatype element can be written with a finite number of constructor applications. We could implement the printing of datatype elements more efficiently, by computing the correct constructor and argument indices using an algorithmic "inverse" of Def. 3.7 (which allows us to compute an element's index from its canonical representation), but the current implementation has the advantage that it is completely independent of the order that we use on datatype fragments.

3.6.3 Recursive Functions

Isabelle/HOL provides convenient ways to define functions by primitive recursion on datatypes: e.g. the addition function on natural numbers may be defined as

consts add :: nat → nat → nat
primrec
  add 0 y = y
  add (Suc x) y = Suc (add x y).

Internally, such a definition is recast in terms of the recursion operator(s) on the datatype, which are provided by the datatype package. The above, for example, becomes

$$\mathsf{add} = \mathsf{nat\_rec}\,(\lambda y_{\mathsf{nat}}.\, y)\,(\lambda x_{\mathsf{nat}}\, f_{\mathsf{nat}\to\mathsf{nat}}\, y_{\mathsf{nat}}.\, \mathsf{Suc}\,(f\, y)).$$

(On a side note, the ability to perform recursion at higher-order types allows one to even define functions which are not (first-order) primitive recursive. The Ackermann function [2], for example, may be defined as

consts Ack :: nat → nat → nat
primrec
Ack 0 y = Suc y
Ack (Suc x) y = (Ack x)^(Suc y) (Suc 0),

where $f\hat{}n$ is defined as n-fold application of f.)

In general, the datatype specification stated at the beginning of Section 3.6 introduces recursion operators $\mathsf{rec}_1, \ldots, \mathsf{rec}_n$ of respective type

$$\begin{aligned}
&(\sigma^1_{1,1} \to \cdots \to \sigma^1_{1,m^1_1} \to \tau^1_{1,1} \to \cdots \to \tau^1_{1,m^1_1} \to \beta_1) \to \cdots \to \\
&\quad (\sigma^1_{k_1,1} \to \cdots \to \sigma^1_{k_1,m^1_{k_1}} \to \tau^1_{k_1,1} \to \cdots \to \tau^1_{k_1,m^1_{k_1}} \to \beta_1) \to \\
&\quad \to \cdots \to \\
&(\sigma^n_{1,1} \to \cdots \to \sigma^n_{1,m^1_1} \to \tau^n_{1,1} \to \cdots \to \tau^n_{1,m^1_1} \to \beta_n) \to \cdots \to \\
&\quad (\sigma^n_{k_n,1} \to \cdots \to \sigma^n_{k_n,m^n_{k_n}} \to \tau^n_{k_n,1} \to \cdots \to \tau^n_{k_n,m^n_{k_n}} \to \beta_n) \to \\
&\qquad\qquad (\alpha_1, \ldots, \alpha_h)t_j \to \beta_j
\end{aligned}$$

(for $1 \le j \le n$), where type $\tau^{j'}_{i,i'}$ is present if and only if $\sigma^{j'}_{i,i'}$ contains a mutually recursive datatype, and in this case $\tau^{j'}_{i,i'} := \theta(\sigma^{j'}_{i,i'})$, where θ is given by

1. $\theta((\alpha_1, \ldots, \alpha_h)t_{j'}) := \beta_{j'}$ (for $1 \le j' \le n$), and
2. $\theta(\sigma_1 \to \sigma_2) := \sigma_1 \to \theta(\sigma_2)$.

Admissibility of $\sigma^{j'}_{i,i'}$ ensures that θ eliminates all recursive datatype occurrences. (Indirect recursion is implemented by the datatype package via mutually recursive datatypes.) The internal definition of the recursion operators is based on an inductive definition of their graphs, from which the operator functions are obtained using Hilbert's choice [25]. Using this internal definition is again prohibitively expensive (in terms of translation time), so instead we have implemented a more direct translation which respects the relevant recursion equations for these operators. The relevant equations for operator rec_j are

$$\begin{aligned}
\mathsf{rec}_j\, f^1_1 \ldots f^1_{k_1} \ldots f^n_1 \ldots f^n_{k_n}\, (C^j_1\, x^j_{1,1} \ldots x^j_{1,m^j_1}) &= f^j_1\, x^j_{1,1} \ldots x^j_{1,m^j_1}\, y^j_{1,1} \ldots y^j_{1,m^j_1}, \\
&\vdots \\
\mathsf{rec}_j\, f^1_1 \ldots f^1_{k_1} \ldots f^n_1 \ldots f^n_{k_n}\, (C^j_{k_j}\, x^j_{k_j,1} \ldots x^j_{k_j,m^j_{k_j}}) &= f^j_{k_j}\, x^j_{k_j,1} \ldots x^j_{k_j,m^j_{k_j}}\, y^j_{k_j,1} \ldots y^j_{k_j,m^j_{k_j}},
\end{aligned}$$

where argument $y^j_{i,i'}$ is present if and only if type $\tau^{j'}_{i,i'}$ is present in the type of the recursion operator, and in this case $y^j_{i,i'} := \Theta(x^j_{i,i'})$, where Θ is given by

1. $\Theta(x_{(\alpha_1,\ldots,\alpha_h)t_{j'}}) := \mathsf{rec}_{j'}\, f^1_1 \ldots f^1_{k_1} \ldots f^n_1 \ldots f^n_{k_n}\, x_{(\alpha_1,\ldots,\alpha_h)t_{j'}}$ (for $1 \le j' \le n$), and

2. $\Theta(x_{\sigma_1\to\sigma_2}) := \lambda z_{\sigma_1}.\,\Theta((x_{\sigma_1\to\sigma_2}\, z_{\sigma_1})_{\sigma_2})$..

Note that $\Theta(x_\sigma)$ has type $\theta(\sigma)$, as required for the above recursion equations to be type-correct. We assume that trees for the arguments $f^1_1, \ldots, f^1_{k_1}, \ldots, f^n_1, \ldots, f^n_{k_n}$ are given (otherwise we use η-expansion to obtain sufficiently many arguments to rec_j), and focus on translating $\mathsf{rec}_j\, f^1_1 \ldots f^1_{k_1} \ldots f^n_1 \ldots f^n_{k_n}$ as a function of type $(\alpha_1, \ldots, \alpha_h)t_j \to \beta_j$ then.

The case where $(\alpha_1, \ldots, \alpha_h)t_j$ is a non-recursive datatype is simple. Because of Def. 3.7, leafs (or subtrees in case β_j is a function type) in $f^j_1, \ldots, f^j_{k_j}$ directly correspond to results of the recursion operator applied to elements of the datatype, in the order given. When $(\alpha_1, \ldots, \alpha_h)t_j$ is a recursive datatype, we have to take into account that constructors with recursive arguments become partial functions. Subtrees in f^j_i that are obtained for a combination of constructor arguments $x^j_{i,1}, \ldots, x^j_{i,m^j_i}$ which are mapped to "undefined" by constructor C^j_i (rather than to a defined element of the datatype) must be ignored.

The hard part is to obtain translations for the recursive arguments $y^j_{i,1}, \ldots, y^j_{i,m^j_i}$ (as far as they are present). This requires that we already know the translation of recursion operators $\mathsf{rec}_{j'}$ of mutually recursive datatypes, at least for the constructor's arguments $x^j_{i,1}, \ldots, x^j_{i,m^j_i}$. Now Def. 3.7 does not (in general) imply $x^j_{i,i'} < C^j_i\, x^j_{i,1} \ldots x^j_{i,m^j_i}$, where $x^j_{i,i'}$ is a recursive argument. Therefore care must be taken again to obtain a terminating implementation. We recursively translate $\mathsf{rec}_{j'}\, f^1_1 \ldots f^1_{k_1} \ldots f^n_1 \ldots f^n_{k_n}\, x$ only for those arguments x that are of interest, rather than $\mathsf{rec}_{j'}\, f^1_1 \ldots f^1_{k_1} \ldots f^n_1 \ldots f^n_{k_n}$ in its entirety. Termination is guaranteed then because each recursive invocation removes one constructor application from the argument.

Note that recursive functions (like addition and multiplication on nat, or list concatenation on type $(\alpha)\mathsf{list}$), similar to datatype constructors with recursive arguments, will frequently become partial functions when we restrict the model to some initial fragment for their recursive codomain type. (E.g. $\mathsf{add}\, m\, n$ is defined in nat^r only if $m+n < r$, and undefined otherwise.) In this case, the translation of equality that was discussed in Section 3.2 will cause our translation of the defining equation for the recursive function to yield a tree whose meaning is either undefined or false, but never true—regardless of how the SAT solver chooses to interpret the function. Therefore if we add the defining equation to the set of relevant axioms, the SAT solver will unfortunately not be able to find a model that satisfies all relevant axioms.

This is not a problem when we simply unfold the definition of recursive functions, replacing the function constant by its definition in terms of the recursion operators. Otherwise however, it makes sense to translate defining equations with a different notion of equality. In contrast to what we said about equality in Section 3.2, we now consider two trees for the left-hand side and right-hand side of a definition equal iff they both denote the same (possibly partial) function. (This is the usual notion of equality on the space of partial functions.) Two "undefined" elements are considered equal to each other. This is more in the spirit of a definition, which should allow one to use the (tree for) the left-hand side as an equivalent abbreviation for the (tree for) the right-hand side, without changing the meaning of a formula from true to undefined. Note that we must use this notion of equality for definitions only, but not for equality operators in general: we do not want e.g. $\mathsf{Suc}\,0 = \mathsf{Suc}\,(\mathsf{Suc}\,0)$ to hold in any model, not even when we are considering nat^1 only (where both sides of this equation are undefined), as this would violate freeness of constructors.

3.7 Sets and Records

Isabelle/HOL provides a type $(\alpha)\mathsf{set}$, containing sets whose elements are of type α. This type is isomorphic to (and, wrt. to the translation, is treated as) $\alpha \to \mathsf{bool}$. The constant $\mathsf{Collect}$ (set comprehension) of type $(\alpha \to \mathsf{bool}) \to (\alpha)\mathsf{set}$ is simply ignored when encountered with an argument, and translated as the identity function otherwise. The constant op : (set membership) of type $\alpha \to (\alpha)\mathsf{set} \to \mathsf{bool}$, when translated, applies its second argument to its first argument, i.e. $x : P$ is translated as $P\,x$. Eta expansion is used when op : occurs with less than two arguments. This translation clearly satisfies the two relevant axioms

$$\text{mem\_Collect\_eq}: \quad (a : \{x.\, P\,x\}) = P\,a$$

and

$$\text{Collect\_mem\_eq}: \quad \{x.\, x : A\} = A,$$

which are declared in HOL/Set.thy. (The notation $\{x.\, P\,x\}$ is merely syntactic sugar for $\mathsf{Collect}\,(\lambda x.\, P\,x)$, and op : is written in infix notation as usual.) All other operations on sets, e.g. union, intersection, and the power set operator, as well as related constants, e.g. $\{\}$ (the empty set) and UNIV (the set containing every element of the underlying type) are derived concepts that can be treated by considering their respective definitions.

Moreover, Isabelle/HOL offers a simple interface to define extensible records with structural subtyping [121]. A general record definition has the form

$$\begin{aligned}&\textbf{record}\ (\alpha_1, \ldots, \alpha_k)r\ =\ \sigma +\\ &\quad F_1\ ::\ \sigma_1\\ &\quad \ldots\\ &\quad F_n\ ::\ \sigma_n,\end{aligned}$$

where α_1, ..., α_k are type variables, σ is an (entirely optional) instance of a previously defined parent record type (with $\text{TyVars}(\sigma) \subseteq \{\alpha_1, \ldots, \alpha_k\}$), and F_1, ..., F_n are the field names (annotated with their respective types σ_1, ..., σ_n) of the new record type r. Each field type σ_i may again contain at most the type variables α_1, ..., α_k.

Internally, such a record definition is translated into a set of type and constant definitions (see Section 3.4). The new record type is defined to be isomorphic to the cartesian product $\sigma \times \sigma_1 \times \cdots \times \sigma_n \times \mathsf{unit}$. The final unit component (denoting a type with a single element) is added for technical reasons, as an instance of a polymorphic *more* field to achieve extensibility of the record type. In addition, an accessor function F_i of type $(\alpha_1, \ldots, \alpha_k)r \to \sigma_i$ is defined for each field of the record type.

Currently we translate records simply by considering the generated type and constant definitions, without further special treatment. This could be improved by making immediate use of the fact that record types are isomorphic to cartesian products. Treating them as cartesian products directly would avoid introducing an isomorphic type copy with its abstraction and representation functions, but this—although simple in theory—is future work.

3.8 HOLCF

HOLCF [120, 141], as it has been implemented in Isabelle, is the definitional extension of Church's Higher-Order Logic with Scott's Logic for Computable Functions. The logic supports standard domain theory (in particular fixed-point reasoning and recursive domain equations), but also coinductive arguments about lazy datatypes. A detailed description can be found in [140].

HOLCF, as a definitional extension of HOL, only makes use of concepts that we have discussed before, in particular of axiomatic type classes. Hence in principle no special treatment is necessary. However, the treatment of certain class axioms can be optimized because they trivially hold for *every* finite model. Central to HOLCF are the notions of *partial order* and *chain*.

Definition 3.11 (Partial Order)**.** A function $\sqsubseteq_{\alpha\to\alpha\to\mathsf{bool}}$ is a *partial order* iff $\sqsubseteq$ is reflexive (i.e. $\forall x.\, x \sqsubseteq x$), antisymmetric (i.e. $\forall x, y.\, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$), and transitive (i.e. $\forall x, y, z.\, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$).

Definition 3.12 (Chain)**.** Let $\sqsubseteq_{\alpha\to\alpha\to\mathsf{bool}}$ be a partial order. A function $f_{\mathsf{nat}\to\alpha}$ is a *chain* (wrt. $\sqsubseteq$) iff $\forall i.\, f\, i \sqsubseteq f\,(\mathsf{Suc}\, i)$.

When we consider a model where the naturals are restricted to an initial fragment, e.g. $\{0, \ldots, r\}$, we require $f\, i \sqsubseteq f\,(\mathsf{Suc}\, i)$ to hold only for all i where $\mathsf{Suc}\, i$ is defined, i.e. for all $i \in \{0, \ldots, r-1\}$.

Definition 3.13 (Chain Maximum)**.** A chain f *contains its maximum at position* i iff $\forall j.\, i \leq j \implies f\, i = f\, j$.

Definition 3.14 (Finite Chain)**.** A chain is *finite* iff it contains its maximum (at some position).

The following lemma is immediate for the kind of finite models considered by our translation to propositional logic.

Lemma 3.15. *When the meaning of* nat *is given by an initial fragment* $\mathsf{nat}^r = \{0, \ldots, r-1\}$ *(for some* $r > 0$*), every chain is finite.*

Proof. In such a model, every chain contains its maximum at position $r - 1$. □

Furthermore, the maximum is also the chain's least upper bound.

Definition 3.16 (Upper Bound)**.** A chain f has *upper bound* x iff $\forall i.\, f\, i \sqsubseteq x$.

Definition 3.17 (Least Upper Bound)**.** A chain f has *least upper bound* x iff f has upper bound x, and whenever f has upper bound y, then $x \sqsubseteq y$.

Lemma 3.18. *A chain* f *that contains its maximum at position* i *has least upper bound* $f\, i$.

Proof. For $j < i$, we have $f\, j \sqsubseteq f\, i$ because f is a chain, and $\sqsubseteq$ is transitive. For $i \leq j$, $f\, i = f\, j$ because f contains its maximum at position i. Hence $\forall j.\, f\, j \sqsubseteq f\, i$, so f has upper bound $f\, i$.

Now suppose that f has upper bound y. Then $\forall j.\, f\, j \sqsubseteq y$, hence in particular $f\, i \sqsubseteq y$. □

A partial order is said to be *complete* iff every chain has a least upper bound.

Definition 3.19 (Complete Partial Order)**.** A partial order $\sqsubseteq_{\alpha\to\alpha\to\mathsf{bool}}$ is *complete* iff every chain $f_{\mathsf{nat}\to\alpha}$ has a least upper bound (wrt. $\sqsubseteq$).

Lemma 3.20. *When the meaning of* nat *is given by an initial fragment* $\mathsf{nat}^r = \{0, \ldots, r-1\}$ *(for some* $r > 0$*), every partial order is complete.*

Proof. This is an easy corollary of Lemma 3.15 and Lemma 3.18. □

Likewise we can show that every partial order in a finite model is *chain-finite*.

Definition 3.21 (Chain-Finite)**.** A partial order $\sqsubseteq_{\alpha\to\alpha\to\mathsf{bool}}$ is *chain-finite* iff every chain $f_{\mathsf{nat}\to\alpha}$ is finite (wrt. $\sqsubseteq$).

Lemma 3.22. *When the meaning of* nat *is given by an initial fragment* $\mathsf{nat}^r = \{0, \ldots, r-1\}$ *(for some* $r > 0$*), every partial order is chain-finite.*

Proof. This is an immediate consequence of Lemma 3.15. □

These results perhaps raise the question of how interesting the HOLCF setting really is when we restrict ourselves to finite models, but first and foremost they show that the class axioms characterizing complete and chain-finite partial orders do not need to be translated to propositional logic and passed to the SAT solver. They can safely be ignored, even if they had otherwise been relevant for a given formula: any model that the SAT solver may find will automatically satisfy these axioms anyway.

3.9 Conclusion

We have discussed some optimizations of the translation from HOL to propositional logic that was presented in Chapter 2, and we have shown how the translation can be extended from the core HOL language (which is based on the simply typed λ-calculus) to the various definitional mechanisms and logical features that are present in the Isabelle/HOL implementation of higher-order logic. Both aspects are important steps towards a practically useful tool for (counter-)model generation, which should ideally cover the full input language of the theorem prover, while being reasonably efficient.

It is not enough to aim; you must hit.
Italian proverb.

Chapter 4

Case Studies

We have applied Isabelle's finite model generation techniques to obtain a correctness proof for a security protocol, counterexamples to conjectures about probabilistic programs, and a Sudoku solver. The details are presented in this chapter.

4.1 Introduction

The model generation algorithm presented in Chapters 2 and 3, when applied to formulae of higher-order logic, has non-elementary complexity. Consider for instance HOL terms $f_{\sigma\to\alpha}$, where σ is of the form bool, $\mathsf{bool} \to \mathsf{bool}$, $(\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool}$, etc. Let n be the total number of bool type constructors in σ. Then the meaning of σ (wrt. any standard type model, cf. Def. 2.14) is given by a set of size $2 \uparrow\uparrow n$. (Here $\uparrow$ is Knuth's arrow notation [91], i.e. $2 \uparrow\uparrow n = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n}$.) Consequently, $f_{\sigma\to\alpha}$ is translated as a tree of width $2 \uparrow\uparrow n$ (cf. Section 2.3.3), which (for $n \in \mathbb{N}$) is a rather fast growing function: already $2 \uparrow\uparrow 4 = 65,536$, and $2 \uparrow\uparrow 5$ has $19,729$ digits [151]. It is therefore easy to give HOL formulae for which the translation to propositional logic fails due to time or memory constraints.

Moreover, even if the translation succeeds, the SAT solver's task of finding a satisfying assignment for the resulting propositional formula is of course NP-complete [44], hence of exponential complexity (in the number of Boolean variables) unless P = NP. It is therefore also quite easy to find HOL formulae for which all current SAT solvers will run out of resources, despite the translation to propositional logic being trivial: any sufficiently complex propositional formula will do.

This may seem discouraging, but it does not necessarily render the model generation algorithm useless for practical purposes. In fact, higher-order types like the one above rarely occur in

practice. We do not aim to make a strong case for Daniel Jackson's "small scope hypothesis", which states that most design flaws in system models can be illustrated by small instances [9], but we want to demonstrate the utility of our algorithm further by applying it to several (small to medium-sized) case studies. In this chapter, we briefly consider the RSA-PSS security protocol (Section 4.2), counterexamples for probabilistic programs (Section 4.3), and Sudoku puzzles (Section 4.4).

4.2 The RSA-PSS Security Protocol

A significant amount of research is concerned with the formal modeling and verification of secure systems, and security protocols in particular. In Isabelle, Larry Paulson's inductive approach [135] has been used to verify various protocols, e.g. the Transport Layer Security (TLS) protocol [137]. In this section, we shall use an encoding of protocols as first-order clauses that is based on the popular Dolev-Yao threat model [51]. The Dolev-Yao model assumes a worst-case scenario, where the attacker can intercept and possibly alter any message in any way (within his computational power). Central is a unary predicate knows that describes the attacker's knowledge. Jürjens [85, 86] sketches a mechanic translation of protocols (given as UML [80] sequence diagrams) into this first-order encoding, and he discusses how automated first-order theorem provers can be used to obtain attacks on protocols. We want to show how (counter-)model generation can be used to prove protocols secure.

4.2.1 Abstract Protocol Formalization

We investigate the RSA-PSS security protocol [145], a digital signature scheme that follows the usual "hash-then-sign" paradigm. RSA refers to the (now classic) algorithm for public-key cryptography devised by Ron Rivest, Adi Shamir, and Leonard Adleman [142, 143]. PSS stands for "Probabilistic Signature Scheme", first described by Mihir Bellare and Phillip Rogaway [20]. Starting with a message M that is to be signed, the RSA-PSS protocol—at a very abstract level—proceeds in two steps:

1. Apply a one-way hash function to the message M to produce an encoded message $\mathsf{hashpss}\, M$.
2. Apply a signature function to the encoded message, using a private key k, to produce a signature $\mathsf{sign}\,(\mathsf{hashpss}\, M)\, k$.

The message M is then sent together with its signature, $\mathsf{sign}\,(\mathsf{hashpss}\, M)\, k$. The signature can be verified by the receiver using the sender's public key k^{-1}. (RSA-PSS is not an encryption algorithm. If the contents of M need to be kept secret, any such algorithm can be used to encrypt M before transmission. The ciphertext is then decrypted by the receiver before signature verification.)

A detailed Isabelle/HOL formalization of the RSA-PSS protocol (by Christina Lindenberg and Kai Wirt) is available [96]. For our purposes however, it will be sufficient to model hashing and signing as uninterpreted functions. We take these functions as primitives, without aiming to verify the mathematics that is underlying their implementation. A third function, conc, forms the concatenation of two messages. We write $a :: b$ for $\mathsf{conc}\, a\, b$, and $\{M\}_k$ for $\mathsf{sign}\, M\, k$.

A naive implementation of the RSA signature function suffers from an undesirable homomorphism property, which allows to compute the signature of concatenated messages from signatures for their components (and vice versa) without knowledge of the private key k:

$$\{a :: b\}_k = \{a\}_k :: \{b\}_k.$$

Our goal is rather simple: we want to show that PSS hashing (when considered a primitive) breaks this homomorphism property, thereby improving security of the signature scheme. An early version of this analysis is described in [180]. The first-order clauses that model the RSA-PSS protocol and the abilities of a potential Dolev-Yao attacker are as follows.

1. The attacker can decrypt, provided he knows the decryption key:

$$\forall A, K.\, \mathsf{knows}\,\{A\}_K \wedge \mathsf{knows}\,K^{-1} \Longrightarrow \mathsf{knows}\,A.$$

2. We only assume the signature function to be cryptographically strong, but not the hash function. The attacker can recover hashed messages:

$$\forall A.\, \mathsf{knows}\,(\mathsf{hashpss}\,A) \Longrightarrow \mathsf{knows}\,A.$$

3. The attacker can concatenate and encrypt:

$$\forall A, B.\, \mathsf{knows}\,A \wedge \mathsf{knows}\,B \Longrightarrow \mathsf{knows}\,A :: B \wedge \mathsf{knows}\,\{A\}_B.$$

4. The attacker can decompose messages:

$$\forall A, B.\, \mathsf{knows}\,A :: B \Longrightarrow \mathsf{knows}\,A \wedge \mathsf{knows}\,B.$$

5. The attacker can hash:

$$\forall A.\, \mathsf{knows}\,A \Longrightarrow \mathsf{knows}\,(\mathsf{hashpss}\,A).$$

6. The signature function satisfies the homomorphism property discussed above (1):

$$\forall A, B, K.\, \mathsf{knows}\,\{A :: B\}_K \Longrightarrow \mathsf{knows}\,\{A\}_K \wedge \mathsf{knows}\,\{B\}_K.$$

7. The signature function satisfies the homomorphism property discussed above (2):

$$\forall A, B, K.\, \mathsf{knows}\,\{A\}_K \wedge \mathsf{knows}\,\{B\}_K \Longrightarrow \mathsf{knows}\,\{A :: B\}_K.$$

We consider a protocol run where the message $a :: b$ is signed with some private key k according to the RSA-PSS protocol and then transmitted over an insecure connection, so the attacker also has initial knowledge of this message and its signature. In addition, the attacker knows the sender's public key.

8.

$$\mathsf{knows}\,(a :: b) :: \{\mathsf{hashpss}\,(a :: b)\}_k.$$

9.

$$\mathsf{knows}\, k^{-1}.$$

The knows predicate can be seen as an inductive characterization of the attacker's knowledge. Starting from some initial knowledge, the attacker can extend his knowledge with every message exchange of the protocol, and by computing new facts (within his computational abilities) from facts that he knows already. However, we do not ensure that knows actually denotes a least fixed point. The knows predicate is merely an upper bound on the attacker's knowledge; an interpretation where the attacker knows everything is allowed, but would be uninteresting.

If we consider a modified protocol without PSS hashing (by replacing Clause (8) with $\mathsf{knows}\,(a :: b) :: \{a :: b\}_k$), then it is easy to show that the attacker can forge the signature for message $b :: a$, using the homomorphism property that we assumed for the signature function. A machine-checked version of this result is shown below.

```
lemma
  assumes remove_sign: "∀A K. knows (sign A K) ∧ knows (invs K) ⟶ knows A"
  and remove_hashpss: "∀A. knows (hashpss A) ⟶ knows A"
  and construct_msg: "∀A B. knows A ∧ knows B ⟶
    knows (conc A B) ∧ knows (sign A B)"
  and deconstruct_msg: "∀A B. knows (conc A B) ⟶ knows A ∧ knows B"
  and hashpss: "∀A. knows A ⟶ knows (hashpss A)"
  and sign_hom_1: "∀A B K. knows (sign (conc A B) K) ⟶
    knows (sign A K) ∧ knows (sign B K)"
  and sign_hom_2: "∀A B K. knows (sign A K) ∧ knows (sign B K) ⟶
    knows (sign (conc A B) K)"
  and modified_protocol_msg: "knows (conc (conc a b) (sign (conc a b) k))"
  and public_key: "knows (invs k)"
shows "knows (sign (conc b a) k)"
proof -
  from protocol_message deconstruct_msg sign_hom_1
    have sign_b_k: "knows (sign b k)" by blast
  from modified_protocol_msg deconstruct_msg sign_hom_1
    have sign_a_k: "knows (sign a k)" by blast
  from sign_b_k sign_a_k sign_hom_2 show ?thesis by blast
qed
```

However, from Clauses (1)–(9), can we conclude that the attacker knows $\{\mathsf{hashpss}\,(b :: a)\}_k$? (In our formalization, it is certainly *possible* that the attacker knows this signature—since the knows predicate, as discussed above, may be true everywhere—, but is it also *necessary*?) An encoding of this problem in TPTP syntax [155] is shown in Figure 4.1. Various first-order provers, including E-SETHEO [60] and SPASS [169], fail to provide meaningful output for this problem. The Isabelle/HOL encoding of the same problem (which, apart from Clause (8) and the conjecture, is identical to the problem shown above) can be used directly as input to the model generation algorithm:

```
lemma
  assumes remove_sign: "∀A K. knows (sign A K) ∧ knows (invs K) ⟶ knows A"
  and remove_hashpss: "∀A. knows (hashpss A) ⟶ knows A"
  and construct_msg: "∀A B. knows A ∧ knows B ⟶
    knows (conc A B) ∧ knows (sign A B)"
  and deconstruct_msg: "∀A B. knows (conc A B) ⟶ knows A ∧ knows B"
```

```
  and hashpss: "∀A. knows A ⟶ knows (hashpss A)"
  and sign_hom_1: "∀A B K. knows (sign (conc A B) K) ⟶
    knows (sign A K) ∧ knows (sign B K)"
  and sign_hom_2: "∀A B K. knows (sign A K) ∧ knows (sign B K) ⟶
    knows (sign (conc A B) K)"
  and protocol_msg: "knows (conc (conc a b) (sign (hashpss (conc a b)) k))"
  and public_key: "knows (invs k)"
shows "knows (sign (hashpss (conc b a)) k)"
  apply (cut_tac prems)
  refute
oops
```

(Isabelle's **oops** command aborts a failed proof attempt. No theorem is established in this case.) Using zChaff [119] as the underlying SAT solver, our Isabelle implementation of model generation finds a counterexample with 4 elements in about 3 seconds on a current personal computer. (This implies that there are no counterexamples with less than 4 elements, cf. Section 2.4.2.) The model—in text form, as rendered by Isabelle—is shown in Figure 4.2. One can easily verify by hand that the model indeed satisfies Clauses (1)–(9) and the negation of the conjecture. Other model generators, e.g. Paradox [41], find similar counterexamples.

4.2.2 Avoiding Confusion

There is a subtlety present in our formalization. The conc function is supposed to denote the concatenation of messages; in the model however, conc is (necessarily, since the model is finite) not injective. This means that certain intuitively different messages are identified in the model. More generally, if we view the functions hashpss, sign and conc, as well as the constants a and b, as free generators of a term algebra of messages, then the model violates the *no confusion* property [70], which would require it to identify only those terms that are syntactically equal. To solve this problem, we note that Clauses (1)–(9) are all (equivalent to one or two) *strict Horn clauses*. The following definitions are taken from [71].

Definition 4.1 (Horn Formula)**.** A quantifier-free formula is said to be a *Horn formula* iff it has one of the three forms

1. φ,
2. $\varphi_1 \wedge \cdots \wedge \varphi_n \Longrightarrow \varphi$,
3. $\neg(\varphi_1 \wedge \cdots \wedge \varphi_n)$,

where the formulae $\varphi_1, \ldots, \varphi_n, \varphi$ are all atomic.

Definition 4.2 (Horn Clause)**.** A *Horn clause* is a formula that consists of universal (first-order) quantifiers followed by a quantifier-free Horn formula.

Definition 4.3 (Strict Horn Clause)**.** A Horn clause is said to be *strict* iff no negation sign occurs in it, i.e. iff it comes from a Horn formula of the first or second kind.

Theories consisting of strict Horn clauses always have an *initial model*. This is known as the *initial model theorem* [70, 71].

```
input_formula(remove_sign, axiom, (
  ! [A, K] : ( ( knows(sign(A, K)) & knows(invs(K)) ) => knows(A) )
)).

input_formula(remove_hashpss, axiom, (
  ! [A] : ( knows(hashpss(A)) => knows(A) )
)).

input_formula(construct_msg, axiom, (
  ! [A, B] : ( ( knows(A) & knows(B) ) =>
    ( knows(conc(A, B)) & knows(sign(A, B)) ) )
)).

input_formula(deconstruct_msg, axiom, (
  ! [A, B] : ( knows(conc(A, B)) => ( knows(A) & knows(B) ) )
)).

input_formula(hashpss, axiom, (
  ! [A] : ( knows(A) => knows(hashpss(A)) )
)).

input_formula(sign_hom_1, axiom,  (
  ! [A, B, K] : ( knows(sign(conc(A, B), K)) =>
    ( knows(sign(A, K)) & knows(sign(B, K)) ) )
)).

input_formula(sign_hom_2, axiom, (
  ! [A, B, K] : ( ( knows(sign(A, K)) & knows(sign(B, K)) ) =>
    knows(sign(conc(A, B), K)) )
)).

input_formula(protocol_msg, axiom, (
  knows(conc(conc(a, b), sign(hashpss(conc(a, b)), k)))
)).

input_formula(public_key, axiom, (
  knows(invs(k))
)).

input_formula(attack, conjecture, (
  knows(sign(hashpss(conc(b, a)), k))
)).
```

Figure 4.1: TPTP encoding of the RSA-PSS protocol

```
*** Model found: ***
Size of types: 'a: 4
k: a2
b: a0
a: a3
conc: {(a0, {(a0, a0), (a1, a1), (a2, a2), (a3, a0)}),
 (a1, {(a0, a1), (a1, a1), (a2, a2), (a3, a1)}),
 (a2, {(a0, a2), (a1, a2), (a2, a2), (a3, a2)}),
 (a3, {(a0, a3), (a1, a1), (a2, a2), (a3, a0)})}
hashpss: {(a0, a1), (a1, a3), (a2, a2), (a3, a3)}
invs: {(a0, a2), (a1, a2), (a2, a0), (a3, a2)}
sign: {(a0, {(a0, a1), (a1, a1), (a2, a0), (a3, a1)}),
 (a1, {(a0, a1), (a1, a1), (a2, a2), (a3, a0)}),
 (a2, {(a0, a0), (a1, a0), (a2, a2), (a3, a1)}),
 (a3, {(a0, a1), (a1, a1), (a2, a3), (a3, a0)})}
knows: {(a0, True), (a1, True), (a2, False), (a3, True)}
```

Figure 4.2: Model showing security of RSA-PSS hashing

Theorem 4.4 (Initial Model Theorem)**.** *Let $\mathcal{T}$ be a theory consisting of strict universal Horn sentences. Then $\mathcal{T}$ has a model A with the property that for every model B of $\mathcal{T}$ there is a unique homomorphism from A to B. (Such a model A is called an* initial model *of $\mathcal{T}$. It is unique up to isomorphism.)*

The initial model satisfies the *no confusion* property [70]; its universe is indeed the (freely generated) term algebra of messages. Moreover, since $\mathsf{knows}\,\{\mathsf{hashpss}\,(b :: a)\}_k$ fails in the 4-element model found by Isabelle, existence of a homomorphism (in the sense of [71]) from the initial model to the 4-element model implies that $\mathsf{knows}\,\{\mathsf{hashpss}\,(b :: a)\}_k$ must also fail in the initial model. Thus we have shown that also with a more traditional algebraic interpretation of message concatenation, a Dolev-Yao attacker cannot generally forge RSA-PSS signatures (provided the actual implementations of hashing and signing do not have cryptographic weaknesses that would allow him to do so).

4.3 Probabilistic Programs

The mechanization of proofs for probabilistic programs is particularly challenging due to the verification of real-valued properties that are entailed by probability. Experience has shown that there are difficulties in automating real-number arithmetic in the context of other program features. The infinite domain of reals needed for quantitative analysis for example prevents the provision of counterexample search via state exploration [148].

In this section, we describe a framework for verification of probabilistic distributed systems (developed jointly with Annabelle McIver [108]) based on a generalization of Kleene algebra with tests [94]. First, it is shown how a model for probabilistic systems $\mathcal{L}S$ can be interpreted over a Kleene-style program algebra, so that explicit probabilistic reasoning is reduced significantly. Second, we propose a model of abstract propabilities $\mathcal{K}S$ that is susceptible to complete semantic exploration, yielding counterexamples even for the probabilistic model $\mathcal{L}S$.

The abstract model has been formalized in Isabelle/HOL, thereby making our implementation of finite model generation applicable to probabilistic programs.

4.3.1 The Probabilistic Model $\mathcal{L}S$

Probabilistic systems can show both quantifiable and unquantifiable non-deterministic behavior. The chance of winning an automated lottery is an example of the former, while the precise order of concurrent events can be an example of the latter. The transition-system style model that is now generally accepted for probabilistic systems [107] uses probability distributions to model quantifiable behavior, and sets of distributions to model unquantifiable non-determinism. This model is closely related to Markov decision processes [174].

Definition 4.5 (Discrete Probability Distribution)**.** Let S be a set. A function $f\colon S \to [0,1]$ is called a *discrete probability distribution (over S)* iff $\{s \in S \mid f(s) \neq 0\}$ is at most countable, and $\sum_{s\in S} f(s) = 1$.

In this section, we consider finite state spaces S only. We write $\overline{S}$ for the set of discrete probability distributions over S. A *point distribution* centered at point k is denoted by δ_k, i.e. $\delta_k(s) := 1$ if $s = k$, and $\delta_k(s) := 0$ otherwise. The $(p, 1-p)$-weighted average of two distributions d and d', i.e. $p \cdot d + (1-p) \cdot d'$ (for $0 \leq p \leq 1$), is written $d \,{}_p\!\oplus\, d'$. If K is a subset of S, and d is a discrete probability distribution over S, we write $d(K)$ for $\sum_{s\in K} d(s)$.

Thus, a probabilistic system with finite state space S is modelled by a function from initial states to subsets of distributions over final states, i.e. a function from S to $\mathcal{P}(\overline{S})$. For example, a program that simulates a fair coin is modelled by a function that maps an arbitrary state s to the evenly weighted average of the two point distributions representing heads and tails (but see below):

$$s \mapsto \{\delta_{\text{head}} \,{}_{1/2}\!\oplus\, \delta_{\text{tail}}\}.$$

The details are still a bit more complicated. We follow Morgan et al. [118] in taking a domain-theoretic approach, where the result sets of the semantic functions are restricted according to an underlying order on the state space. In addition, we distinguish special "miraculous" or infeasible behavior. Miracles, which will be associated with a special state $\top$ in the semantics, have various applications in program semantics [94, 116, 117]. Here, they will work particularly well with our simple Kleene-style program algebra. We write $S^\top$ for $S \cup \{\top\}$, where $\top$ is assumed to be not in S. The underlying order $\sqsubseteq$ is chosen so that $\top$ dominates all states in S, which are otherwise unrelated.

Definition 4.6 (Probabilistic Power Domain)**.** A *probabilistic power domain* is a pair $(\overline{S^\top}, \sqsubseteq_{\mathcal{D}})$, where $\sqsubseteq_{\mathcal{D}}$ is the order on $\overline{S^\top}$ that is induced from $\sqsubseteq$, i.e.

$$d \sqsubseteq_{\mathcal{D}} d' \quad \text{iff} \quad \forall s \in S.\, d(s) \geq d'(s).$$

Remark 4.7. Let d, $d' \in \overline{S^\top}$. Then $d \sqsubseteq_{\mathcal{D}} d'$ implies $d(\top) \leq d'(\top)$.

Proof. Using Def. 4.5, we have $d(\top) = 1 - d(S) \leq 1 - d'(S) = d'(\top)$. □

We now impose certain closure conditions (so-called *healthiness conditions*) on the sets of distributions that may be returned by probabilistic programs. These conditions, which are motivated

in detail in [107], reflect characteristics that are inherent in our model of non-determinism and probability. In particular, we require *up-closure* (the inclusion of all $\sqsubseteq_{\mathcal{D}}$-dominating distributions), *convex closure* (the inclusion of all convex combinations of distributions), and *Cauchy closure* (the inclusion of all limits of distributions, where distributions are viewed as vectors in $\mathbb{R}^{|S^\top|}$). This leads to the following definition, whose particular program model was first suggested by Carroll Morgan.

Definition 4.8 (Space of Probabilistic Programs)**.** The *space of probabilistic programs* is given by $(\mathcal{L}S, \sqsubseteq_{\mathcal{L}})$, where

$$\mathcal{L}S := \big\{ P \in S^\top \to \mathcal{P}(\overline{S^\top}) \quad | \quad \forall s \in S^\top.\, P(s) \text{ is up-, convex-, and Cauchy-closed} \\ \wedge\, P(s) \neq \emptyset \,\wedge\, P(\top) = \{\delta_\top\} \big\},$$

and the order between programs is defined by

$$P \sqsubseteq_{\mathcal{L}} P' \quad \text{iff} \quad \forall s \in S.\, P'(s) \subseteq P(s).$$

For a set of distributions $D \subseteq \overline{S^\top}$, we write $\lceil D \rceil$ for the smallest up-, convex-, and Cauchy-closed set of distributions containing D. We write $H_{\text{convex}}(D)$ for the convex hull (i.e. the smallest convex-closed superset) of D, and $H_{\text{up}}(D)$ for the up-closure (i.e. the smallest up-closed superset) of D.

Definition 4.9 (Up-, Convex-, Cauchy-Closed Hull)**.** For $D \subseteq \overline{S^\top}$, let

$$\lceil D \rceil := \bigcap \{ X \supseteq D \mid X \subseteq \overline{S^\top} \text{ is up-, convex-, and Cauchy-closed} \}.$$

Definition 4.10 (Convex Hull)**.** For $D \subseteq \overline{S^\top}$, let

$$H_{\text{convex}}(D) := \left\{ \sum_{i=1}^{k} \alpha_i d_i \,\middle|\, k \in \mathbb{N}, k \geq 1, d_i \in D, \alpha_i \in \mathbb{R}, \alpha_i \geq 0, \sum_{i=1}^{k} \alpha_i = 1 \right\}.$$

Definition 4.11 (Up-Closure)**.** For $D \subseteq \overline{S^\top}$, let

$$H_{\text{up}}(D) := \{ d' \in \overline{S^\top} \mid \exists d \in D.\, d \sqsubseteq_{\mathcal{D}} d' \}.$$

Remark 4.12. Let $D \subseteq \overline{S^\top}$. $\lceil D \rceil$ is the smallest up-, convex-, and Cauchy-closed superset of D. $H_{\text{convex}}(D)$ is the smallest convex-closed superset of D. $H_{\text{up}}(D)$ is the smallest up-closed superset of D. Hence $\lceil \cdot \rceil$, H_{convex}, and H_{up} are closure operators, i.e. extensive ($D \subseteq \lceil D \rceil$), monotonically increasing ($D \subseteq D' \subseteq \overline{S^\top}$ implies $\lceil D \rceil \subseteq \lceil D' \rceil$), and idempotent ($\lceil \lceil D \rceil \rceil = \lceil D \rceil$, likewise for H_{convex} and H_{up}).

Proof. We note that an arbitrary intersection of up-closed subsets of $\overline{S^\top}$ is up-closed again; likewise for an intersection of convex-closed and Cauchy-closed subsets, respectively. The various claims of Remark 4.12 now have standard proofs. □

Some basic lemmas about these closure operators will be useful later.

Lemma 4.13. *If $D \subseteq \overline{S^\top}$ is up-closed, then $H_{\text{convex}}(D)$ is up-closed.*

Proof. Let $d \sqsubseteq_{\mathcal{D}} d'$ with $d \in H_{\text{convex}}(D)$, i.e. $d = \sum_{i=1}^{k} \alpha_i d_i$ for some $k \in \mathbb{N}$, $k \geq 1$, $d_i \in D$, $\alpha_i \in \mathbb{R}$, $\alpha_i \geq 0$, and $\sum_{i=1}^{k} \alpha_i = 1$. For $1 \leq i \leq k$, define $d_i' \colon S^\top \to [0,1]$ by

$$d_i'(s) := \begin{cases} d_i(s) + \frac{d_i(s)}{d(s)}(d'(s) - d(s)), & \text{if } s \in S \text{ and } d(s) > 0; \\ 0, & \text{if } s \in S \text{ and } d(s) = 0; \\ 1 - \sum_{u \in S, d(u) > 0} \left(d_i(u) + \frac{d_i(u)}{d(u)}(d'(u) - d(u)) \right), & \text{otherwise (i.e. if } s = \top). \end{cases}$$

Clearly $d_i' \in \overline{S^\top}$, and $d_i \sqsubseteq_{\mathcal{D}} d_i'$. Hence up-closure implies $d_i' \in D$. Consequently, $d' = \sum_{i=1}^{k} \alpha_i d_i' \in H_{\text{convex}}(D)$. □

Lemma 4.14. *If $D \subseteq \overline{S^\top}$ is Cauchy-closed, then $H_{\text{convex}}(D)$ is Cauchy-closed.*

Proof. The proof is standard. Let $(d_n)_{n \in \mathbb{N}}$ be a convergent sequence in $H_{\text{convex}}(D)$, with $d := \lim_{n \to \infty} d_n$. Let (for all $n \in \mathbb{N}$) $d_n = \sum_{i=1}^{k_n} \alpha_n^i d_n^i$ for some $k_n \in \mathbb{N}$, $k_n \geq 1$, $d_n^i \in D$, $\alpha_n^i \in \mathbb{R}$, $\alpha_n^i \geq 0$, and $\sum_{i=1}^{k_n} \alpha_n^i = 1$. Let $k := |S^\top| + 1$. By Carathéodory's theorem [33, 153], we may assume $k_n \leq k$ (and hence, without loss of generality, $k_n = k$) for all $n \in \mathbb{N}$. For each $1 \leq i \leq k$, the sequences $(\alpha_n^i)_{n \in \mathbb{N}}$ and $(d_n^i)_{n \in \mathbb{N}}$ have a convergent subsequence by the Bolzano-Weierstraß theorem; call the limits of these subsequences α^i and d^i, respectively. Cauchy closure implies $d^i \in D$ (for $1 \leq i \leq k$). Hence $d = \sum_{i=1}^{k} \alpha^i d^i \in H_{\text{convex}}(D)$. □

Combining these lemmas and Remark 4.12, we obtain the following characterization of $\lceil \cdot \rceil$.

Lemma 4.15. *If $D \subseteq \overline{S^\top}$ is Cauchy- and up-closed, then $\lceil D \rceil = H_{\text{convex}}(D)$.*

Proof. $H_{\text{convex}}(D)$ is an up-closed (Lemma 4.13), convex-closed (Remark 4.12), and Cauchy-closed (Lemma 4.14) superset of D. Hence we have $\lceil D \rceil \subseteq H_{\text{convex}}(D)$ by Remark 4.12. The other inclusion is immediate. □

The up-closure of a Cauchy-closed set of probability distributions is Cauchy-closed.

Lemma 4.16. *If $D \subseteq \overline{S^\top}$ is Cauchy-closed, then $H_{\text{up}}(D)$ is Cauchy-closed.*

Proof. Let $(d_n')_{n \in \mathbb{N}}$ be a convergent sequence in $H_{\text{up}}(D)$, with $d' := \lim_{n \to \infty} d_n'$. Then there exists a sequence $(d_n)_{n \in \mathbb{N}}$ with $d_n \in D$ and $d_n \sqsubseteq_{\mathcal{D}} d_n'$ for all $n \in \mathbb{N}$. By the Bolzano-Weierstraß theorem, $(d_n)_{n \in \mathbb{N}}$ has a convergent subsequence; call the limit of this subsequence d. Cauchy closure implies $d \in D$. Clearly $d \sqsubseteq_{\mathcal{D}} d'$, hence $d' \in H_{\text{up}}(D)$. □

Together with Lemma 4.15, the previous lemma implies $\lceil D \rceil = H_{\text{convex}}(H_{\text{up}}(D))$, provided $D \subseteq \overline{S^\top}$ is Cauchy-closed.

Lemma 4.17. *If $D \subseteq \overline{S^\top}$ is Cauchy-closed, then $\lceil D \rceil = H_{\text{convex}}(H_{\text{up}}(D))$.*

Proof. $H_{\text{up}}(D)$ is a Cauchy-closed (Lemma 4.16) and up-closed (Remark 4.12) superset of D. Hence $\lceil D \rceil \subseteq \lceil H_{\text{up}}(D) \rceil = H_{\text{convex}}(H_{\text{up}}(D))$ by Lemma 4.15 and Remark 4.12. The other inclusion is immediate. □

We write $H_{\text{Cauchy}}(D)$ for the Cauchy closure of D. Again we need some basic lemmas relating H_{Cauchy} to the other closure operators. First, the Cauchy closure of an up-closed set is up-closed.

Lemma 4.18. *If $D \subseteq \overline{S^\top}$ is up-closed, then $H_{\text{Cauchy}}(D)$ is up-closed.*

Proof. Let $d \sqsubseteq_{\mathcal{D}} d'$ with $d \in H_{\text{Cauchy}}(D)$, i.e. $d = \lim_{n\to\infty} d_n$ with $d_n \in D$ (for all $n \in \mathbb{N}$). For $n \in \mathbb{N}$, define $d'_n \colon S^\top \to [0,1]$ by

$$d'_n(s) := \begin{cases} \min\{d_n(s), d'(s)\}, & \text{if } s \in S; \\ 1 - \sum_{u\in S} \min\{d_n(u), d'(u)\}, & \text{otherwise (i.e. if } s = \top). \end{cases}$$

Clearly $d'_n \in \overline{S^\top}$, and $d_n \sqsubseteq_{\mathcal{D}} d'_n$. Hence up-closure implies $d'_n \in D$. Consequently, $d' = \lim_{n\to\infty} d'_n \in H_{\text{Cauchy}}(D)$. □

Second, the Cauchy closure of a convex-closed set is convex-closed.

Lemma 4.19. *If $D \subseteq \overline{S^\top}$ is convex-closed, then $H_{\text{Cauchy}}(D)$ is convex-closed.*

Proof. The proof is standard. Let $k \in \mathbb{N}$, $k \geq 1$, $d_i \in H_{\text{Cauchy}}(D)$ (for $1 \leq i \leq k$), $\alpha_i \in \mathbb{R}$, $\alpha_i \geq 0$, and $\sum_{i=1}^k \alpha_i = 1$. Then $d_i = \lim_{n\to\infty} d_{i,n}$ for some $d_{i,n} \in D$ (for all $n \in \mathbb{N}$). Convex closure implies $\sum_{i=1}^k \alpha_i d_{i,n} \in D$. Hence $\sum_{i=1}^k \alpha_i d_i = \lim_{n\to\infty}(\sum_{i=1}^k \alpha_i d_{i,n}) \in H_{\text{Cauchy}}(D)$. □

The previous two lemmas lead to the following alternative characterization of $\lceil\cdot\rceil$.

Lemma 4.20. *If $D \subseteq \overline{S^\top}$ is convex- and up-closed, then $\lceil D\rceil = H_{\text{Cauchy}}(D)$.*

Proof. $H_{\text{Cauchy}}(D)$ is an up-closed (Lemma 4.18), convex-closed (Lemma 4.19), and Cauchy-closed superset of D. Hence we have $\lceil D\rceil \subseteq H_{\text{Cauchy}}(D)$ by Remark 4.12. The other inclusion is immediate. □

Various mathematical operators on the space of probabilistic programs are defined next. These operators will be needed to interpret Kleene algebra expressions; the definitions are taken from [107].

Definition 4.21 (Operators on $\mathcal{L}S$)**.** For arbitrary states $s \in S^\top$, programs P, $P' \in \mathcal{L}S$, and $0 \leq p \leq 1$, we define

Identity	$\mathrm{Id}(s)$	:=	$\lceil\{\delta_s\}\rceil$,
Top	$\top(s)$	:=	$\{\delta_\top\}$,
Composition	$(P;P')(s)$	:=	$\{\sum_{u\in S^\top} d(u)\cdot d'_u \mid d \in P(s), \forall u \in S^\top.\, d'_u \in P'(u)\}$,
Probability	$(P\,{}_p\!\oplus P')(s)$	:=	$\{d\,{}_p\!\oplus d' \mid d \in P(s), d' \in P'(s)\}$,
Non-determinism	$(P \sqcap P')(s)$	:=	$\lceil P(s) \cup P'(s)\rceil$,
Iteration	P^*	:=	$\nu X.\,(P;X) \sqcap \mathrm{Id}$,

where $\nu X.\, f(X)$ denotes the greatest fixed point of the function $f \colon \mathcal{L}S \to \mathcal{L}S$ (with respect to $\sqsubseteq_{\mathcal{L}}$).

Remark 4.22. Let P, $P' \in \mathcal{LS}$, and $0 \leq p \leq 1$. Then $\mathrm{Id} \in \mathcal{LS}$, $\top \in \mathcal{LS}$, $(P;P') \in \mathcal{LS}$, $P \,{}_p\!\oplus P' \in \mathcal{LS}$, $P \sqcap P' \in \mathcal{LS}$, and $P^* \in \mathcal{LS}$.

Proof. We note that for arbitrary $d \in \overline{S^\top}$, $\delta_\top \sqsubseteq_{\mathcal{D}} d$ iff $d = \delta_\top$. Hence $\lceil\{\delta_\top\}\rceil = \{\delta_\top\}$.

Let $s \in S^\top$. For $\mathrm{Id}(s)$ and $\top(s)$, all relevant properties (i.e. up-closure, convex closure, Cauchy closure, non-emptiness, and $\mathrm{Id}(\top) = \top(\top) = \{\delta_\top\}$) now follow immediately from Def. 4.21.

Up-closure of $(P;P')(s)$ follows from up-closure of $P'(u)$ for all $u \in S^\top$. To prove convex closure, we note that for arbitrary probability distributions d, d'_u, e, $e'_u \in \overline{S^\top}$ (where u ranges over $S^\top$) and $0 \leq q \leq 1$, we have

$$\left(\sum_{u\in S^\top} d(u)\cdot d'_u\right) {}_q\!\oplus \left(\sum_{u\in S^\top} e(u)\cdot e'_u\right) = \sum_{u\in S^\top} (d \,{}_q\!\oplus e)(u)\cdot (d'_u \,{}_{\frac{q\cdot d(u)}{(d\,{}_q\!\oplus e)(u)}}\!\oplus e'_u),$$

assuming (without loss of generality) $(d \,{}_q\!\oplus e)(u) > 0$ for all $u \in S^\top$. Hence convex closure of $(P;P')(s)$ follows from convex closure of $P(s)$ and $P'(u)$ for all $u \in S^\top$. To prove Cauchy closure, let $(d_n)_{n\in\mathbb{N}}$ and $(d'_{u,n})_{n\in\mathbb{N}}$ (where u ranges over $S^\top$) be sequences of probability distributions such that $\lim_{n\to\infty}\left(\sum_{u\in S^\top} d_n(u)\cdot d'_{u,n}\right)$ exists. We note that $(d_n)_{n\in\mathbb{N}}$ and (for arbitrary $u \in S^\top$) $(d'_{u,n})_{n\in\mathbb{N}}$ are bounded sequences in $\mathbb{R}^{|S^\top|}$, which must therefore have convergent subsequences by the Bolzano-Weierstraß theorem. Cauchy closure of $(P;P')(s)$ now follows from Cauchy closure of $P(s)$ and $P'(u)$ for all $u \in S^\top$. Non-emptiness of $(P;P')(s)$ and $(P;P')(\top) = \{\delta_\top\}$ are immediate.

Up-closure of $(P \,{}_p\!\oplus P')(s)$ follows from up-closure of $P(s)$ and $P'(s)$. Since (for arbitrary probability distributions d, d', e, $e' \in \overline{S^\top}$, and $0 \leq q \leq 1$) we have $(d \,{}_p\!\oplus d') \,{}_q\!\oplus (e \,{}_p\!\oplus e') = (d \,{}_q\!\oplus e) \,{}_p\!\oplus (d' \,{}_q\!\oplus e')$, convex closure of $(P \,{}_p\!\oplus P')(s)$ follows from convex closure of $P(s)$ and $P'(s)$. Cauchy closure of $(P \,{}_p\!\oplus P')(s)$ follows from Cauchy closure of $P(s)$ and $P'(s)$, with an argument similar to the one for Cauchy closure of $(P;P')(s)$ above. Non-emptiness of $(P \,{}_p\!\oplus P')(s)$ and $(P \,{}_p\!\oplus P')(\top) = \{\delta_\top\}$ are immediate.

For $(P \sqcap P')(s)$, all relevant properties are immediate again from Def. 4.21.

Finally, we note that $X \mapsto (P;X) \sqcap \mathrm{Id}$ is monotonic wrt. to $\sqsubseteq_{\mathcal{L}}$, and $(\mathcal{LS}, \sqsubseteq_{\mathcal{L}})$ is a complete lattice: $\sqsubseteq_{\mathcal{L}}$ is clearly a partial order (i.e. reflexive, antisymmetric, transitive), $(\bigwedge L)(s) := \left\lceil \bigcup_{P\in L} P(s)\right\rceil$ (for $s \in S^\top$) defines the greatest lower bound (meet) of $L \subseteq \mathcal{LS}$, and $(\bigvee L)(s) := \bigcap_{P\in L} P(s)$ defines the least upper bound (join) of L. Therefore P^* exists (and is in $\mathcal{LS}$) by the Knaster-Tarski theorem [157]. □

Iteration is the most intricate of these operations. Operationally P^* represents the program that can iterate P an arbitrary (finite) number of times. This generates all results of finite iterations of $P \sqcap \mathrm{Id}$. In addition, imposition of Cauchy closure implies that also all limits of distributions are contained. Note that P^* is in general not the same as $\mathrm{Id} \sqcap P \sqcap (P;P) \sqcap \ldots$. The latter program requires the number of iterations to be chosen at the start, while P^* allows the choice between P and Id to be made after each iteration. For a concrete counterexample, consider again the program that simulates a fair coin, now (more precisely than before) given by $s \mapsto \lceil\{\delta_{\mathrm{head}\ 1/2\oplus\delta_{\mathrm{tail}}}\}\rceil$. If we take P to be this program, we have $P = P;P = P;P;P = \ldots$, since the probability to be in state "head" is exactly $1/2$ after each iteration (and likewise for state "tail"). The program P^* on the other hand allows us to iterate P until we are in a

desired state; this means that we can reach both states "head" and "tail" with probability arbitrarily close to 1. Cauchy closure then implies that also the limit distributions δ_{head} and δ_{tail} are contained in P^*. The next lemma states this operational characterization of P^* more formally. We write P^n for the n-fold iteration of P, i.e. $P^0 := \mathrm{Id}$, and (for arbitrary $n \in \mathbb{N}$) $P^{n+1} := P; P^n$.

Lemma 4.23. *Let $P \in \mathcal{L}S$. Then for all $s \in S^\top$, $P^*(s) = \lceil \bigcup_{n\in\mathbb{N}} (P \sqcap \mathrm{Id})^n(s) \rceil$.*

Proof. The lemma follows from the definition of P^* as the greatest fixed point of the function $f\colon \mathcal{L}S \to \mathcal{L}S$, given by $f(X) := (P; X) \sqcap \mathrm{Id}$.

Let $(X_n)_{n\in\mathbb{N}}$ be a descending (wrt. $\sqsubseteq_{\mathcal{L}}$, i.e. $X_n(s) \subseteq X_{n+1}(s)$ for all $s \in S$, $n \in \mathbb{N}$) sequence in $\mathcal{L}S$. We note that f, as a composition of program composition and non-determinism, satisfies $f(\bigwedge_{n\in\mathbb{N}} X_n) = \bigwedge_{n\in\mathbb{N}} f(X_n)$: first, we show that for all $s \in S^\top$, $(P; \bigwedge_{n\in\mathbb{N}} X_n)(s) = \{\sum_{u\in S^\top} d(u) \cdot d'_u \mid d \in P(s), \forall u \in S^\top.\, d'_u \in \lceil \bigcup_{n\in\mathbb{N}} X_n(u) \rceil\} = \lceil \bigcup_{n\in\mathbb{N}} \{\sum_{u\in S^\top} d(u) \cdot d'_u \mid d \in P(s), \forall u \in S^\top.\, d'_u \in X_n(u)\} \rceil = (\bigwedge_{n\in\mathbb{N}} (P; X_n))(s)$. Here the "$\subseteq$" inclusion follows from the fact that $\bigcup_{n\in\mathbb{N}} X_n(u)$ (for $u \in S^\top$) is both up-closed (since a union of up-closed sets is up-closed) and convex-closed (since the union of an ascending, wrt. $\subseteq$, sequence of convex-closed sets is convex-closed). Hence $\lceil \bigcup_{n\in\mathbb{N}} X_n(u) \rceil = H_{\text{Cauchy}}(\bigcup_{n\in\mathbb{N}} X_n(u))$ by Lemma 4.20. Therefore $d'_u \in \lceil \bigcup_{n\in\mathbb{N}} X_n(u) \rceil$ can be written as $d'_u = \lim_{k\to\infty} d'_{u,k}$ for some $d'_{u,k} \in \bigcup_{n\in\mathbb{N}} X_n(u)$. Now for any $d \in P(s)$, $\sum_{u\in S^\top} d(u) \cdot d'_u = \sum_{u\in S^\top} (d(u) \cdot \lim_{k\to\infty} d'_{u,k}) = \lim_{k\to\infty} (\sum_{u\in S^\top} d(u) \cdot d'_{u,k})$, and for any $k \in \mathbb{N}$ there exists $n_k \in \mathbb{N}$ such that $d'_{u,k} \in X_{n_k}(u)$ for all $u \in S^\top$ (because $X_n(u) \subseteq X_{n+1}(u)$ for all $n \in \mathbb{N}$). This proves $(P; \bigwedge_{n\in\mathbb{N}} X_n)(s) \subseteq (\bigwedge_{n\in\mathbb{N}} (P; X_n))(s)$. The other inclusion is immediate from Remark 4.12. Second, for all $L \subseteq \mathcal{L}S$ and $s \in S^\top$, $((\bigwedge L) \sqcap \mathrm{Id})(s) = \lceil \lceil \bigcup_{X\in L} X(s) \rceil \cup \mathrm{Id}(s) \rceil = \lceil \bigcup_{X\in L} \lceil X(s) \cup \mathrm{Id}(s) \rceil \rceil = (\bigwedge_{X\in L} (X \sqcap \mathrm{Id}))(s)$ follows also from Remark 4.12.

Hence by the Knaster-Tarski theorem [157], $P^* = \bigwedge_{n\in\mathbb{N}} f^n(\bigvee \mathcal{L}S)$, i.e. for all $s \in S^\top$, $P^*(s) = \lceil \bigcup_{n\in\mathbb{N}} (f^n(\top))(s) \rceil$.

Finally $f^{n+1}(\top) = (P \sqcap \mathrm{Id})^n$ for all $n \in \mathbb{N}$ by induction: it is easy to show $f^1(\top) = \mathrm{Id} = (P \sqcap \mathrm{Id})^0$. Next, using the induction hypothesis and monotonicity of f, we have $f^{n+2}(\top) = f^{n+2}(\top) \sqcap f^{n+1}(\top) = (P; f^{n+1}(\top)) \sqcap \mathrm{Id} \sqcap f^{n+1}(\top) = (P \sqcap \mathrm{Id}); (P \sqcap \mathrm{Id})^n = (P \sqcap \mathrm{Id})^{n+1}$ (see the proof of Theorem 4.26 below for the detailed calculations).

Therefore $\lceil \bigcup_{n\in\mathbb{N}} (f^n(\top))(s) \rceil = \lceil \bigcup_{n\in\mathbb{N}} (P \sqcap \mathrm{Id})^n(s) \rceil$ for all $s \in S^\top$. □

A *Kleene algebra* [93] is an algebraic structure that generalizes the operations known from regular expressions. It consists of a binary sequential composition operator (written as multiplication, $\cdot$), a binary choice operator (written as addition, $+$), and a unary iteration operator (written as a postfix star, $\cdot^*$). Terms are ordered by $\leq$, which is defined via binary choice.

Definition 4.24 (Probabilistic Kleene Algebra)**.** A *probabilistic Kleene algebra (pKA)* is a set A (containing elements 0 and 1) together with two binary operations $\cdot\colon A \times A \to A$ and $+\colon A \times A \to A$ and a unary operation $\cdot^*\colon A \to A$, as well as a binary relation $\leq$ over A, such that the following axioms are satisfied:

1. $0 + a = a$,
2. $a + b = b + a$,

3. $a + a = a$,
4. $a + (b + c) = (a + b) + c$,
5. $a(bc) = (ab)c$,
6. $0a = a0 = 0$,
7. $1a = a1 = a$,
8. $ab + ac \leq a(b + c)$,
9. $(a + b)c = ac + bc$,
10. $a \leq b$ iff $a + b = b$,
11. $a^* = 1 + aa^*$,
12. $a(b + 1) \leq a \Longrightarrow ab^* = a$,
13. $ab \leq b \Longrightarrow a^*b = b$.

Note that Axiom (8) is weaker than the corresponding rule in standard Kleene algebra [42]; this is because of the well-documented ([107, 147], also compare the above discussion on P^* vs. $\mathrm{Id} \sqcap P \sqcap (P; P) \sqcap \ldots$) interaction of probability and non-determinism.

Probabilistic Kleene algebra expressions are built from variables (x, y, z, ...) and constants (0 and 1), using the (unary or binary) operations $\cdot$, $+$, and $\cdot^*$. We can now define an interpretation of *pKA* expressions in the space of probabilistic programs.

Definition 4.25 (Semantic Mapping of *pKA* Expressions to $\mathcal{L}S$)**.** The semantic mapping $[\![\cdot]\!]_\rho$ from *pKA* expressions to $\mathcal{L}S$ is parameterized by a mapping ρ from *pKA* variables to probabilistic programs in $\mathcal{L}S$, and defined as follows:

1. If x is a variable, then $[\![x]\!]_\rho := \rho(x)$.
2. $[\![0]\!]_\rho := \top$, $[\![1]\!]_\rho := \mathrm{Id}$.
3. $[\![ab]\!]_\rho := [\![a]\!]_\rho ; [\![b]\!]_\rho$.
4. $[\![a + b]\!]_\rho := [\![a]\!]_\rho \sqcap [\![b]\!]_\rho$.
5. $[\![a^*]\!]_\rho := [\![a]\!]_\rho{}^*$.

On the right-hand side of the defining equations, Def. 4.25 refers to the operators on $\mathcal{L}S$ that were introduced in Def. 4.21. The following theorem shows that the semantic mapping is a valid interpretation for the *pKA* axioms given in Def. 4.24.

Theorem 4.26. *$\mathcal{L}S$, with 0, 1, $\cdot$, $+$ and $\cdot^*$ as given in Def. 4.25 (and with the additional definition $P \leq P'$ iff $P' \sqsubseteq_{\mathcal{L}} P$), is a probabilistic Kleene algebra.*

Proof. Using Def. 4.25 (and a number of simple lemmas), one verifies that Axioms (1)–(13) of *pKA* are satisfied. Let a, b, c be probabilistic Kleene algebra expressions, let ρ be a mapping from *pKA* variables to probabilistic programs in $\mathcal{L}S$, and let $s \in S^\top$.

1. $0 + a = a$: We note that every non-empty, up-closed subset of $\overline{S^\top}$ contains $\delta_\top$. Hence $[\![0 + a]\!]_\rho(s) = \lceil \{\delta_\top\} \cup [\![a]\!]_\rho(s) \rceil = \lceil [\![a]\!]_\rho(s) \rceil = [\![a]\!]_\rho(s)$.

2. $a + b = b + a$: $[\![a + b]\!]_\rho(s) = \lceil [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s) \rceil = \lceil [\![b]\!]_\rho(s) \cup [\![a]\!]_\rho(s) \rceil = [\![b + a]\!]_\rho(s)$.

3. $a + a = a$: $[\![a + a]\!]_\rho(s) = \lceil [\![a]\!]_\rho(s) \cup [\![a]\!]_\rho(s) \rceil = \lceil [\![a]\!]_\rho(s) \rceil = [\![a]\!]_\rho(s)$.

4. $a + (b + c) = (a + b) + c$: It is easy to show that for arbitrary D_1, $D_2 \subseteq \overline{S^\top}$, $\lceil D_1 \cup D_2 \rceil = \lceil \lceil D_1 \rceil \cup \lceil D_2 \rceil \rceil$. Hence $[\![a + (b + c)]\!]_\rho(s) = \lceil [\![a]\!]_\rho(s) \cup \lceil [\![b]\!]_\rho(s) \cup [\![c]\!]_\rho(s) \rceil \rceil = \lceil [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s) \cup [\![c]\!]_\rho(s) \rceil = \lceil \lceil [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s) \rceil \cup [\![c]\!]_\rho(s) \rceil = [\![(a + b) + c]\!]_\rho(s)$.

5. $a(bc) = (ab)c$: $[\![a(bc)]\!]_\rho(s) = \{\sum_{u \in S^\top} d(u) \cdot (\sum_{v \in S^\top} e_u(v) \cdot f_{u,v}) \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . e_u \in [\![b]\!]_\rho(u), \forall u, v \in S^\top . f_{u,v} \in [\![c]\!]_\rho(v)\}$, and $[\![(ab)c]\!]_\rho(s) = \{\sum_{v \in S^\top} (\sum_{u \in S^\top} d(u) \cdot e_u)(v) \cdot f_v \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . e_u \in [\![b]\!]_\rho(u), \forall v \in S^\top . f_v \in [\![c]\!]_\rho(v)\}$. Because $\sum_{v \in S^\top} (\sum_{u \in S^\top} d(u) \cdot e_u)(v) \cdot f_v = \sum_{u \in S^\top} d(u) \cdot (\sum_{v \in S^\top} e_u(v) \cdot f_v)$ (for $d \in [\![a]\!]_\rho(s)$, $e_u \in [\![b]\!]_\rho(u)$ for all $u \in S^\top$, $f_v \in [\![c]\!]_\rho(v)$ for all $v \in S^\top$), the inclusion $[\![a(bc)]\!]_\rho(s) \supseteq [\![(ab)c]\!]_\rho(s)$ is now immediate, by setting $f_{u,v} := f_v$ for all $u \in S^\top$. To show the other inclusion, we note that (for $d \in [\![a]\!]_\rho(s)$, $e_u \in [\![b]\!]_\rho(u)$ for all $u \in S^\top$, and $f_{u,v} \in [\![c]\!]_\rho(v)$ for all u, $v \in S^\top$) $\sum_{u \in S^\top} d(u) \cdot (\sum_{v \in S^\top} e_u(v) \cdot f_{u,v}) = \sum_{v \in S^\top} (\sum_{u \in S^\top} d(u) \cdot e_u)(v) \cdot f_v$, where
$$f_v := \sum_{u \in S^\top} \frac{d(u) \cdot e_u(v)}{\sum_{u' \in S^\top} d(u') \cdot e_{u'}(v)} \cdot f_{u,v}$$
(assuming, without loss of generality, $\sum_{u \in S^\top} d(u) \cdot e_u(v) > 0$) is in $[\![c]\!]_\rho(v)$ (for all $v \in S^\top$) because the latter is convex-closed.

6. $0a = a0 = 0$: $[\![0a]\!]_\rho(s) = \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![0]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![a]\!]_\rho(u)\} = \{d'_\top \mid d'_\top \in [\![a]\!]_\rho(\top)\} = \{\delta_\top\} = [\![0]\!]_\rho(s)$. Likewise, $[\![a0]\!]_\rho(s) = \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![0]\!]_\rho(u)\} = \{\sum_{u \in S^\top} d(u) \cdot \delta_\top \mid d \in [\![a]\!]_\rho(s)\} = \{\delta_\top\} = [\![0]\!]_\rho(s)$.

7. $1a = a1 = a$: We note that $\lceil \{\delta_s\} \rceil = \{\delta_{s\,p} \oplus \delta_\top \mid 0 \leq p \leq 1\}$. Hence $[\![1a]\!]_\rho(s) = \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![1]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![a]\!]_\rho(u)\} = \{d'_{s\,p} \oplus \delta_\top \mid d'_s \in [\![a]\!]_\rho(s), 0 \leq p \leq 1\} = [\![a]\!]_\rho(s)$. Likewise, $[\![a1]\!]_\rho(s) = \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![1]\!]_\rho(u)\} = \{\sum_{u \in S^\top} d(u) \cdot (\delta_{u\,p} \oplus \delta_\top) \mid d \in [\![a]\!]_\rho(s), 0 \leq p \leq 1\} = \{d\,_p \oplus \delta_\top \mid d \in [\![a]\!]_\rho(s), 0 \leq p \leq 1\} = [\![a]\!]_\rho(s)$.

8. $ab + ac \leq a(b + c)$: $[\![ab + ac]\!]_\rho(s) = \lceil \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![b]\!]_\rho(u)\} \cup \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![c]\!]_\rho(u)\} \rceil$, and $[\![a(b + c)]\!]_\rho(s) = \{\sum_{u \in S^\top} d(u) \cdot d'_u \mid d \in [\![a]\!]_\rho(s), \forall u \in S^\top . d'_u \in \lceil [\![b]\!]_\rho(u)\} \cup [\![c]\!]_\rho(u) \rceil\}$. The inclusion $[\![ab + ac]\!]_\rho(s) \subseteq [\![a(b + c)]\!]_\rho(s)$ is now immediate from Remark 4.12.

9. $(a + b)c = ac + bc$: Let $d \in [\![(a + b)c]\!]_\rho(s)$, i.e. $d = \sum_{u \in S^\top} d'(u) \cdot d'_u$ for some $d' \in \lceil [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s) \rceil$, $d'_u \in [\![c]\!]_\rho(u)$ (for all $u \in S^\top$). Using Lemma 4.15, we can show $\lceil [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s) \rceil = H_{\text{convex}}([\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s))$. If $d' \in [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s)$, then $d \in [\![ac + bc]\!]_\rho(s) = \lceil \{\sum_{u \in S^\top} d'(u) \cdot d'_u \mid d' \in [\![a]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![c]\!]_\rho(u)\} \cup \{\sum_{u \in S^\top} d'(u) \cdot d'_u \mid d' \in [\![b]\!]_\rho(s), \forall u \in S^\top . d'_u \in [\![c]\!]_\rho(u)\} \rceil$ is immediate. Next, if $d' = \sum_{i=1}^k \alpha_i d_i$ for some $k \in \mathbb{N}$, $k \geq 1$, $d_i \in [\![a]\!]_\rho(s) \cup [\![b]\!]_\rho(s)$, $\alpha_i \in \mathbb{R}$, $\alpha_i \geq 0$, and $\sum_{i=1}^k \alpha_i = 1$, then

$d = \sum_{u\in S^\top}(\sum_{i=1}^k \alpha_i d_i(u))\cdot d'_u = \sum_{i=1}^k \alpha_i(\sum_{u\in S^\top} d_i(u)\cdot d'_u) \in H_{\text{convex}}(\llbracket ac\rrbracket_\rho(s)\cup\llbracket bc\rrbracket_\rho(s))$. This proves $\llbracket (a+b)c\rrbracket_\rho(s) \subseteq \llbracket ac+bc\rrbracket_\rho(s)$. The other inclusion is immediate from Remark 4.12.

10. $a \leq b$ iff $a+b=b$: By definition of $\sqsubseteq_{\mathcal{L}}$, $\llbracket a\rrbracket_\rho \leq \llbracket b\rrbracket_\rho$ iff $\llbracket a\rrbracket_\rho(s) \subseteq \llbracket b\rrbracket_\rho(s)$ for all $s \in S$. Also $\llbracket a\rrbracket_\rho(\top) = \llbracket b\rrbracket_\rho(\top) = \{\delta_\top\}$ for all $\llbracket a\rrbracket_\rho, \llbracket b\rrbracket_\rho \in \mathcal{L}S$. On the other hand, $\llbracket a+b\rrbracket_\rho = \llbracket b\rrbracket_\rho$ iff $\lceil \llbracket a\rrbracket_\rho(s) \cup \llbracket b\rrbracket_\rho(s)\rceil = \llbracket b\rrbracket_\rho(s)$ for all $s \in S^\top$. Equivalence is now immediate.

11. $a^* = 1 + aa^*$: Since $\llbracket a\rrbracket_\rho{}^*$ is a fixed point of the function $X \mapsto (\llbracket a\rrbracket_\rho; X) \sqcap \mathrm{Id}$ (with $X \in \mathcal{L}S$), we have $\llbracket a^*\rrbracket_\rho = \llbracket a\rrbracket_\rho{}^* = (\llbracket a\rrbracket_\rho; \llbracket a\rrbracket_\rho{}^*) \sqcap \mathrm{Id} = \llbracket 1+aa^*\rrbracket_\rho$.

12. $a(b+1) \leq a \implies ab^* = a$: Since $\llbracket 1\rrbracket_\rho \leq \llbracket b^*\rrbracket_\rho$ (by Lemma 4.23), $\llbracket a\rrbracket_\rho = \llbracket a1\rrbracket_\rho \leq \llbracket ab^*\rrbracket_\rho$ follows from monotonicity of composition. It remains to show $\llbracket ab^*\rrbracket_\rho \leq \llbracket a\rrbracket_\rho$, i.e. $\llbracket ab^*\rrbracket_\rho(s) \subseteq \llbracket a\rrbracket_\rho(s)$ for all $s \in S$. Let $d \in \llbracket ab^*\rrbracket_\rho(s)$, i.e. $d = \sum_{u\in S^\top} d'(u)\cdot d'_u$ for some $d' \in \llbracket a\rrbracket_\rho(s)$, $d'_u \in \llbracket b^*\rrbracket_\rho(u)$ for all $u \in S^\top$. Using similar arguments as in the proof of Lemma 4.23, we have $\llbracket b^*\rrbracket_\rho(u) = H_{\text{Cauchy}}(\bigcup_{n\in\mathbb{N}} \llbracket (b+1)^n\rrbracket_\rho(u))$. Hence (for all $u \in S^\top$) $d'_u = \lim_{k\to\infty} d'_{u,k}$ with $d'_{u,k} \in \bigcup_{n\in\mathbb{N}} \llbracket (b+1)^n\rrbracket_\rho(u)$ for all $k \in \mathbb{N}$. Since, by induction, $\llbracket (b+1)^n\rrbracket_\rho \leq \llbracket (b+1)^{n+1}\rrbracket_\rho$ for all $n \in \mathbb{N}$, there exists (for any $k \in \mathbb{N}$) $n_k \in \mathbb{N}$ such that $d'_{u,k} \in \llbracket (b+1)^{n_k}\rrbracket_\rho(u)$ for all $u \in S^\top$. Now $d = \sum_{u\in S^\top}(d'(u)\cdot \lim_{k\to\infty} d'_{u,k}) = \lim_{k\to\infty}(\sum_{u\in S^\top} d'(u)\cdot d'_{u,k}) \in \llbracket a\rrbracket_\rho(s)$ because (by induction) $\llbracket a(b+1)^n\rrbracket_\rho \leq \llbracket a\rrbracket_\rho$ for all $n \in \mathbb{N}$, and because $\llbracket a\rrbracket_\rho(s)$ is Cauchy-closed.

13. $ab \leq b \implies a^*b = b$: Since $\llbracket 1\rrbracket_\rho \leq \llbracket a^*\rrbracket_\rho$ (by Lemma 4.23), $\llbracket b\rrbracket_\rho = \llbracket 1b\rrbracket_\rho \leq \llbracket a^*b\rrbracket_\rho$ follows from monotonicity of composition. It remains to show $\llbracket a^*b\rrbracket_\rho \leq \llbracket b\rrbracket_\rho$, i.e. $\llbracket a^*b\rrbracket_\rho(s) \subseteq \llbracket b\rrbracket_\rho(s)$ for all $s \in S$. Let $d \in \llbracket a^*b\rrbracket_\rho(s)$, i.e. $d = \sum_{u\in S^\top} d'(u)\cdot d'_u$ for some $d' \in \llbracket a^*\rrbracket_\rho(s)$, $d'_u \in \llbracket b\rrbracket_\rho(u)$ for all $u \in S^\top$. Using similar arguments as in the proof of Lemma 4.23, we have $\llbracket a^*\rrbracket_\rho(s) = H_{\text{Cauchy}}(\bigcup_{n\in\mathbb{N}} \llbracket (a+1)^n\rrbracket_\rho(s))$. Hence $d' = \lim_{k\to\infty} d'_k$ with $d'_k \in \bigcup_{n\in\mathbb{N}} \llbracket (a+1)^n\rrbracket_\rho(s)$ for all $k \in \mathbb{N}$. Now $d = \sum_{u\in S^\top}(\lim_{k\to\infty} d'_k)(u)\cdot d'_u = \lim_{k\to\infty}(\sum_{u\in S^\top} d'_k(u)\cdot d'_u) \in \llbracket b\rrbracket_\rho(s)$ because (by induction) $\llbracket (a+1)^n b\rrbracket_\rho \leq \llbracket b\rrbracket_\rho$ for all $n \in \mathbb{N}$ (where the case $n=1$, i.e. $\llbracket (a+1)b\rrbracket_\rho \leq \llbracket b\rrbracket_\rho$, follows from the premise $\llbracket ab\rrbracket_\rho \leq \llbracket b\rrbracket_\rho$), and because $\llbracket b\rrbracket_\rho(s)$ is Cauchy-closed.

□

The following corollary is an immediate consequence of Theorem 4.26.

Corollary 4.27. *If $a \leq b$ is a theorem of* pKA *(as given in Def. 4.24), then for any mapping ρ from* pKA *variables to probabilistic programs in $\mathcal{L}S$, $\llbracket b\rrbracket_\rho \sqsubseteq_{\mathcal{L}} \llbracket a\rrbracket_\rho$.*

Theorem 4.26 and Corollary 4.27 enable us to use the probabilistic model $\mathcal{L}S$ to search for counterexamples to conjectures about *pKA* expressions. Unfortunately however, models of finite size are not sufficient for the real-number domain needed to model probability distributions. In the next section, we propose an abstraction of $\mathcal{L}S$ which overcomes this problem and yields genuinely finite models.

4.3.2 The Abstract Model $\mathcal{K}S$

The basic idea of the abstraction is to replace a probability distribution by a simple set, in fact its *support*.

Definition 4.28 (Support)**.** Let $d\colon S^\top \to [0,1]$ be a discrete probability distribution over $S^\top$. The set $\{s \in S^\top \mid d(s) \neq 0\}$ is called the *support of* d, written $\operatorname{supp} d$.

The support only contains the information of which transitions are probabilistic, and the range over which each transition extends. Note that $\operatorname{supp} d$ is non-empty.

Remark 4.29. Let $d\colon S^\top \to [0,1]$ be a discrete probability distribution over $S^\top$. Then $\operatorname{supp} d \neq \emptyset$.

Proof. The remark follows immediately from $d(S^\top) = 1$ (Def. 4.5). □

We call $\operatorname{supp} d$ the *abstract distribution* associated with the probability distribution d. This abstraction induces an order on subsets of $S^\top$: two subsets (i.e. two abstract distributions) are comparable iff there exist corresponding probability distributions that are comparable under $\sqsubseteq_{\mathcal{D}}$. The next definition reformulates this idea without referring to probability distributions at all.

Definition 4.30 (Order on Abstract Distributions)**.** Let a, $a' \subseteq S^\top$ be two abstract distributions. Then

$$a \sqsubseteq_{\mathcal{A}} a' \quad \text{iff} \quad a = a' \vee \{\top\} \subseteq a' \subseteq \{\top\} \cup a.$$

With this definition, the order $\sqsubseteq_{\mathcal{D}}$ on probability distributions is preserved by the abstraction.

Lemma 4.31. *Let* d, $d' \in \overline{S^\top}$. *If* $d \sqsubseteq_{\mathcal{D}} d'$, *then* $\operatorname{supp} d \sqsubseteq_{\mathcal{A}} \operatorname{supp} d'$.

Proof. Suppose $d \sqsubseteq_{\mathcal{D}} d'$. If $d = d'$, then trivially $\operatorname{supp} d \sqsubseteq_{\mathcal{A}} \operatorname{supp} d'$. If $d \neq d'$, then $d(s) \geq d'(s)$ for all $s \in S$ implies $d(\top) < d'(\top)$ (cf. Remark 4.7). Thus $\top \in \operatorname{supp} d'$. Furthermore, $d(s) \geq d'(s)$ for all $s \in S$ also implies $S \cap \operatorname{supp} d' \subseteq S \cap \operatorname{supp} d$. Hence $\operatorname{supp} d' \subseteq \{\top\} \cup \operatorname{supp} d$. □

The converse of Lemma 4.31 is not true: clearly $\operatorname{supp} d \sqsubseteq_{\mathcal{A}} \operatorname{supp} d'$ does not (in general) imply $d \sqsubseteq_{\mathcal{D}} d'$. (For a counterexample, consider e.g. $d := \delta_{a\,p} \oplus \delta_b$, $d' := \delta_{a\,p'} \oplus \delta_b$, with a, $b \in S$, $a \neq b$, $0 < p < 1$, $0 < p' < 1$, and $p \neq p'$.) This shows that replacing a probability distribution by its abstract counterpart entails a certain loss of information. However, we have the following—only slightly weaker—implication.

Lemma 4.32. *Let* a, $a' \subseteq S^\top$ *be two abstract distributions with* $a \sqsubseteq_{\mathcal{A}} a'$. *Suppose* $a = \operatorname{supp} d$ *for some* $d \in \overline{S^\top}$. *Then* $a' = \operatorname{supp} d'$ *for some* $d' \in \overline{S^\top}$ *with* $d \sqsubseteq_{\mathcal{D}} d'$.

Proof. If $a = a'$, take d' to be equal to d. Otherwise we have $\{\top\} \subseteq a' \subseteq \{\top\} \cup a$. Define $d'\colon S^\top \to [0,1]$ as follows:

$$d'(s) := \begin{cases} \frac{d(s)}{2}, & \text{if } s \in a' \setminus \{\top\}; \\ 1 - \frac{d(a' \setminus \{\top\})}{2}, & \text{if } s = \top; \\ 0, & \text{otherwise.} \end{cases}$$

Clearly $\sum_{s \in S^\top} d'(s) = 1$, $a' = \operatorname{supp} d'$, and $d(s) \geq d'(s)$ for all $s \in S$ (hence $d \sqsubseteq_{\mathcal{D}} d'$ as required). □

The *space of abstract probabilistic programs* now uses abstract distributions. Again we impose certain healthiness conditions; these are suitable abstractions of those required in Def. 4.8. In particular *union closure* is an abstraction of convex closure. Cauchy closure on the other hand has no corresponding condition in the abstract model.

Definition 4.33 (Space of Abstract Probabilistic Programs)**.** The *space of abstract probabilistic programs* is given by $(\mathcal{K}S, \sqsubseteq_{\mathcal{K}})$, where

$$\mathcal{K}S := \big\{A \in S^\top \to \mathcal{P}(\mathcal{P}(S^\top)) \quad | \quad \forall s \in S^\top.\, A(s) \text{ is up- and union-closed} \\ \wedge\, A(s) \neq \emptyset \wedge \emptyset \notin A(s) \wedge A(\top) = \{\{\top\}\}\big\},$$

and the order between abstract programs is defined by

$$A \sqsubseteq_{\mathcal{K}} A' \quad \text{iff} \quad \forall s \in S.\, A'(s) \subseteq A(s).$$

Based on the association of abstract distributions with probability distributions, we define an abstraction function from probabilistic programs to their abstract counterparts.

Definition 4.34 (Abstraction of Probabilistic Programs)**.** The *abstraction projection* $\varepsilon\colon \mathcal{L}S \to \mathcal{K}S$ is given by $\varepsilon(P)(s) := \{\operatorname{supp} d \mid d \in P(s)\}$.

Remark 4.35. $\varepsilon\colon \mathcal{L}S \to \mathcal{K}S$ is well-defined, i.e. if P is a probabilistic program, then $\varepsilon(P) \in \mathcal{K}S$.

Proof. Let $s \in S^\top$. Up-closure of $\varepsilon(P)(s)$ follows from up-closure of $P(s)$ (using Lemma 4.32), and union closure of $\varepsilon(P)(s)$ follows from convex closure of $P(s)$ (noting that for d, $d' \in \overline{S^\top}$, we have $\operatorname{supp} d \cup \operatorname{supp} d' = \operatorname{supp}(d\,{}_{1/2}\!\oplus d')$). Next $P(s) \neq \emptyset$ implies $\varepsilon(P)(s) \neq \emptyset$, and Remark 4.29 implies $\emptyset \notin \varepsilon(P)(s)$. Finally $P(\top) = \{\delta_\top\}$ implies $\varepsilon(P)(\top) = \{\{\top\}\}$. □

The abstraction projection preserves the order on programs.

Lemma 4.36. *Let P, $P' \in \mathcal{L}S$. If $P \sqsubseteq_{\mathcal{L}} P'$, then $\varepsilon(P) \sqsubseteq_{\mathcal{K}} \varepsilon(P')$.*

Proof. The lemma follows immediately from the definitions of ε (Def. 4.34), $\sqsubseteq_{\mathcal{L}}$ (Def. 4.8), and $\sqsubseteq_{\mathcal{K}}$ (Def. 4.33). □

For a set of abstract distributions $A \subseteq \mathcal{P}(S^\top)$, we write $\lceil A \rceil$ for the smallest up- and union-closed set of abstract distributions containing A.

Definition 4.37 (Up-, Union-Closed Hull)**.** For $A \subseteq \mathcal{P}(S^\top)$, $\emptyset \notin A$, let

$$\lceil A \rceil := \bigcap \{X \supseteq A \mid X \subseteq \mathcal{P}(S^\top) \text{ is up- and union-closed}\}.$$

Remark 4.38. Let $A \subseteq \mathcal{P}(S^\top)$, $\emptyset \notin A$. $\lceil A \rceil$ is the smallest up- and union-closed superset of A. Hence $\lceil \cdot \rceil$ is a closure operator, i.e. extensive ($A \subseteq \lceil A \rceil$), monotonically increasing ($A \subseteq A' \subseteq \mathcal{P}(S^\top)$ implies $\lceil A \rceil \subseteq \lceil A' \rceil$), and idempotent ($\lceil \lceil A \rceil \rceil = \lceil A \rceil$). Moreover, $\emptyset \notin \lceil A \rceil$.

Proof. We note that an arbitrary intersection of up-closed subsets of $\mathcal{P}(S^\top)$ is up-closed again; likewise an intersection of union-closed subsets is union-closed. The various claims of Remark 4.38 now have standard proofs. To show $\emptyset \notin \lceil A \rceil$, we note that $\lceil A \rceil \setminus \{\emptyset\}$ is an up-closed, union-closed superset of A. □

From Remark 4.38, characterizing $\lceil A \rceil$ as the smallest up- and union-closed superset of A, we derive an induction principle that is useful for proofs: let $a \in \lceil A \rceil$. If P is a predicate that is (i) satisfied by every element of A, and (ii) $P(a_1)$ and $a_1 \sqsubseteq_{\mathcal{A}} a_2$ imply $P(a_2)$ (for every pair of abstract distributions a_1, $a_2 \subseteq S^\top$), and (iii) $P(a_1)$ and $P(a_2)$ imply $P(a_1 \cup a_2)$ (again for every pair of abstract distributions a_1, $a_2 \subseteq S^\top$), then $P(a)$ holds.

The following lemma, although slightly technical, states an important fact about the relationship between (up-, convex) closure of sets of probability distributions, and (up-, union) closure of sets of abstract distributions.

Lemma 4.39. *If $D \subseteq \overline{S^\top}$ is Cauchy-closed, then $\{\operatorname{supp} d \mid d \in \lceil D \rceil\} = \lceil \{\operatorname{supp} d \mid d \in D\} \rceil$.*

Proof. First we show $\lceil \{\operatorname{supp} d \mid d \in D\} \rceil \subseteq \{\operatorname{supp} d \mid d \in \lceil D \rceil\}$, using the induction principle that follows from Remark 4.38.

1. Suppose $a \subseteq S^\top$ is an abstract distribution with $a = \operatorname{supp} d$ for some $d \in D$. Then clearly $a \in \{\operatorname{supp} d \mid d \in \lceil D \rceil\}$ (because $D \subseteq \lceil D \rceil$ by Remark 4.12).

2. Suppose a_1, $a_2 \subseteq S^\top$ are abstract distributions with $a_1 = \operatorname{supp} d_1$ for some $d_1 \in \lceil D \rceil$, and $a_1 \sqsubseteq_{\mathcal{A}} a_2$. By Lemma 4.32, $a_2 = \operatorname{supp} d_2$ for some $d_2 \in \overline{S^\top}$ with $d_1 \sqsubseteq_{\mathcal{D}} d_2$. Then $d_2 \in \lceil D \rceil$ because $\lceil D \rceil$ is up-closed. Hence $a_2 \in \{\operatorname{supp} d \mid d \in \lceil D \rceil\}$.

3. Suppose a_1, $a_2 \subseteq S^\top$ are abstract distributions with $a_i = \operatorname{supp} d_i$ for some $d_i \in \lceil D \rceil$ (for $i = 1, 2$). Then $d_1 \mathrel{{}_{1/2}\!\oplus} d_2 \in \lceil D \rceil$ because $\lceil D \rceil$ is convex-closed. Hence $a_1 \cup a_2 = \operatorname{supp}(d_1 \mathrel{{}_{1/2}\!\oplus} d_2) \in \{\operatorname{supp} d \mid d \in \lceil D \rceil\}$.

Second we show $\{\operatorname{supp} d \mid d \in \lceil D \rceil\} \subseteq \lceil \{\operatorname{supp} d \mid d \in D\} \rceil$. By Lemma 4.17, we have $\lceil D \rceil = H_{\text{convex}}(H_{\text{up}}(D))$. Now suppose $a \subseteq S^\top$ is an abstract distribution with $a = \operatorname{supp} d$ for some $d \in H_{\text{convex}}(H_{\text{up}}(D))$. Then $d = \sum_{i=1}^{k} \alpha_i d_i$ for some $k \in \mathbb{N}$, $k \geq 1$, $d_i \in H_{\text{up}}(D)$, $\alpha_i \in \mathbb{R}$, $\alpha_i \geq 0$, $\sum_{i=1}^{k} \alpha_i = 1$. For each d_i there exists $d_i' \in D$ with $d_i' \sqsubseteq_{\mathcal{D}} d_i$. Clearly (for $1 \leq i \leq k$) $\operatorname{supp} d_i' \in \lceil \{\operatorname{supp} d \mid d \in D\} \rceil$ (because $\{\operatorname{supp} d \mid d \in D\} \subseteq \lceil \{\operatorname{supp} d \mid d \in D\} \rceil$ by Remark 4.38). Lemma 4.31 implies $\operatorname{supp} d_i' \sqsubseteq_{\mathcal{A}} \operatorname{supp} d_i$. Hence also $\operatorname{supp} d_i \in \lceil \{\operatorname{supp} d \mid d \in D\} \rceil$ because $\lceil \{\operatorname{supp} d \mid d \in D\} \rceil$ is up-closed. Now (assuming, without loss of generality, $\alpha_i > 0$ for $1 \leq i \leq k$) $a = \bigcup_{i=1}^{k} \operatorname{supp} d_i \in \lceil \{\operatorname{supp} d \mid d \in D\} \rceil$ because $\lceil \{\operatorname{supp} d \mid d \in D\} \rceil$ is union-closed. □

The abstraction projection maps probabilistic programs to their abstract counterparts, but we can also go the other way. For every abstract program A, there exists a probabilistic program P such that $\varepsilon(P) = A$. In other words, $\varepsilon \colon \mathcal{L}S \to \mathcal{K}S$ is onto.

Lemma 4.40. *For every abstract program $A \in \mathcal{K}S$ exists a probabilistic program $P \in \mathcal{L}S$ with $\varepsilon(P) = A$.*

Proof. For every abstract distribution $a \subseteq S^\top$, let $d_a \colon S^\top \to [0, 1]$ denote the uniform distribution over a (i.e. $d_a(s) := \frac{1}{|a|}$ if $s \in a$, $d_a(s) := 0$ otherwise). Clearly $\operatorname{supp} d_a = a$.

Define $P \colon S^\top \to \mathcal{P}(\overline{S^\top})$ by $P(s) := \lceil \{d_a \mid a \in A(s)\} \rceil$. $P(s)$ is up-, convex-, and Cauchy-closed by definition of $\lceil \cdot \rceil$, $P(s) \neq \emptyset$ because $A(s) \neq \emptyset$, and $A(\top) = \{\{\top\}\}$ implies $P(\top) = \{\delta_\top\}$. Hence $P \in \mathcal{L}S$.

It remains to show $\varepsilon(P) = A$. Let $s \in S^\top$. We note that $\{d_a \mid a \in A(s)\}$ is finite (since $S^\top$ is finite), hence Cauchy-closed. Therefore, using Lemma 4.39, $\varepsilon(P)(s) = \{\operatorname{supp} d \mid d \in \lceil\{d_a \mid a \in A(s)\}\rceil = \lceil\{\operatorname{supp} d_a \mid a \in A(s)\}\rceil = \lceil A(s)\rceil = A(s)$. □

We remark that if two probabilistic programs have the same abstraction, then their iterations also have the same abstraction.

Lemma 4.41. *Let P, $P' \in \mathcal{L}S$. If $\varepsilon(P) = \varepsilon(P')$, then $\varepsilon(P^*) = \varepsilon(P'^*)$.*

A proof of this non-trivial lemma is given in [108] (see in particular [108, Lemma 1] and [108, Lemma 4]).

Next, we define operators on $\mathcal{K}S$ that correspond to the operators on $\mathcal{L}S$ given in Def. 4.21.

Definition 4.42 (Operators on $\mathcal{K}S$)**.** For arbitrary states $s \in S^\top$ and abstract programs A, $A' \in \mathcal{K}S$, we define

Identity	$\mathrm{Id}(s)$	:=	$\lceil\{\{s\}\}\rceil$,
Top	$\top(s)$	:=	$\{\{\top\}\}$,
Composition	$(A; A')(s)$	:=	$\{\bigcup_{u\in a} a'_u \mid a \in A(s), \forall u \in a.\, a'_u \in A'(u)\}$,
Probability	$(A \oplus A')(s)$	:=	$\{a \cup a' \mid a \in A(s), a' \in A'(s)\}$,
Non-determinism	$(A \sqcap A')(s)$	:=	$\lceil A(s) \cup A'(s)\rceil$,
Iteration	$A^*(s)$	:=	$\varepsilon(P^*)(s)$, for an arbitrary $P \in \mathcal{L}S$ with $\varepsilon(P) = A$.

Remark 4.43. Let A, $A' \in \mathcal{K}S$. Then $\mathrm{Id} \in \mathcal{K}S$, $\top \in \mathcal{K}S$, $(A; A') \in \mathcal{K}S$, $A \oplus A' \in \mathcal{K}S$, $A \sqcap A' \in \mathcal{K}S$, and $A^* \in \mathcal{K}S$.

Proof. We note that for every abstract distribution $a \subseteq S^\top$, $\{\top\} \sqsubseteq_{\mathcal{A}} a$ iff $a = \{\top\}$. Hence $\lceil\{\{\top\}\}\rceil = \{\{\top\}\}$.

Let $s \in S^\top$. For $\mathrm{Id}(s)$ and $\top(s)$, all relevant properties (i.e. up-closure, union closure, non-emptiness, $\emptyset \notin \mathrm{Id}(s)$, $\emptyset \notin \top(s)$, and $\mathrm{Id}(\top) = \top(\top) = \{\{\top\}\}$) now follow immediately from Def. 4.42.

Up-closure of $(A; A')(s)$ follows from up-closure of $A'(u)$ for all $u \in a$. Using $(\bigcup_{u\in a} a'_u) \cup (\bigcup_{u\in b} b'_u) = \bigcup_{u\in a\cup b} x_u$ (for arbitrary abstract distributions a, a'_u, b, $b'_u \subseteq S^\top$, where $x_u := a'_u$ if $u \in a \setminus b$, $x_u := b'_u$ if $u \in b \setminus a$, and $x_u := a'_u \cup b'_u$ if $u \in a \cap b$), union closure of $(A; A')(s)$ follows from union closure of $A(s)$ and union closure of $A'(u)$ for all $u \in a \cap b$. The remaining properties, i.e. $(A; A')(s) \neq \emptyset$, $\emptyset \notin (A; A')(s)$, and $(A; A')(\top) = \{\{\top\}\}$, are immediate.

Up-closure of $(A \oplus A')(s)$ follows from up-closure of $A(s)$ and $A'(s)$. Union closure of $(A \oplus A')(s)$ follows from union closure of $A(s)$ and $A'(s)$, using $(a \cup a') \cup (b \cup b') = (a \cup b) \cup (a' \cup b')$ (for arbitrary abstract distributions a, a', b, $b' \subseteq S^\top$). The remaining properties, i.e. $(A \oplus A')(s) \neq \emptyset$, $\emptyset \notin (A \oplus A')(s)$, and $(A \oplus A')(\top) = \{\{\top\}\}$, are immediate.

For $(A \sqcap A')(s)$, all relevant properties are immediate again from Def. 4.42.

For A^*, we only need to show that $A^*(s)$ is well-defined. All relevant properties then follow immediately from Remark 4.35. Lemma 4.40 implies the existence of at least one probabilistic program $P \in \mathcal{L}S$ with $\varepsilon(P) = A$, and Lemma 4.41 shows that for any two programs P, $P' \in \mathcal{L}S$ with $\varepsilon(P) = \varepsilon(P') = A$, we have $\varepsilon(P^*) = \varepsilon(P'^*)$. □

The above definition of A^* still refers to the probabilistic model $\mathcal{L}S$. In [108] we show how A^* can be computed without referring to any underlying probabilistic program, by determining the sets of states that are *reachable with probability* 1. It is well-known that this is possible using the information provided by the abstract transitions alone; for example de Alfaro and Henzinger [46] provide such an algorithm with complexity quadratic in the size of the underlying transition system.

Finally we can give an interpretation of *pKA* expressions in the space of abstract programs.

Definition 4.44 (Semantic Mapping of *pKA* Expressions to $\mathcal{K}S$)**.** The semantic mapping $[\![\cdot]\!]_\rho$ from *pKA* expressions to $\mathcal{K}S$ is parameterized by a mapping ρ from *pKA* variables to abstract probabilistic programs in $\mathcal{K}S$, and defined as follows:

1. If x is a variable, then $[\![x]\!]_\rho := \rho(x)$.
2. $[\![0]\!]_\rho := \top$, $[\![1]\!]_\rho := \mathrm{Id}$.
3. $[\![ab]\!]_\rho := [\![a]\!]_\rho ; [\![b]\!]_\rho$.
4. $[\![a + b]\!]_\rho := [\![a]\!]_\rho \sqcap [\![b]\!]_\rho$.
5. $[\![a^*]\!]_\rho := [\![a]\!]_\rho{}^*$.

While this definition looks very similar to Def. 4.25 (due to the use of overloaded notation), the operators on the right-hand side of the above equations are now of course those on $\mathcal{K}S$, as given in Def. 4.42.

We do not claim that the axioms of probabilistic Kleene algebra are satisfied by this interpretation; in fact Axiom (13) fails to hold. The abstract program $s_0 \mapsto \lceil\{\{s_0, s_1\}\}\rceil$, $s_1 \mapsto \lceil\{\{s_1\}\}\rceil$ denoting both a and b is a counterexample. Thus there is no analogue of Theorem 4.26 for $\mathcal{K}S$. This is because the abstraction does not (in general) preserve inequalities; see the earlier discussion on the converse of Lemma 4.31.

The next lemma gives the relationship between interpretations in $\mathcal{L}S$ and in $\mathcal{K}S$: they correspond homomorphically.

Lemma 4.45. *Let e be a* pKA *expression, and let ρ be a mapping from* pKA *variables to probabilistic programs in $\mathcal{L}S$. Then*

$$\varepsilon([\![e]\!]_\rho) = [\![e]\!]_{\varepsilon\circ\rho}.$$

Proof. By structural induction on e. If e is a *pKA* variable, then

$$\varepsilon([\![e]\!]_\rho) \overset{4.25}{=} \varepsilon(\rho(e)) = (\varepsilon \circ \rho)(e) \overset{4.44}{=} [\![e]\!]_{\varepsilon\circ\rho}.$$

Moreover

$$\varepsilon([\![0]\!]_\rho) \overset{4.25}{=} \varepsilon(\top) = \top \overset{4.44}{=} [\![0]\!]_{\varepsilon\circ\rho},$$

and

$$\varepsilon([\![1]\!]_\rho) \overset{4.25}{=} \varepsilon(\mathrm{Id}) \overset{4.39}{=} \mathrm{Id} \overset{4.44}{=} [\![1]\!]_{\varepsilon\circ\rho}.$$

If $e = ab$ for *pKA* expressions a and b, then for any $s \in S^\top$,

$$\begin{aligned}
\varepsilon(\llbracket e \rrbracket_\rho)(s) &\overset{4.34}{=} \{\operatorname{supp} d \mid d \in \llbracket ab \rrbracket_\rho(s)\} \\
&\overset{4.25}{=} \left\{ \operatorname{supp}\left(\sum_{u \in S^\top} a'(u) \cdot b'_u \right) \,\middle|\, a' \in \llbracket a \rrbracket_\rho(s), \forall u \in S^\top . \, b'_u \in \llbracket b \rrbracket_\rho(u) \right\} \\
&= \left\{ \bigcup_{u \in \operatorname{supp} a'} \operatorname{supp} b'_u \,\middle|\, a' \in \llbracket a \rrbracket_\rho(s), \forall u \in \operatorname{supp} a' . \, b'_u \in \llbracket b \rrbracket_\rho(u) \right\} \\
&\overset{4.34}{=} \left\{ \bigcup_{u \in a'} b'_u \,\middle|\, a' \in \varepsilon(\llbracket a \rrbracket_\rho)(s), \forall u \in a' . \, b'_u \in \varepsilon(\llbracket b \rrbracket_\rho)(u) \right\} \\
&\overset{\text{IH}}{=} \left\{ \bigcup_{u \in a'} b'_u \,\middle|\, a' \in \llbracket a \rrbracket_{\varepsilon \circ \rho}(s), \forall u \in a' . \, b'_u \in \llbracket b \rrbracket_{\varepsilon \circ \rho}(u) \right\} \\
&\overset{4.44}{=} \llbracket ab \rrbracket_{\varepsilon \circ \rho}(s).
\end{aligned}$$

If $e = a + b$ for *pKA* expressions a and b, then for any $s \in S^\top$,

$$\begin{aligned}
\varepsilon(\llbracket e \rrbracket_\rho)(s) &\overset{4.34}{=} \{\operatorname{supp} d \mid d \in \llbracket a + b \rrbracket_\rho(s)\} \\
&\overset{4.25}{=} \{\operatorname{supp} d \mid d \in \lceil \llbracket a \rrbracket_\rho(s) \cup \llbracket b \rrbracket_\rho(s) \rceil\} \\
&\overset{4.39}{=} \lceil \{\operatorname{supp} d \mid d \in \llbracket a \rrbracket_\rho(s) \cup \llbracket b \rrbracket_\rho(s)\} \rceil \\
&\overset{4.34}{=} \lceil \varepsilon(\llbracket a \rrbracket_\rho)(s) \cup \varepsilon(\llbracket b \rrbracket_\rho)(s) \rceil \\
&\overset{\text{IH}}{=} \lceil \llbracket a \rrbracket_{\varepsilon \circ \rho}(s) \cup \llbracket b \rrbracket_{\varepsilon \circ \rho}(s) \rceil \\
&\overset{4.44}{=} \llbracket a + b \rrbracket_{\varepsilon \circ \rho}(s).
\end{aligned}$$

If $e = a^*$ for some *pKA* expression a, then

$$\varepsilon(\llbracket e \rrbracket_\rho) \overset{4.25}{=} \varepsilon(\llbracket a \rrbracket_\rho{}^*) \overset{4.42}{=} \varepsilon(\llbracket a \rrbracket_\rho)^* \overset{\text{IH}}{=} \llbracket a \rrbracket_{\varepsilon \circ \rho}{}^* \overset{4.44}{=} \llbracket a^* \rrbracket_{\varepsilon \circ \rho}.$$

□

Thus we have set up a model for abstract probabilistic programs in which the precise weights attached to probabilistic transitions have been suppressed, while the limit properties of probability theory are retained. Next we show how the abstract model can be used to obtain counterexamples in $\mathcal{L}S$.

Lemma 4.46. *Let e and f be* pKA *expressions. If $e \neq f$ is satisfiable in $\mathcal{K}S$, then it is also satisfiable in $\mathcal{L}S$.*

Proof. Suppose $e \neq f$ is satisfiable in $\mathcal{K}S$, i.e. there exists a mapping ρ from *pKA* variables to abstract programs in $\mathcal{K}S$ such that $\llbracket e \rrbracket_\rho \neq \llbracket f \rrbracket_\rho$.

By Lemma 4.40, there exists a mapping ρ' from *pKA* variables to probabilistic programs in $\mathcal{L}S$ such that $\varepsilon \circ \rho' = \rho$. Now

$$\varepsilon(\llbracket e \rrbracket_{\rho'}) \overset{4.45}{=} \llbracket e \rrbracket_{\varepsilon \circ \rho'} = \llbracket e \rrbracket_\rho \neq \llbracket f \rrbracket_\rho = \llbracket f \rrbracket_{\varepsilon \circ \rho'} \overset{4.45}{=} \varepsilon(\llbracket f \rrbracket_{\rho'}),$$

hence $[\![e]\!]_{\rho'} \neq [\![f]\!]_{\rho'}$ in $\mathcal{L}S$. □

Finally this section's main result follows. If a counterexample exists in $\mathcal{K}S$ to a conjectured pKA equality, then the equality is not provable in pKA.

Corollary 4.47. *Let e and f be* pKA *expressions. If $e \neq f$ is satisfiable in $\mathcal{K}S$, then the equality $e = f$ is not provable by probabilistic Kleene algebra rules.*

Proof. By Lemma 4.46, $e \neq f$ is satisfiable in $\mathcal{L}S$, and by Theorem 4.26, interpretations in $\mathcal{L}S$ satisfy the rules of probabilistic Kleene algebra. □

The corollary implies that automated counterexample search for equalities in pKA can be based on state exploration of finite models in $\mathcal{KS}$.

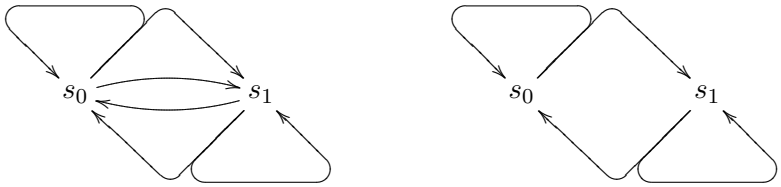
4.3.3 Mechanization of Counterexample Search

We have defined the abstract model $\mathcal{K}S$ in Isabelle/HOL. The type S option $\to$ S option set set is used for abstract programs, with None encoding $\top$. A well-formedness predicate selects those functions in this type that satisfy the constraints of Def. 4.33. Next, we have defined the various operators on $\mathcal{K}S$ (see Defs. 4.42 and 4.44), in particular composition, non-determinism, and iteration. With the exception of the iteration operator, $\cdot^*$, formulae that contain these operators can be translated to propositional logic by the algorithm presented in Chapters 2 and 3. Iteration could in principle be translated as well (by unfolding its Isabelle/HOL definition, as it is done for the other operators), but this unfortunately leads to unacceptable performance. That the translation of iteration is challenging also seems to be the case in other systems which use SAT solving in the context of $\cdot^*$-like operators [82].

We have therefore implemented dedicated SML code for the iteration operator, which translates this operator to a tree directly, without unfolding its definition. This is an application of the first optimization technique described in the paragraph on unfolding and specialization in Section 3.2. Our code does not actually implement a reachability algorithm, but merely contains precomputed result trees for small state spaces. Due to the exponential growth of S option $\to$ S option set set (in the size of S), we are still limited to small state spaces anyway, despite our optimizations. Fortunately, counterexamples in practice do appear to be exhibited within very small state spaces.

Two of the perhaps more interesting conjectures about probabilistic programs that Isabelle can refute automatically are (i) $P^* \leq P$ and (ii) $P^*; P = P; P^*$. Graphical representations of the abstract programs that were found as counterexamples are shown in Figure 4.3. (For clarity, only $\sqsubseteq_{\mathcal{A}}$-minimal distributions are shown, and also the transition $\top \mapsto \{\{\top\}\}$ has been omitted.) Both programs use a two-element state space $S = \{s_0, s_1\}$. The second program happens to be the abstraction of the coin toss example discussed earlier. One may notice that each of the two programs refutes *both* conjectures. That different counterexamples are returned for the two conjectures is a coincidence; ultimately it depends on the underlying SAT solver that was used for model generation.

To deal with slightly larger state spaces, we could use specialization (as described in Section 3.2): the iteration operator, although it might occur without an argument in a HOL



(i) $P^* \leq P$ (ii) $P^*; P = P; P^*$

Figure 4.3: Abstract probabilistic counterexamples

formula, never does so in a *pKA* expression; instead, it is always applied to some program P. Therefore we do not need to build a tree for the entire function representing $\cdot^*$, but we only need the much smaller tree for P^*. Of course this would require the implementation of a symbolic reachability algorithm, which could compute the tree for P^* from a tree (possibly containing variables or complex propositional formulae as label elements) for P.

With this technique, the main bottleneck will then be the size of propositional formulae in labels of the tree for P^*. Even if the tree for P contains variables only (and not more complex formulae), formulae in the tree for P^* would quickly become huge. The problem is aggravated because at least some of these formulae later need to be translated to CNF. A hybrid approach, where the program P is partially known (e.g. because of an earlier case distinction), while other parts are only given symbolically, could solve this issue. The known parts of the program could be used to immediately simplify the resulting propositional formulae, as described in the paragraph on propositional simplification in Section 3.2, thereby keeping these formulae reasonably small.

4.4 A SAT-based Sudoku Solver

This section presents a SAT-based Sudoku solver. A Sudoku is translated into a propositional formula that is satisfiable if and only if the Sudoku has a solution. A standard SAT solver can then be applied, and a solution for the Sudoku can be read off from the satisfying assignment returned by the SAT solver. No coding was necessary to implement this Sudoku solver: the translation into propositional logic is provided by our algorithm for finite model generation that was described in Chapters 2 and 3 of this thesis. Only the constraints on a Sudoku solution have to be specified in Isabelle/HOL.

Sudoku, also known as *Number Place* in the United States, is a placement puzzle. Given a grid—most frequently a 9×9 grid made up of 3×3 subgrids called *regions*—with various digits given in some cells (the *givens*), the aim is to enter a digit from 1 through 9 in each cell of the grid so that each row, column and region contains only one instance of each digit. Figure 4.4 shows a Sudoku on the left, along with its unique solution on the right [172]. Note that other symbols (e.g. letters, icons) could be used instead of digits, as their arithmetic properties are irrelevant in the context of Sudoku. This is currently a rather popular puzzle that is featured in a number of newspapers and puzzle magazines [5, 49, 158].

Several Sudoku solvers are available already [97, 163]. Since there are more than $6 \cdot 10^{21}$ pos-

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.4: Sudoku example and solution

sible Sudoku grids [55], a naive backtracking algorithm would be infeasible. Sudoku solvers therefore combine backtracking with—sometimes complicated—methods for constraint propagation. Here we propose a SAT-based approach: a Sudoku is translated into a propositional formula that is satisfiable if and only if the Sudoku has a solution. The propositional formula is then presented to a standard SAT solver, and if the SAT solver finds a satisfying assignment, this assignment can readily be transformed into a solution for the original Sudoku. The presented translation into SAT is simple, and requires minimal implementation effort since we can reuse our framework for finite model generation.

4.4.1 Implementation in Isabelle/HOL

An implementation of the Sudoku rules in Isabelle/HOL is straightforward. Digits are modelled by a datatype with nine elements $1, \ldots, 9$. We say that nine grid cells $x_1, \ldots, x_9$ are *valid* iff they contain every digit.

Definition 4.48 (valid)**.**

$$\text{valid}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) \equiv \bigwedge_{d=1}^{9} \bigvee_{i=1}^{9} x_i = d.$$

Labeling the 81 cells of a 9×9 grid as shown in Figure 4.5, we can now define what it means for them to be a Sudoku solution: each row, column and region must be valid.

Definition 4.49 (sudoku)**.**

$$\begin{aligned}\text{sudoku}(\{x_{ij}\}_{i,j\in\{1,\ldots,9\}}) \equiv{}& \bigwedge_{i=1}^{9} \text{valid}(x_{i1}, x_{i2}, x_{i3}, x_{i4}, x_{i5}, x_{i6}, x_{i7}, x_{i8}, x_{i9}) \\ \wedge{}& \bigwedge_{j=1}^{9} \text{valid}(x_{1j}, x_{2j}, x_{3j}, x_{4j}, x_{5j}, x_{6j}, x_{7j}, x_{8j}, x_{9j}) \\ \wedge{}& \bigwedge_{i,j\in\{1,4,7\}} \text{valid}(x_{ij}, x_{i(j+1)}, x_{i(j+2)}, x_{(i+1)j}, x_{(i+1)(j+1)}, x_{(i+1)(j+2)}, \\ & x_{(i+2)j}, x_{(i+2)(j+1)}, x_{(i+2)(j+2)}).\end{aligned}$$

The next section describes the translation of these definitions into propositional logic.

x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}	x_{29}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}	x_{39}
x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}
x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}	x_{57}	x_{58}	x_{59}
x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}
x_{71}	x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}
x_{81}	x_{82}	x_{83}	x_{84}	x_{85}	x_{86}	x_{87}	x_{88}	x_{89}
x_{91}	x_{92}	x_{93}	x_{94}	x_{95}	x_{96}	x_{97}	x_{98}	x_{99}

Figure 4.5: Sudoku grid

4.4.2 Translation to Propositional Logic

The translation to propositional logic is an application of the general translation for HOL formulae that was described in Section 2.3. We encode a Sudoku by introducing 9 Boolean variables for each cell of the 9×9 grid, i.e. $9^3 = 729$ variables in total. Each Boolean variable p_{ij}^d (with $1 \leq i, j, d \leq 9$) represents the truth value of the equation $x_{ij} = d$. A clause

$$\bigvee_{d=1}^{9} p_{ij}^d$$

ensures that the cell x_{ij} denotes one of the nine digits, and 36 clauses

$$\bigwedge_{1 \leq d < d' \leq 9} \neg p_{ij}^d \vee \neg p_{ij}^{d'}$$

make sure that the cell does not denote two different digits at the same time.

Since there are just as many digits as cells in each row, column, and region, Def. 4.48 is equivalent to the following characterization of validity, stating that the nine grid cells $x_1, \ldots, x_9$ contain distinct values.

Lemma 4.50 (Equivalent Characterization of Validity)**.**

$$\begin{aligned} \text{valid}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) &\iff \bigwedge_{1 \leq i < j \leq 9} x_i \neq x_j \\ &\iff \bigwedge_{1 \leq i < j \leq 9} \bigwedge_{d=1}^{9} x_i \neq d \vee x_j \neq d. \end{aligned}$$

The latter characterization turns out to be much more efficient than the original definition when translated to SAT. While Def. 4.48, when translated directly, produces 9 clauses with 9 literals each (one literal for each equation), the formula given in Lemma 4.50 is translated to

	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

Figure 4.6: Hard Sudoku example and solution

324 clauses (9 clauses for each of the 36 inequations $x_i \neq x_j$), but each clause of length 2 only. This allows for more unit propagation [178] by the SAT solver, which—in terms of the original Sudoku—corresponds to cross-hatching [172] of digits, a technique that is essential to reduce the search space. The 9 clauses obtained from a direct translation of Def. 4.48 could still be used as well; unit propagation on these clauses would correspond to counting the digits 1–9 in regions, rows, and columns to identify missing numbers. However, in our experiments we did not experience any speedup by including these clauses.

This encoding yields a total of 11745 clauses: 81 definedness clauses of length 9, $81 \cdot 36$ uniqueness clauses of length 2, and $27 \cdot 324$ validity clauses,[1] again of length 2. However, we do not need to introduce Boolean variables for cells whose value is given in the original Sudoku, and we can omit definedness and uniqueness clauses for these cells as well as some of the validity clauses—therefore the total number of variables and clauses used in the encoding of a Sudoku with givens will be less than 729 and 11745, respectively. (The importance of this optimization for larger Sudoku puzzles is emphasized in [95], although the authors' claim that we do not perform it is erroneous.)

Note that our encoding already yields a propositional formula in conjunctive normal form (CNF). Therefore conversion into DIMACS CNF format [50]—the standard input format used by most SAT solvers—is trivial. Isabelle can search for a satisfying assignment using either an internal DPLL-based [45] SAT solver, or write the formula to a file in DIMACS format and execute an external solver. We have employed zChaff [119] to find the solution to various Sudoku classified as "hard" by their respective authors (see Figure 4.6 for an example), and in every case the runtime was only a few milliseconds.

Traditionally the givens in a Sudoku are chosen so that the puzzle's solution is unique. Nevertheless in case different solutions exist, our algorithm can be extended to enumerate all of them (by explicitly disallowing all solutions found so far, and perhaps using an incremental SAT solver that allows adding clauses on-the-fly to avoid searching through the same search space multiple times).

Particularly remarkable is the fact that our solver, while it can certainly compete with handcrafted Sudoku solvers, some of which use rather complex patterns and search heuristics, re-

[1] This number includes some duplicates, caused by the overlap between rows/columns and regions: certain cells that must be distinct because they belong to the same row (or column) must also be distinct because they belong to the same region.

quired very little implementation effort. Aside from Lemma 4.50, no domain-specific knowledge was used. The impressive performance is largely due to the SAT solver. Even the translation into propositional logic was not written by hand, but is merely an application of the framework for finite model generation that is readily available in Isabelle/HOL. Only the Sudoku rules had to be defined in the theorem prover's logic, and this was a trouble-free task.

4.5 Conclusion

In this chapter, we have discussed three case studies that illustrate possible applications of finite model generation in general, and of our framework for model generation in Isabelle/HOL in particular: the RSA-PSS security protocol, probabilistic programs, and a Sudoku solver.

In Section 4.2, we have shown security of an abstract formalization of the RSA-PSS security protocol by providing a model that does not allow the attacker to forge certain signatures. The model was finite, but the well-known initial model theorem implies that security also holds in the (infinite) initial model, where the functions for message generation are interpreted in the usual algebraic way.

A model for probabilistic programs that is susceptible to counterexample search via finite model generation has been presented in Section 4.3. This model has been formalized in Isabelle/HOL, where it was used to obtain counterexamples to conjectures about probabilistic Kleene algebra expressions. There are performance issues (especially with the star operator, $\cdot^*$), but for small state spaces, the approach works reasonably well.

In Section 4.4, we have presented a straightforward translation of a Sudoku into a propositional formula. The translation can easily be generalized from 9×9 grids to grids of arbitrary dimension. It is polynomial in the size of the grid, and since Sudoku is NP-complete [176], no algorithm with better complexity is known. The translation, combined with a state-of-the-art SAT solver, is also practically successful: 9×9 Sudoku puzzles are solved within milliseconds.

Together these case studies show that the model generation algorithm presented in Chapters 2 and 3, despite its non-elementary complexity, can be useful for many interesting problems that occur in practice. The algorithm's integration into the Isabelle/HOL theorem prover allows its easy application to any formal development carried out in this system. Higher-order logic is a rich specification language that permitted natural formalizations of all three case studies. Some of the logic's (higher-order) features pose potential performance issues for model generation, but to remedy the situation, it is sometimes possible to carry out formalization "with model generation in mind", avoiding a state-space explosion through the use of logically equivalent (but combinatorially harmless) encodings in the theorem prover.

Love all, trust a few.
William Shakespeare,
1564–1616.

Chapter 5

Integration of Proof-producing SAT Solvers

This chapter describes the integration of zChaff and MiniSat, currently two leading SAT solvers, with Isabelle/HOL. Both SAT solvers generate resolution-style proofs for (instances of) propositional tautologies. These proofs are verified by the theorem prover. The presented approach significantly improves Isabelle's performance on propositional problems.

5.1 Introduction

So far we have discussed the generation of finite models for HOL formulae, the main application being the generation of countermodels for unprovable conjectures. But what if the search for a finite countermodel fails? More specifically, can we use SAT solvers to *prove* theorems as well?

Clearly the failure to produce a finite countermodel up to a certain size for some formula ϕ does not imply validity of this formula. There might still exist a finite countermodel larger than the given size, or even an infinite countermodel. For example, consider the following first-order formula, which *only* has infinite models:

$$(\forall x \exists y.\, P\, x\, y) \wedge (\forall x\, y\, z.\, P\, x\, y \implies P\, y\, z \implies P\, x\, z) \wedge (\forall x.\, \neg P\, x\, x).$$

In general it is undecidable already for first-order logic if a formula ϕ has a model at all [37], and also if it has a finite model [160]. If ϕ has the finite model property [26] however, and the bound on the model size is effectively computable, we could use this bound to limit the search for a model. Since our search algorithm is complete (provided the underlying SAT solver is),

failure in this case does indeed imply validity of $\neg\phi$. Another application of our algorithm could be to find witnesses for monomorphic existential statements, thereby proving them.

Hence model generation could—at least in principle—be used to prove formulae from certain fragments of HOL. However, the semantic reasoning indicated above would be difficult to formalize in an LCF-style [61] theorem prover like Isabelle/HOL, where all proofs in the end must be expressed in terms of the logic's inference rules. (Even proving existential statements wouldn't be without practical challenges. If an incomplete, randomized SAT solver was used, a proof script might work one time and fail another—certainly not a desirable property.) Therefore we only consider instances of *propositional* tautologies in this chapter. Furthermore, we use proof-producing SAT solvers: they are not only able to find a satisfying assignment if one exists, but they also return a (resolution-style) proof of unsatisfiability in case the input formula is not satisfiable. Currently the most successful SAT solvers are DPLL-based [119], and extending such solvers with the ability to produce unsatisfiability proofs is relatively straightforward [179].

5.2 Related Work

Perhaps most closely related to the work in this chapter is John Harrison's LCF-style integration of Stålmarck's algorithm and BDDs into HOL Light and Hol90 respectively [68, 69]. Harrison found that doing BDD operations inside HOL performed about 100 times worse (after several optimizations) than a C implementation.

Michael Gordon implemented *HolSatLib* [62] in Hol98, a precursor to HOL 4. This library provided functions to convert Hol98 terms into CNF, and to analyze them using a SAT solver. In the case of unsatisfiability however, the user only had the option to trust the external solver. No proof reconstruction took place, "since there is no efficient way to check for unsatisfiability using pure Hol98 theorem proving" [62]. A bug in the SAT solver could ultimately lead to an inconsistency in Hol98. The HOL 4 implementation of this library is instead based on ideas discussed in this chapter.

A custom-built SAT solver has been integrated with the CVC Lite system [15] by Clark Barrett et al. [16]. While this solver produces proofs that can be checked independently, our work shows that it is possible to integrate existing, highly efficient solvers with an LCF-style prover: the information provided by recent versions of zChaff and MiniSat is sufficient to produce a proof object in a theorem prover, no custom-built solver is necessary.

Further afield, the integration of automated first-order provers with HOL provers has been explored by Joe Hurd [74, 75], Jia Meng [111], and Lawrence Paulson [112, 113]. Proofs found by the automated system are either verified by the interactive prover immediately [74], or translated into a proof script that can be executed later [112]. Andreas Meier's TRAMP system [109] transforms the output of various automated first-order provers into natural deduction proofs. The main focus of that work however is on the necessary translation from the interactive prover's specification language to first-order logic. In contrast our approach is so far restricted to instances of propositional tautologies, but we have focused on performance (rather than on difficult translation issues), and we use a SAT solver, rather than a first-order prover. Other work on combining proof and model search includes [48].

An earlier version of this work was presented in [167], and improved by Alwen Tiu et al. [58].

Furthermore Hasan Amjad has recently integrated proof-generating versions of zChaff and MiniSat with HOL 4 in a similar fashion [168]. Here we discuss our most recent implementation [166], which also incorporates ideas by John Harrison, John Matthews, and Markus Wenzel. It constitutes a significant performance improvement when compared to earlier implementations.

5.3 System Description

To prove a propositional tautology ϕ in the Isabelle/HOL system with the help of zChaff or MiniSat, we proceed in several steps. First ϕ is negated, and the negation is converted into an equivalent formula ϕ^* in conjunctive normal form. ϕ^* is then written to a file in DIMACS CNF format [50], the standard input format supported by most SAT solvers. zChaff and MiniSat, when run on this file, return either "unsatisfiable", or a satisfying assignment for ϕ^*.

In the latter case, the satisfying assignment is displayed to the user. The assignment constitutes a counterexample to the original (unnegated) conjecture. When the solver returns "unsatisfiable" however, things are more complicated. If we have confidence in the SAT solver, we can simply trust its result and accept ϕ as a theorem in Isabelle. The theorem is tagged with an "oracle" flag to indicate that it was proved not through Isabelle's own inference rules, but by an external tool. In this scenario, a bug in the SAT solver (or in our translation from HOL to propositional logic) could potentially allow us to derive inconsistent theorems in Isabelle/HOL.

The LCF-approach instead demands that we verify the solver's claim of unsatisfiability within Isabelle/HOL. While this is not as simple as the validation of a satisfying assignment, the increasing complexity of SAT solvers has before raised the question of support for independent verification of their results, and in 2003 L. Zhang and S. Malik [179] extended zChaff to generate resolution-style proofs that can be verified by an independent checker. This issue has also been acknowledged by the annual SAT Competition, which introduced a special track on certified "unsatisfiable" answers in 2005. More recently, a proof-logging version of MiniSat was released [53], and John Matthews extended this version to produce human-readable proofs that are easy to parse [101], similar to those produced by zChaff.

One could use an independent (external) proof checker (e.g. written in C) to verify the SAT solver's answer. This might increase the degree of confidence in the result, but it still suffers from potential soundness issues. The independent proof checker, as well as the translation between the different tools, would become part of the trusted code base. Therefore in the LCF framework our main task boils down to using Isabelle/HOL itself as an independent checker for the resolution proofs found by zChaff and MiniSat.

Both solvers store their proof in a text file that is read in by Isabelle, and the individual resolution steps are replayed in Isabelle/HOL. Section 5.3.1 describes the necessary preprocessing of the input formula, and details of the proof reconstruction are explained in Section 5.3.2. The overall system architecture is shown in Figure 5.1.

5.3.1 Preprocessing

Isabelle/HOL offers higher-order logic (on top of Isabelle's meta logic, cf. Section 3.3), whereas most SAT solvers only support formulae of propositional logic in conjunctive normal form.

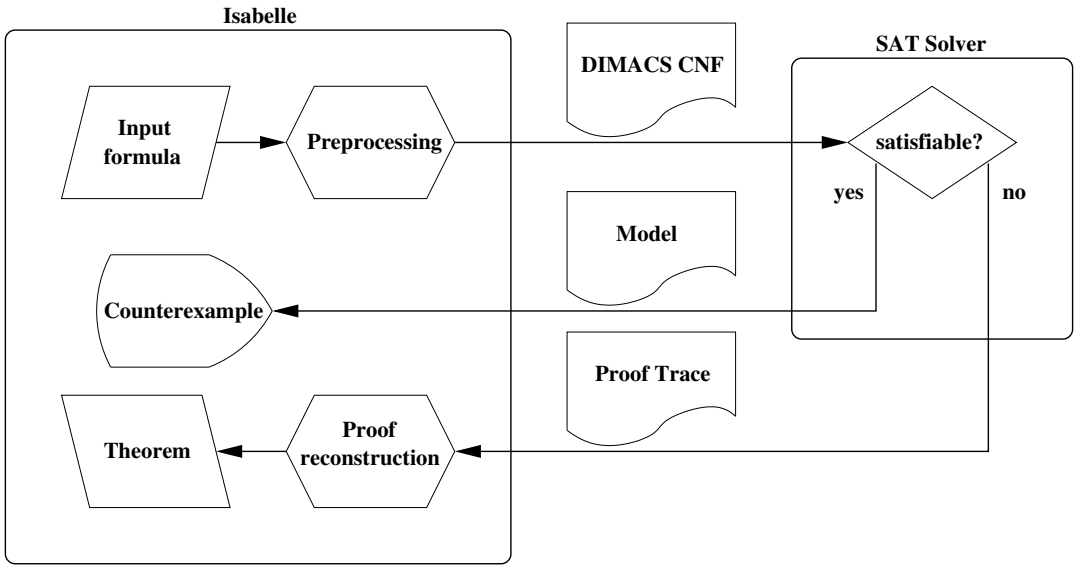


Figure 5.1: Isabelle – SAT system architecture

Therefore the (negated) input formula ϕ must be preprocessed before it can be passed to the solver.

First connectives of the meta logic, namely meta implication ($\Longrightarrow$) and meta equivalence ($\equiv$), are replaced by the corresponding HOL connectives $\longrightarrow$ and $=$. This is merely a technicality. Then the Boolean constants True and False are eliminated from ϕ, as are implication, $\longrightarrow$, and equivalence, $=$. The only remaining connectives are conjunction, disjunction, and negation. Finally ϕ is converted into negation normal form, and then into conjunctive normal form (CNF). Two different CNF conversions are currently implemented in Isabelle/HOL: a naive encoding that may cause an exponential blowup of the formula, and a Tseitin-style encoding [162] that may introduce (existentially quantified) auxiliary Boolean variables, cf. [62]. Quantified subformulae of ϕ are treated as atomic.

Note that it is not sufficient to convert ϕ into an equivalent formula ϕ^* in CNF. Rather, we have to *prove* this equivalence inside Isabelle/HOL. The result is not a single formula, but a theorem of the form $\phi = \phi^*$.

The fact that our CNF transformation must be proof-producing leaves some potential for optimization. One could implement a non proof-producing (and therefore much faster) version of the same CNF transformation, and use it for preprocessing instead. Application of the proof-producing version would then be necessary only if the SAT solver has shown a formula to be unsatisfiable. This scheme can be implemented using lazy proofs [6], thus avoiding the penalty for doing the conversion twice: first without, and later with proofs. This way, preprocessing times for unprovable formulae would improve. In [168] we discuss further ideas to speed up the CNF transformation. The benchmarks used to evaluate the performance of proof reconstruction in Section 5.4 however are already given in conjunctive normal form, so the CNF transformation does not affect the timings reported there.

Unless one of the premises is already syntactically equal to False after CNF transformation

```
datatype prop_formula =
    True
  | False
  | BoolVar of int
  | Not of prop_formula
  | Or of prop_formula * prop_formula
  | And of prop_formula * prop_formula
```

Figure 5.2: SML datatype of propositional formulae

(in which case we can prove the conjecture outright), the non-trivial premises are then written to a file in DIMACS CNF format. A clause is *trivial* if it is syntactically equal to True, or if it contains both an atom and the atom's negation. Filtering out these clauses is crucial to keep our clause numbering consistent with the one maintained by zChaff: zChaff removes trivial clauses during its own preprocessing, without further notice in its proof trace. (This is a general issue when integrating external provers in a proof-producing fashion. "Simple" preprocessing steps are often not recorded in the proof trace. A common solution in this case is to implement a proof-producing preprocessor which is at least as powerful as the one in the external system, thereby making the external preprocessing essentially redundant.)

An intermediate SML [115] datatype (shown in Figure 5.2) is employed to translate HOL terms into DIMACS format. A formula of propositional logic is either True, False, a Boolean variable (with a numerical index as its name), the negation of a formula, or the disjunction or conjunction, respectively, of a pair of formulae. The translation of HOL terms into this datatype is straightforward: HOL's True and False are translated as their constructor counterparts True and False, HOL's $\neg$, $\vee$ and $\wedge$ are translated as Not, Or and And, respectively. All other terms are considered atomic and replaced by Boolean variables. The translation is parameterized by a table (implemented as a balanced 2-3 tree [173] for logarithmic time insertion and lookup) which maintains a mapping from atomic terms to their corresponding variable index. This table, which is initially empty, is updated every time a new atomic term is encountered. α-equivalent terms are mapped to the same index.

Translation from this intermediate datatype of propositional formulae into DIMACS format is almost trivial. Each Boolean variable is mapped to (a string representation of) its index, logical negation is mapped to unary minus, disjunction simply inserts a space between literals, and individual clauses are separated by "0". We translate a formula to a list of strings, rather than to a single string, since list concatenations are generally faster than concatenations of long strings in today's SML systems. Furthermore, strings in SML have a fixed maximal length, which may not allow us to represent the result of the translation as a single string anyway. Finally a DIMACS problem line [50] is prepended to the list of strings, which is then written to a file in proper DIMACS CNF format. The SAT solver is invoked on this input file.

5.3.2 Proof Reconstruction

When zChaff and MiniSat return "unsatisfiable", they generate a resolution-style proof of unsatisfiability and store the proof in a text file. This happens on the fly, to keep memory free

```
type proof = int list Inttab.table * int
```

Figure 5.3: SML type of resolution proofs

for the SAT algorithm itself. While the precise format of this file differs between the solvers, the essential proof structure is the same. Both SAT solvers use propositional resolution to derive new clauses from existing ones:

$$\frac{P \vee x \qquad Q \vee \neg x}{P \vee Q}$$

It is well-known that this single inference rule is sound and complete for propositional logic [144]. A set of clauses is unsatisfiable iff the empty clause is derivable via resolution. For the purpose of proof reconstruction, we are only interested in the proof returned by the SAT solver, not in the techniques and heuristics that the solver uses internally to find this proof. Therefore the integration of zChaff and MiniSat is quite similar, and further SAT solvers capable of generating resolution-style proofs could be integrated in the same manner.

We assign a unique identifier—a non-negative integer—to each clause of the original CNF formula. Further clauses derived by resolution are assigned unique identifiers by the solver. We are usually interested in the result of a resolution *chain*, where two clauses are resolved, the result is resolved with yet another clause, and so on. Consequently, we define an SML type of propositional resolution proofs (see Figure 5.3) as a pair whose first component is a table mapping integers (to be interpreted as the identifiers of clauses derived by resolution) to lists of integers (to be interpreted as the identifiers of previously derived clauses that are part of the defining resolution chain). The second component of the proof is just the identifier of the empty clause.

This type is intended as an internal format to store the information contained in a resolution proof. There are many restrictions on valid proofs that are not enforced by this type. For example, it does not ensure that its second component indeed denotes the empty clause, that every resolution step is legal, or that there are no circular dependencies between derived clauses. It is only important that every resolution proof can be represented as a value of type proof, not conversely. The proof returned by zChaff or MiniSat is translated into this internal format, and passed to the actual proof reconstruction algorithm. This algorithm will either generate an Isabelle/HOL theorem, or fail in case the proof is invalid. Of course the latter should not happen, unless the SAT solver—or our translation from HOL to DIMACS—contains a bug.

zChaff Proof Traces

The format of the proof trace generated by zChaff has not been documented before (aside from our own presentation in [168]). Therefore we explain it here. We use version 2004.11.15 of zChaff; this version is mostly identical to the more recent version 2007.3.12. See Section 5.3.2 below for a simple example of a proof trace.

The proof file generated by zChaff consists of three sections, the first two of which are optional (but present in any non-trivial proof). A formal definition of its syntax in Extended BNF [79]

```
zChaff proof trace  =  {clause line}, {variable line}, conflict clause;
clause line         =  'CL: ', clause id, ' <= ', clause id list, new line;
variable line       =  'VAR: ', variable id, ' L: ', integer, ' V: ', ('0' | '1'),
                          ' A: ', clause id, ' Lits: ', literal id list, new line;
conflict clause     =  'CONF: ', clause id, ' == ', literal id list, new line;
clause id list      =  clause id, {' ', clause id};
clause id           =  integer;
literal id list     =  literal id, {' ', literal id};
literal id          =  integer;
variable id         =  integer;
```

Figure 5.4: EBNF syntax for zChaff proof traces

is given in Figure 5.4. The first section defines clauses derived from the original problem by resolution. A typical line would be "`CL: 7 <= 2 3 0`", meaning that a new clause, assigned the fresh identifier 7, was derived by resolving clauses 2 and 3, and resolving the result with clause 0. Initial clauses are implicitly assigned identifiers starting from 0, in the order they occur in the DIMACS file. We store the information contained in the first section of the proof file in the table of integer lists that constitutes the first component of our SML proof type.

The second section of the proof file records variable assignments that are implied by the first section, and by other variable assignments. As an example, consider "`VAR: 3 L: 2 V: 0 A: 1 Lits: 4 7`". This line states that variable 3 must be false (i.e. its value must be `0`; "`V: 1`" marks true variables) at decision level 2, the *antecedent* being clause 1. The antecedent is a clause in which every literal except for the one containing the assigned variable must evaluate to false because of earlier variable assignments (or because the antecedent is already a unit clause). The antecedent's literals are given explicitly by zChaff, using an encoding that multiplies each variable by 2 and adds 1 for negative literals. (Thus the variable encoded by a literal n is given by $n \div 2$. The variable occurs positively if $n \bmod 2 = 0$, and negatively if $n \bmod 2 = 1$. Hence "`Lits: 4 7`" corresponds to the clause $x_2 \vee \neg x_3$.) Our internal proof format does not allow us to record variable assignments directly, but we can translate them by observing that they correspond to unit clauses. For each variable assignment in zChaff's trace, a new clause identifier is generated (using the number of clauses derived in the first section as a basis, and the variable itself as offset) and added as a key to the proof's table. The associated chain of clauses begins with the antecedent, and continues with the unit clauses corresponding to the explicitly given literals. We ignore both the value and the level information in zChaff's trace. The former is implicit in the derived unit clause (which contains the variable either positively or negatively), and the latter is implicit in the overall proof structure.

The last section of the proof file consists of a single line which specifies the *conflict clause*, a clause which has only false literals: e.g. "`CONF: 3 == 4 6`" says that clause 3 is the conflict clause. (Literals are encoded the same way as in the second section, so clause 3 would be $x_2 \vee x_3$ in this case.) We translate this line into our internal proof format by generating a new clause identifier i which is added to the proof's table, with the conflict clause itself and the unit clauses for each of its variables forming the chain. Finally, we set the proof's second component to i.

For each resolution, we need to determine the pivot literals (i.e. the literals to be resolved on)

```
MiniSat proof trace  =  {reference line | clause line | delete line}, conflict line;
reference line       =  'R ', clause id, ' <= ', literal id list, new line;
clause line          =  'C ', clause id, ' <= ', clause id,
                           {' ', variable id, ' ', clause id}, new line;
delete line          =  'D ', clause id, new line;
conflict line        =  'X ', clause id, ' ', clause id, new line;
clause id            =  integer;
literal id list      =  literal id, {' ', literal id};
literal id           =  integer;
variable id          =  integer;
```

Figure 5.5: EBNF syntax for MiniSat proof traces

before resolving two clauses. This could be done by directly comparing the two clauses, and searching for a term that occurs both positively and negatively. It turns out to be slightly faster however (and also more robust, since we make fewer assumptions about the actual implementation of clauses in Isabelle) to use our own data structure. With each clause, we associate a table that maps integers—one for each literal in the clause—to the prover term representation of a literal. The table is an inverse of the mapping from literals to integers that was constructed for the translation into DIMACS format, but restricted to the literals that actually occur in a clause. Positive integers are mapped to positive literals (atoms), and negative integers are mapped to negative literals (negated atoms). This way term negation simply corresponds to integer negation. The table associated with the result of a resolution step is the union of the two tables that were associated with the resolvents, but with the entries for the pivots removed.

MiniSat Proof Traces

The proof-logging version of MiniSat generates proof traces in a rather compact (and again undocumented) binary format. This is most likely because SAT competitions currently suggest a limit of 2 GB on proof traces. We use version 1.14p of MiniSat. John Matthews [101] has adapted this version so that it can produce readable proof traces in ASCII format, similar to those produced by zChaff. We describe the precise proof trace format, and its translation into our SML proof type. An Extended BNF syntax definition is shown in Figure 5.5.

MiniSat's proof traces, unlike zChaff's, are not divided into sections. They contain four different types of statements: "`R`" to reference original clauses, "`C`" for clauses derived via resolution, "`D`" to delete clauses that are not needed anymore, and "`X`" to indicate the end of proof. Aside from "`X`", which must appear exactly once and at the end of the proof trace, the other statements may appear in any number and (almost) any order.

MiniSat does not implicitly assign identifiers to clauses in the original CNF formula. Instead, "`R`" statements, e.g. "`R 0 <= -1 3 4`", are used to establish clause identifiers. This particular line introduces a clause identifier 0 for the clause $\neg x_1 \vee x_3 \vee x_4$, which must have been one of the original clauses in this example. (Note that MiniSat, unlike zChaff, uses the DIMACS encoding of literals in its proof trace.) Since our internal proof format uses different identifiers for the original clauses, the translation of MiniSat's proof trace into the internal format becomes

parameterized by a renaming $\mathcal{R}$ of clause identifiers. An "`R`" statement does not affect the proof itself, but it extends the renaming. The given literals are used to look up the identifier of the corresponding original clause, and the clause identifier introduced by the "`R`" statement is mapped to the clause's original (internal) identifier.

New clauses are derived from existing clauses via resolution chains. A typical line would be "`C 7 <= 2 5 3 4 0`", meaning that a new clause with identifier 7 was derived by resolving clauses 2 and 3 (with x_5 as the pivot variable), and resolving the result with clause 0 (with x_4 as the pivot variable). In zChaff's notation, this would correspond to "`CL: 7 <= 2 3 0`". We add this line to the proof's table just like for zChaff, but with one difference: MiniSat's clause identifiers cannot be used directly. Instead, we generate a new internal clause identifier for this line, extend the renaming $\mathcal{R}$ by mapping MiniSat's clause identifier (7 in this example) to the newly generated identifier, and apply $\mathcal{R}$ to the identifiers of resolvents as well.

Clauses that are not needed anymore can be indicated by a "`D`" statement, followed by a clause identifier. Currently we ignore such statements. Making beneficial use of them would require not only a modified proof format, but also a different algorithm for proof reconstruction.

Finally a line like "`X 0 17`" indicates the end of proof. The numbers are the minimum and maximum, respectively, identifiers of clauses used in the proof. We ignore the first identifier (which is usually 0 anyway), and use the second identifier, mapped from MiniSat's identifier scheme to our internal one by applying $\mathcal{R}$, as the identifier of the empty clause, i.e. as the proof's second component.

There is one significant difference between MiniSat's and zChaff's proof traces that should have become apparent from the foregoing description. MiniSat, unlike zChaff, records the pivot variable for each resolution step in its trace, i.e. the variable that occurs positively in one clause partaking in the resolution, and negatively in the other. This information is redundant, as the pivot variable can always be determined from those two clauses: If two clauses containing more than one variable both positively and negatively were to be resolved, the resulting clause would be tautological, i.e. contain a variable and its negation. Both zChaff and MiniSat are smart enough to not derive such tautological clauses in the first place. We have decided to ignore the pivot information in MiniSat's traces, since proof reconstruction for zChaff requires the pivot variable to be determined anyway, and using MiniSat's pivot data would need a modified SML proof type. Hence there is minor potential for optimization wrt. replaying MiniSat proofs in our current implementation.

A Simple Example

We use a small example to illustrate the proof reconstruction. Consider the following input formula

$$\phi \equiv (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3).$$

Since ϕ is already in conjunctive normal form, preprocessing simply yields the theorem $\vdash \phi = \phi$. The corresponding DIMACS CNF file, aside from its header, contains one line for each clause in ϕ:

```
-1  2 0
-2 -3 0
```

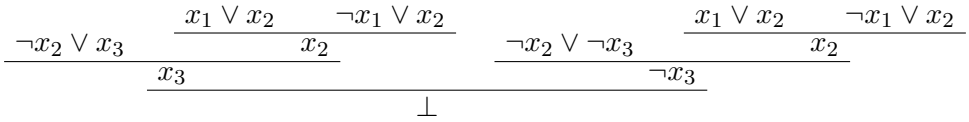


Figure 5.6: Resolution proof found by zChaff

```
 1  2 0
-2  3 0
```

zChaff and MiniSat easily detect that this problem is unsatisfiable. zChaff creates a text file with the following data:

```
CL: 4 <= 2 0
VAR: 2 L: 0 V: 1 A: 4 Lits: 4
VAR: 3 L: 1 V: 0 A: 1 Lits: 5 7
CONF: 3 == 5 6
```

We see that first a new clause, with identifier 4, is derived by resolving clause 2, $x_1 \vee x_2$, with clause 0, $\neg x_1 \vee x_2$. The pivot variable which occurs both positively (in clause 2) and negatively (in clause 0) is x_1; this variable is eliminated by resolution.

Now the value of x_2 (`VAR: 2`) can be deduced from clause 4 (`A: 4`). x_2 must be true (`V: 1`). Clause 4 contains only one literal (`Lits: 4`), namely x_2 (since $4 \div 2 = 2$), occuring positively (since $4 \bmod 2 = 0$)—recall the above section on zChaff proof traces for an explanation of the encoding of literals. This decision is made at level 0 (`L: 0`), before any decision at higher levels.

Likewise, the value of x_3 can then be deduced from clause 1, $\neg x_2 \vee \neg x_3$. x_3 must be false (`V: 0`).

Finally clause 3 is our conflict clause. It contains two literals, $\neg x_2$ (since $5 \div 2 = 2$, $5 \bmod 2 = 1$) and x_3 (since $6 \div 2 = 3$, $6 \bmod 2 = 0$). But we already know that both literals must be false, so this clause is not satisfiable.

In Isabelle, the resolution proof corresponding to zChaff's proof trace is constructed backwards from the conflict clause. A tree-like representation of the proof is shown in Figure 5.6. Note that information concerning the level of decisions, the actual value of variables, or the literals that occur in a clause is redundant in the sense that it is not needed by Isabelle to validate zChaff's proof. The clause x_2, although used twice in the proof, is derived only once during resolution (and reused the second time), saving one resolution step in this example.

The proof trace produced by MiniSat for the same problem happens to encode a different resolution proof:

```
R 0 <= -1 2
R 1 <= -2 -3
R 2 <= 1 2
R 3 <= -2 3
```

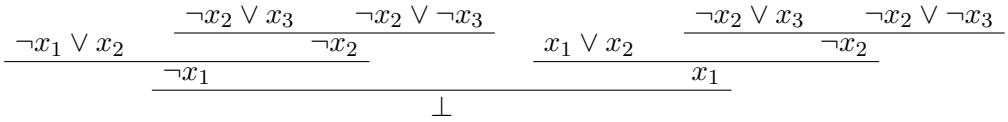


Figure 5.7: Resolution proof found by MiniSat

```
C 4 <= 3 3 1
C 5 <= 0 2 4
C 6 <= 2 2 4
C 7 <= 5 1 6
X 0 7
```

The first four lines introduce clause identifiers for all four clauses in the original problem, in their original order as well (effectively making the renaming $\mathcal{R}$ from MiniSat's clause identifiers to internal clause identifiers the identity in this case). The next four lines define four new clauses (one clause per line), derived by resolution. Clause 4 is the result of resolving clause 3 ($\neg x_2 \vee x_3$) with clause 1 ($\neg x_2 \vee \neg x_3$), where x_3 is used as pivot literal. Hence clause 4 is equal to $\neg x_2$. Likewise, clause 5 is the result of resolving clauses 0 and 4, and clause 6 is obtained by resolving clauses 2 and 4. Finally resolving clauses 5 and 6 yields the empty clause, which is assigned clause identifier 7. The full proof is shown in Figure 5.7.

Proof Trace Compaction

Before proof reconstruction begins, we can remove redundant and unused information from the proof trace. This can be done without proof, saving time.

An obvious optimization is the removal of unused clauses. During proof search, the SAT solver may derive many clauses that are never used to derive the empty clause. Since the proof is logged to file on the fly, these derivations end up in the final proof trace. Instead of replaying the whole proof trace in chronological order, we perform "backwards" proof reconstruction, starting with the identifier of the empty clause, and recursively proving only the required resolvents using depth-first search.

While some clauses may not be needed at all, others may be used multiple times in the resolution proof. It would be inefficient to prove these clauses more than once. Therefore all clauses are stored in an associative array, keyed on their clause identifier, and upon first use converted into the sequent representation described in Section 5.3.3 below. Reusing a clause merely causes an array lookup.

This suggests that it could be beneficial to analyze resolution chains in more detail. Often very similar chains occur in a proof, differing only in a clause or two. Common parts of resolution chains could be stored as additional lemmas (which need to be derived only once), thereby reducing the total number of resolution steps. Hasam Amjad reports on some preliminary results in [7], but we leave a detailed evaluation of this idea for future work.

5.3.3 Clause Representations

The task of proof reconstruction is to derive False from the original clauses, using information from a value of type proof (which represents a resolution proof found by a SAT solver). This can be done in various ways. In particular the precise representation of the problem as an Isabelle/HOL theorem (or a collection of Isabelle/HOL theorems) turns out to be crucial for performance.

Naive HOL Representation

In an early implementation [165], the whole problem was represented as a single theorem $\vdash\ (\phi^* \Longrightarrow \text{False}) \Longrightarrow (\phi^* \Longrightarrow \text{False})$, where ϕ^* was completely encoded in HOL as a conjunction of disjunctions. Step by step, this theorem was then modified to reduce the antecedent $\phi^* \Longrightarrow \text{False}$ to True, which would eventually prove $\vdash \phi^* \Longrightarrow \text{False}$.

This was extremely inefficient for two reasons. First, every resolution step required manipulation of the whole (possibly huge) problem at once. Second, and just as important, SAT solvers treat clauses as *sets* of literals, making implicit use of associativity, commutativity and idempotence of disjunction. Likewise, CNF formulae are treated as sets of clauses, making implicit use of the same properties for conjunction. The encoding in HOL however required numerous explicit rewrites (with theorems like $\vdash (P \vee Q) = (Q \vee P)$) to reorder clauses and literals before each resolution step. Detailed performance figures may be found in [165].

Separate Clauses Representation

A better representation of the CNF formula was discussed in [58]. So far we have mostly considered theorems of the form $\vdash \phi$ in this chapter, i.e. with no hypotheses. This was motivated by the normal user-level view of theorems, where assumptions are encoded using (meta) implication $\Longrightarrow$, rather than hypotheses. Isabelle's inference kernel however provides rules that let us convert between hypotheses and implications as we like:

$$\frac{}{\{\phi\} \vdash \phi}\text{Assume} \qquad \frac{\Gamma \vdash \psi}{\Gamma \setminus \phi \vdash \phi \Longrightarrow \psi}\text{impI} \qquad \frac{\Gamma \vdash \phi \Longrightarrow \psi \quad \Delta \vdash \phi}{\Gamma \cup \Delta \vdash \psi}\text{impE}$$

In [58], each clause $p_1 \vee \cdots \vee p_n$ is encoded as an implication $\overline{p_1} \Longrightarrow \ldots \Longrightarrow \overline{p_n} \Longrightarrow \text{False}$ (where $\overline{p_i}$ denotes the negation normal form of $\neg p_i$, for $1 \leq i \leq n$), and turned into a separate theorem

$$\{p_1 \vee \ldots \vee p_n\} \vdash [\![\overline{p_1}; \ldots; \overline{p_n}]\!] \Longrightarrow \text{False}.$$

This allows resolution to operate on comparatively small objects, and resolving two clauses $\Gamma \vdash [\![p_1; \ldots; p_n]\!] \Longrightarrow \text{False}$ and $\Delta \vdash [\![q_1; \ldots; q_m]\!] \Longrightarrow \text{False}$, where $\neg p_i = q_j$ for some i and j, essentially becomes an application of the cut rule. The first clause is rewritten to $\Gamma \vdash [\![p_1; \ldots; p_{i-1}; p_{i+1}; \ldots; p_n]\!] \Longrightarrow \neg p_i$. A derived tactic then performs the cut to obtain

$$\Gamma \cup \Delta \vdash [\![q_1; \ldots; q_{j-1}; p_1; \ldots; p_{i-1}; p_{i+1}; \ldots; p_n; q_{j+1}; \ldots; q_m]\!] \Longrightarrow \text{False}$$

from the two clauses. Note that this representation, while breaking apart the given clauses into separate theorems allows us to view the CNF formula as a set of clauses, still does not allow

us to view each individual clause as a set of literals. Some reordering of literals is necessary before cuts can be performed, and after each cut, duplicate literals have to be removed from the result.

This representation improved on the proof replay times reported in [165] by up to two orders of magnitude. Detailed numbers are given in [58].

Sequent Representation

We can further exploit the fact that the inference kernel treats a theorem's hypotheses as a set of formulae, by encoding each clause using hypotheses only. Consider the following representation of a clause $p_1 \vee \ldots \vee p_n$ as a theorem:

$$\{p_1 \vee \ldots \vee p_n, \overline{p_1}, \ldots, \overline{p_n}\} \vdash \text{False}.$$

Resolving two clauses $p_1 \vee \ldots \vee p_n$ and $q_1 \vee \ldots \vee q_m$, where $\neg p_i = q_j$, now starts with two applications of the impI rule to obtain theorems

$$\{p_1 \vee \ldots \vee p_n, \overline{p_1}, \ldots, \overline{p_{i-1}}, \overline{p_{i+1}}, \ldots, \overline{p_n}\} \vdash \neg p_i \Longrightarrow \text{False}$$

and

$$\{q_1 \vee \ldots \vee q_m, \overline{q_1}, \ldots, \overline{q_{j-1}}, \overline{q_{j+1}}, \ldots, \overline{q_m}\} \vdash p_i \Longrightarrow \text{False}.$$

We then instantiate a previously proven lemma

$$\vdash (P \Longrightarrow \text{False}) \Longrightarrow (\neg P \Longrightarrow \text{False}) \Longrightarrow \text{False}$$

(where P is an arbitrary proposition) with p_i for P. Instantiation is another basic operation provided by Isabelle's inference kernel. Finally two applications of impE yield

$$\{p_1 \vee \ldots \vee p_n, \overline{p_1}, \ldots, \overline{p_{i-1}}, \overline{p_{i+1}}, \ldots, \overline{p_n}\} \cup \{q_1 \vee \ldots \vee q_m, \overline{q_1}, \ldots, \overline{q_{j-1}}, \overline{q_{j+1}}, \ldots, \overline{q_m}\} \vdash \text{False}.$$

This approach requires no explicit reordering of literals anymore, nor removal of duplicate literals after resolution. That is all handled by the inference kernel now, which treats a theorem's hypotheses as a set of formulae (implemented as an ordered list internally). The sequent representation is as close to a SAT solver's view of clauses as sets of literals as is possible in Isabelle/HOL. With this representation, we do not rely on derived rules to perform resolution, but we gave a precise specification in terms of a few inference rules of natural deduction.

CNF Sequent Representation

The sequent representation has the disadvantage that each clause contains itself as a hypothesis. Since hypotheses are accumulated during resolution, this leads to larger and larger sets of hypotheses, which will eventually contain every clause used in the resolution proof as an individual term. Forming the union of these sets takes the kernel a significant amount of time.

It is therefore faster to use a slightly different clause representation, where each clause contains the whole CNF formula ϕ^* as a hypothesis. Let $\phi^* \equiv \bigwedge_{i=1}^{k} C_i$, where k is the number of clauses. Using the Assume rule, we obtain a theorem $\{\bigwedge_{i=1}^{k} C_i\} \vdash \bigwedge_{i=1}^{k} C_i$. Repeated elimination of

conjunction yields a list of theorems $\{\bigwedge_{i=1}^{k} C_i\} \vdash C_1, \ldots, \{\bigwedge_{i=1}^{k} C_i\} \vdash C_k$. Each of these theorems is then converted into the sequent form described above, with literals as hypotheses and False as the theorem's conclusion. Now, throughout the entire proof, the set of hypotheses for each clause consists of a single term $\bigwedge_{i=1}^{k} C_i$ and the clause's literals only. It is therefore much smaller than before, which speeds up resolution.

Furthermore, memory requirements do *not* increase significantly: the term $\bigwedge_{i=1}^{k} C_i$ needs to be kept in memory only once, and can be shared between different clauses. This can also be exploited when the union of hypotheses is formed (assuming that the inference kernel and the underlying SML system support it): a simple pointer comparison is sufficient to determine that both theorems contain $\bigwedge_{i=1}^{k} C_i$ as a hypothesis (and hence that the resulting theorem needs to contain it only once); no lengthy term traversal is required. Thus, even though the size of the sequent using this representation increases in terms of the number of symbols, there is no detrimental effect on either performance or memory use.

We should mention that this representation of clauses, despite its superior performance, has a small downside. The resulting theorem always has *every* given clause as a premise, while the theorem produced by the sequent representation only has those clauses as premises that were actually used in the proof. To obtain the latter, logically stronger theorem, the resolution proof can be analyzed to identify the clauses that are used in the proof, and the unused ones can be filtered out *before* proof reconstruction.

5.4 Evaluation

Isabelle/HOL offers three major automatic proof tactics: *auto*, which performs simplification and splitting of a goal, *blast* [136], a tableau-based prover, and *fast*, which searches for a proof using standard Isabelle inference. Details can be found in [125]. In [165], we compared the performance of proof reconstruction with the naive HOL representation to that of Isabelle's existing proof procedures. As benchmarks we used all 42 problems contained in version 2.6.0 of the TPTP library [155] that have a representation in propositional logic. The problems were negated, so that unsatisfiable problems became provable.

We found that 19 of these 42 problems are rather easy, and were solved in less than a second each by both the existing tactics and the SAT solver approach. Table 5.1 shows the times (in seconds) that Isabelle's procedures and our first implementation—using the naive HOL encoding—required to solve the remaining 23 problems. These timings were obtained on a machine with a 3 GHz Intel Xeon CPU and 1 GB of main memory. An **x** indicates that the procedure ran out of memory or failed to terminate within an hour. The timings in the SAT column include preprocessing time, zChaff solving time, and proof reconstruction in Isabelle.

None of the existing tactics could prove more than 7 of the 16 unsatisfiable problems with the given time and memory constraints. The SAT solver approach however solved all, and only the first problem, MSC007-1.008, took a significant amount of time to prove in Isabelle. Moreover, all existing tactics timed out on the 7 satisfiable problems (failing to notice that their negation is unprovable), while zChaff quickly provided counterexamples for each of them. We conclude that on propositional problems, already our first, rather inefficient implementation of SAT proof reconstruction was clearly superior to Isabelle's built-in proof procedures.

This has become even more obvious with the new sequent representations that were discussed

Problem	Status	auto	blast	fast	SAT
MSC007-1.008	unsat.	x	x	x	726.5
NUM285-1	sat.	x	x	x	0.2
PUZ013-1	unsat.	0.5	x	5.0	0.1
PUZ014-1	unsat.	1.4	x	6.1	0.1
PUZ015-2.006	unsat.	x	x	x	10.5
PUZ016-2.004	sat.	x	x	x	0.3
PUZ016-2.005	unsat.	x	x	x	1.6
PUZ030-2	unsat.	x	x	x	0.7
PUZ033-1	unsat.	0.2	6.4	0.1	0.1
SYN001-1.005	unsat.	x	x	x	0.4
SYN003-1.006	unsat.	0.9	x	1.6	0.1
SYN004-1.007	unsat.	0.3	822.2	2.8	0.1
SYN010-1.005.005	unsat.	x	x	x	0.4
SYN086-1.003	sat.	x	x	x	0.1
SYN087-1.003	sat.	x	x	x	0.1
SYN090-1.008	unsat.	13.8	x	x	0.5
SYN091-1.003	sat.	x	x	x	0.1
SYN092-1.003	sat.	x	x	x	0.1
SYN093-1.002	unsat.	1290.8	16.2	1126.6	0.1
SYN094-1.005	unsat.	x	x	x	0.8
SYN097-1.002	unsat.	x	19.2	x	0.2
SYN098-1.002	unsat.	x	x	x	0.4
SYN302-1.003	sat.	x	x	x	0.4

Table 5.1: Runtimes (in seconds) for TPTP problems, naive HOL representation

Problem Representation	SAT
Naive HOL	726.5
Separate Clauses	7.8
Sequent	1.2
CNF Sequent	0.7

Table 5.2: Runtimes (in seconds) for MSC007-1.008

above. To give an impression of the effect that the different clause representations have on performance, Table 5.2 shows the different times required to prove problem MSC007-1.008 in Isabelle. The proof found by zChaff for this problem has 8,705 resolution steps. (MiniSat finds a proof with 40,790 resolution steps for the same problem, which is reconstructed in about 3.8 seconds total with the sequent representation, and in 1.9 seconds total with the CNF sequent representation.) The times to prove the other problems from Table 5.1 have decreased in a similar fashion and are well below one second each now.

This enables us to evaluate the performance on some significantly larger problems, such as pigeonhole instances and industrial problems taken from the SATLIB [73] library. These problems do not only push Isabelle's inference kernel to its limits, but also other components of the prover; in particular its term parser and pretty-printer. The TPTP problems were converted to Isabelle's input syntax by a Perl [164] script. This turned out to be infeasible for the larger SATLIB problems. The script still works fine for these problems, but Isabelle's parser (which is mainly intended for small, hand-crafted terms) is unable to parse the resulting theory files, which are several megabytes large, in reasonable time. Also, the prover's user interface

Problem	Variables	Clauses	Resolutions	zChaff	zChaff+	zverify_df	Isabelle
c7552mul.miter	11282	69529	242509	45	45	1.1	69
6pipe	15800	394739	310813	134	137	3.7	192
6pipe_6_ooo	17064	545612	782903	263	265	5.1	421
7pipe	23910	751118	497019	440	440	6.5	609

Table 5.3: Runtimes (in seconds) for SATLIB problems, CNF sequent representation

Problem	Variables	Clauses	Resolutions	zChaff	zChaff+	zverify_df	Isabelle
pigeon-7	56	204	8705	< 1	< 1	< 0.1	< 1
pigeon-8	72	297	25369	< 1	< 1	0.1	1
pigeon-9	90	415	73472	1	1	0.2	3
pigeon-10	110	561	215718	5	6	0.4	10
pigeon-11	132	738	601745	23	24	1.2	36
pigeon-12	156	949	3186775	242	247	6.5	315

Table 5.4: Runtimes (in seconds) for pigeonhole instances, CNF sequent representation

is unable to display the resulting formulae. We have therefore implemented our own parser, which builds SML terms directly from DIMACS files, and we work entirely at the system's SML level, avoiding the usual user interface, to prove unsatisfiability.

Statistics for four SATLIB problems (chosen from those that were used to evaluate zChaff's performance in [179]) are shown in Table 5.3. Runtimes for selected pigeonhole instances are given in Table 5.4. The time for zChaff is time taken to solve the problem, without (zChaff) and with (zChaff+) proof logging. (Note that we measure CPU time only, which does not include time spent blocked on I/O. Measuring wall time is pointless because of other processes that may be running simultaneously.) The times reported for Isabelle are total times again, including zChaff solving time, proof replay, parsing of input and output files, and any other intermediate pre- and post-processing. These timings were obtained on a 1.87 GHz Pentium M notebook with 1.5 GB of main memory. Timings are rounded to the nearest second. For comparison, runtimes for zChaff's own proof checker zverify_df [179] are shown as well, rounded to the nearest tenth of a second.[1]

The proof logging version of MiniSat 1.14 ran out of memory on all problems in Table 5.3 except c7552mul.miter. This is probably because MiniSat 1.14 tends to find longer proofs than zChaff, which becomes costly when proof logging is enabled. The latest version of MiniSat often performs better than zChaff (considering the results of the 2006 SAT-Race competition [54]), but unfortunately it did not support proof logging at the time of writing. Therefore we do not give performance data for replaying MiniSat proofs.

Needless to say, none of the SATLIB problems can be solved automatically by Isabelle's built-in tactics. Only the smallest of the pigeonhole instances succumbs, and takes far longer to do so. Pigeonhole instances are known to be pathologically hard problems for resolution proof systems [67]. Isabelle ran out of memory on the pigeonhole problem with 13 holes, even though zChaff found a proof in about 10 minutes. It is hard to do a fine-grained memory analysis,

[1] The version of zevrify_df that comes with zChaff 2004.11.15 contains a minor bug (related to the decision level of variables) which increases its runtime significantly. The above timings were measured with the bug fixed. Our bug fix has been incorporated into the 2007.3.12 release of zChaff.

but we can safely say that having to store terms rather than numbers in memory contributed to this failure.

Proof checking in Isabelle/HOL, despite all optimizations that we have implemented, is about an order of magnitude slower than proof verification with zChaff's own proof checker zverify_df, written in C++. This additional overhead is to be expected; it is the price that we have to pay for using an LCF-style theorem prover for higher-order logic, whose inference kernel is not geared towards propositional logic. However, we also see that proof reconstruction in Isabelle scales quite well with our latest implementation, and that it remains feasible even for large SAT problems.

In [168], we report timings for proof reconstruction in HOL 4 and HOL Light. While comparing these values directly is of limited significance (because of fundamental differences like the underlying SML system and the kernels' implementation of theorems), it is still worth noting that our Isabelle/HOL implementation performs up to an order of magnitude better than the (conceptually similar) implementations of proof reconstruction in those provers.

5.5 Persistent Theorems

We have seen that SAT solvers can greatly enhance Isabelle's abilities. The same holds true for other automated theorem provers, e.g. for first-order logic [113] or Satisfiability Modulo Theories (SMT) [77]. Verification of proof scripts becomes more difficult however. A user who wants to verify an Isabelle proof script that invokes external tools not only has to install Isabelle itself, but also those additional programs. Essentially, verification of proof scripts is restricted to systems which have a very similar configuration, and setting up a system so that proof scripts which rely on external tools become verifiable can be a cumbersome task.

Proof terms, which were implemented for Isabelle by Stefan Berghofer [23], suggest a simple solution to make the verification of proof scripts more independent of the presence and configuration of external tools. Proof terms encode the proof of a theorem in terms of primitive inference rules. They can be significantly larger than proof scripts (since the invocation of a powerful automated tactic from a script may result in a large number of primitive inferences—our SAT-solver based tactic is a good example of this). But while turning proof scripts into theorems requires not only full Isabelle, but potentially also a number of external tools, proof terms on the other hand can be verified by a very simple proof checker, which comprises a few dozen lines of code only.

We have implemented a *tactical* (a function which transforms proof tactics) in SML that turns a given tactic, say t, into one that potentially uses proof terms. Our implementation proceeds as follows. Given t's input theorem ϕ, we first attempt to read a proof term that proves the conclusion of ϕ from a file on disk. A 32-bit hash function (taken from [146]) is used to compute the fixed-length file name. Chaining is employed to deal with potential hash collisions. We have applied the hash function to Isabelle's entire theorem library however, and there are no hash collisions in the library at the time of writing.

If reading the proof term (and turning it into a theorem that proves the conclusion of ϕ, using the simple proof checker that is integrated with Isabelle) succeeds, we are done. In this case we can simply return the theorem, with no need to apply the original tactic t. This is the case that allows verification of proof scripts by the core Isabelle system (and in fact by the proof

Figure 5.8: Persistent theorems

checker component only), if the necessary proof terms are given together with the proof script.

Reading the proof term from disk may fail for several reasons. Perhaps the file does not exist, or it does not contain a valid proof, or the term in the file proves a different theorem. In this case we apply the tactic t (which may call external provers) to the given input theorem. This yields a sequence of successor theorems, of which we only consider the first. (We can save proof terms for a single theorem to disk, but not for a possibly infinite sequence of theorems—at least not easily.) This theorem's proof term is stored in the aforementioned file (as an XML [28] document to simplify parsing and enhance portability), so that future invocations of the adapted tactic can retrieve it from there. In addition, the theorem is returned as the tactic's result.

A flowchart depicting the algorithm and the different data formats involved is given in Fig-

ure 5.8. To subsume, this approach allows us to easily turn a tactic which may require a number of external tools into one that attempts to read a proof term from a file first, and performs the original computation only if this fails. The same tactic can be used both to produce the proof terms when they are not present (provided the required external tools are available), and to verify them when they are. Proof scripts that use such a tactic can be verified on systems where the necessary external provers are configured, and also on systems where they aren't, if the proof term files are available.

5.6 Conclusion

The SAT solver approach dramatically outperforms the automatic procedures that were previously available in Isabelle/HOL. With the help of MiniSat or zChaff, many formulae that were previously out of the scope of built-in tactics can now be proved—or refuted—automatically, often within seconds. Isabelle's applicability as a tool for formal verification, where large propositional problems occur in practice, has thereby improved considerably.

Furthermore, using the data structures and optimizations described in this chapter, proof reconstruction for propositional logic scales quite well even to large SAT problems and proofs with millions of resolution steps. The additional confidence gained by using an LCF-style prover to check the proof obviously comes at a price (in terms of running time), but it's not nearly as expensive as one might have expected after earlier implementations.

While improving the performance of proof reconstruction, we discovered inefficiencies in the implementation of the Isabelle kernel. Subsequently the prover implementation was modified, and these inefficiencies were removed. Tuning an implementation to the extend presented here requires a great deal of familiarity with the underlying theorem prover. Nevertheless our results are applicable beyond Isabelle/HOL, really to any prover that supports propositional logic and is able to simulate propositional resolution.

We did not find any soundness bugs in the SAT solvers during proof reconstruction. This is not surprising, since the solvers had already been tested thoroughly on all the problems evaluated above. We did note an odd completeness bug in the verifier bundled with zChaff, which refuses to verify a proof of unsatisfiability if the original problem contains trivial clauses. Definitional CVF conversions often generate trivial clauses, so in our setting this is perhaps more important than for the usual verification of unsatisfiable SATLIB problems.

Regarding the proofs produced by SAT solvers, we would like to emphasize the importance of having a well-documented standard, similar to what the DIMACS format is for a SAT solver's input. At present, the mere fact that different solvers use different (and partially undocumented) proof formats makes their integration a bit more of an engineering challenge than it would have to be. Also, solver developers need to be aware that even trivial preprocessing steps (like reordering of clauses) may need to be reproduced in the proof checker. Therefore these steps should (perhaps optionally) be logged in the proof trace as well, or the checker must implement the same preprocessing algorithm as the solver.

We have already mentioned some possible directions for future work. There is probably not very much potential left to optimize the implementation of resolution itself at this point. However, to further improve the performance of proof reconstruction, it could be beneficial to analyze the resolution proof found by the SAT solver in more detail. Merging similar resolution chains may

reduce the overall number of resolutions required, and re-sorting resolutions may help to derive shorter clauses during the proof, which should improve the performance of individual resolution steps. Some preliminary results along these lines are reported in [7]. Also preprocessing of CNF formulae for SAT solvers has recently shown very promising results [52, 8], so it might be worthwhile to integrate a preprocessing SAT solver with an LCF-style prover. Note that this is not a trivial task, as the preprocessing must be mimicked inside the HOL prover in a proof-producing fashion.

The approach presented in this chapter has applications beyond propositional reasoning. The decision problem for richer logics (or fragments thereof) can be reduced to SAT [11, 154, 110, 139]. Consequently, proof reconstruction for propositional logic can serve as a foundation for proof reconstruction for other logics. Based on our work, only a proof-generating implementation of the reduction is needed to integrate the more powerful, yet SAT-based decision procedure with an LCF-style theorem prover. This has already been used to integrate the SMT solver haRVey with Isabelle [76, 77]. haRVey, like other SMT systems, uses various decision procedures (e.g. congruence closure for uninterpreted functions) on top of a SAT solver.

Every solution breeds new problems.
Arthur Bloch, born 1948.

Chapter 6

Conclusion

This chapter summarizes the results presented in this thesis, and gives directions for possible future work.

6.1 Summary

In this thesis, we have presented a finite model generation algorithm for higher-order logic. The main theoretical contribution is a correctness proof for the underlying translation from higher-order logic to propositional logic. On the practical side, we have achieved a seamless integration of the model generator with Isabelle/HOL. In particular its support for many specification techniques available in this logic, including datatypes, recursive functions, type classes, records, set types, etc., makes the model generator applicable to a wide class of conjectures stated in the theorem prover. If a counterexample is found, it is displayed to the user, potentially saving a significant amount of time otherwise spent on fruitless proof attempts.

The successful application of the model generator to three case studies, namely to obtain a correctness proof for an abstract version of the RSA-PSS security protocol, counterexamples to conjectures about probabilistic programs, and a Sudoku solver, shows that the algorithm is of practical utility. For the second case study, an abstract model of probabilistic programs was developed that is susceptible to counterexample search via finite model generation, thereby contributing to the theory of probabilistic programs.

The LCF-style integration of zChaff and MiniSat with Isabelle/HOL that has been presented in Chapter 5 shows that an interactive theorem prover for higher-order logic can serve as a viable proof checker for propositional resolution proofs with millions of proof steps. Our optimization techniques are applicable also to other higher-order logic theorem provers, e.g. to HOL 4 and HOL Light. The use of state-of-the-art SAT solvers has greatly improved Isabelle's performance on propositional problems, thereby enhancing its applicability for hardware and

software verification, where many problems can be encoded in propositional logic. A prototype implementation of persistent proofs makes the verification of proof scripts indepent of external tools; this can facilitate the exchange of proof scripts between different Isabelle installations.

6.2 Future Work

To conclude, we give directions for possible future work. Some of the following research questions arose directly from the work presented in this thesis, while others are linked to alternative approaches that we did not investigate in detail.

Integration with Isabelle. The model generator that was presented in this thesis has been integrated with Isabelle/HOL, and it supports various definitional mechanisms and extensions that this logic offers, most notably recursive datatypes. Isabelle/HOL continues to evolve however, and support for some of its features is currently lacking or incomplete. In particular the model generator is not yet *context-aware*. This applies both to theory contexts (called *locales* in Isabelle [13]) and proof contexts [14], which permit e.g. local definitions in proofs.

The model generator could also be integrated with Isabelle's meta logic, Isabelle/Pure (see Section 3.3), and consequently be made available for other object logics, e.g. Zermelo-Fraenkel set theory.

Both issues are mainly software engineering tasks. The model generator, due to its tight integration with Isabelle, inevitably depends on the internal interfaces of some of Isabelle/HOL's packages. It will therefore continue to evolve as these interfaces change over time.

Optimizations. Our focus has been the integration of the model generator with Isabelle, in particular with Isabelle/HOL. This allows the model generator to be applied to a wide class of formulae, and its performance is sufficient for interesting case studies. While we have implemented some optimizations (see Section 3.2), further work is necessary to obtain a tool whose performance is generally competitive to that of existing (first-order) model generators. For first-order logic, techniques have been developed to bound or estimate the size of the model [138], to reduce the number of Boolean variables, to reuse search information between consecutive model sizes, or to perform symmetry reduction in order to reduce the number of isomorphic models [41, 156]. It should be a worthwhile research project to transfer these techniques to higher-order logic.

External model generators. An orthogonal approach that would be interesting to evaluate, both in terms of performance and feasibility, is the use of external (first-order) model generators. The necessary translation from HOL to first-order logic could be based on recent work by Meng and Paulson [113] (which would need to be adapted however, as it targets automated theorem provers, not model generators). The integration of external provers with interactive proof assistants has been pursued for a long time, and the integration of external model generators could have similar benefits. Most notably, Isabelle could profit directly from advanced, highly efficient algorithms implemented in external tools.

Other methods of disproving. Aside from finite model generation, various other techniques exist that can be used to refute false conjectures. Berghofer has integrated *quickcheck*, a tool based on random testing, with Isabelle [24]. The performance of *quickcheck* is sometimes superior to finite model generation, but the tool is limited to an executable fragment of HOL. It might be possible to combine *quickcheck* with finite model generation to obtain the best of both worlds: an efficient tool that is applicable to a wide class of HOL formulae.

Future work could also focus on the generation of counterexamples from failed proof attempts. Isabelle contains built-in decision procedures for various fragments of HOL, e.g. quantifier elimination algorithms for dense linear orders, real and integer linear arithmetic [124]. While some of these procedures output a (possibly spurious) counterexample when they cannot find a proof, counterexample generation was generally not of high priority in their development until now. Notably *blast*, Isabelle's built-in tableau prover [136], currently does not output any useful information when it fails.

Methods for the generation of infinite models (see [90] for a survey) could be used to refute formulae that have no finite countermodels. These techniques could again be implemented as part of Isabelle, or in an external tool that is then integrated with Isabelle.

SAT solving. Both the MACE-style algorithm for finite model generation and the resolution proof reconstruction presented in this thesis crucially depend on efficient SAT solvers. Our model generator supports various state-of-the-art SAT solvers, including zChaff and MiniSat. The integration of other SAT solvers that use the DIMACS input format is straightforward. However, for convenience we have also implemented our own SAT solver in Standard ML. This simple, DPLL-based solver is now part of the Isabelle system. It would be interesting to see if the performance of an optimized SAT solver written in Standard ML can be competitive to that of one of the currently leading solvers (which are usually written in C or C++), and it would be useful to extend Isabelle's own SAT solver with the ability to generate unsatisfiability proofs.

Regarding the proof-producing integration of SAT solvers with LCF-style theorem provers, we have already mentioned several directions for future work in Section 5.6. Our integration can serve as a foundation for the integration of proof-generating decision procedures for richer logics that are based on SAT, e.g. satisfiability modulo theories (SMT), or fragments of first-order logic. This has already been used to integrate the SMT solver haRVey with Isabelle [77].

In Chapter 5 we have focused on an efficient implementation of proof reconstruction, where the resolution proof was found by a SAT solver. Another promising line of research is concerned with obtaining an efficient (compressed) representation of the proof. One can sometimes merge similar resolution chains and re-sort certain resolution steps to obtain a shorter proof; see e.g. [7] for recent work.

An alternative to LCF-style proof checking is *reflection*, a technique where an algorithm is proved correct in the theorem prover, and then code is generated from the algorithm's definition that produces trusted results [35]. Reflection offers good performance (as the reflected code does not need to produce LCF-style proofs) and high reliability (as the trusted code base must include the code generator, but not any reflected proof procedure). It would be interesting to compare our LCF-style proof checker to a reflection-based approach. Some initial results were recently given in [31].

Formalization. Most theorems in this thesis were proved by traditional "pen-and-paper" proofs only. We have focused on extending Isabelle, rather than on using it to establish fully formal theorems. It seems natural however to suggest a machine-readable formalization of our results. This should be a straightforward (albeit laborious) task for the results about the model of probabilistic programs presented in Section 4.3. A formalization of the model generator's correctness (Theorem 2.100) on the other hand would be technically challenging, because the set-theoretic semantics of higher-order logic cannot be defined in Isabelle/HOL directly. HOL-ST, an extension of Isabelle/HOL proposed by Agerholm [3, 4] and Gordon [65], could be a suitable framework for such a formalization. HOL-ST, which was later called HOLZF by Obua [128], adds a type V (for the set-theoretic universe) and a function $\in\colon V \times V \to \mathsf{bool}$ (for set membership) to HOL. Then the usual axioms of ZF set theory are asserted.

List of Figures

List of Tables

Bibliography

[1] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.

[2] Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[3] Sten Agerholm. Formalising a model of the λ-calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, 1994.

[4] Sten Agerholm and Michael J. C. Gordon. Experiments with ZF set theory in HOL and Isabelle. In E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications – 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*, volume 971 of *Lecture Notes in Computer Science*, pages 32–45. Springer, 1995.

[5] Col Allan, editor. *New York Post*. News Corporation, New York City, NY, USA, 2005.

[6] Hasan Amjad. Shallow lazy proofs. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics – 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2005.

[7] Hasan Amjad. Compressing propositional refutations. In Stephan Merz and Tobias Nipkow, editors, *Proceedings of the 6th International Workshop on Automated Verification of Critical Systems (AVoCS 2006)*, volume 185 of *Electronic Notes in Theoretical Computer Science*, pages 3–15. Elsevier, July 2007.

[8] Anbulagan and John Slaney. Multiple preprocessing for systematic SAT solvers. In C. Benzmüller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on the Implementation of Logics*, volume 212 of *CEUR Workshop Proceedings*, pages 100–116, Phnom Penh, Cambodia, 2006.

[9] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "Small Scope Hypothesis", September 2002. Available from `http://sdg.csail.mit.edu/pubs/2002/SSH.pdf`.

[10] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, second edition, July 2002.

[11] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 195–210, Copenhagen, Denmark, July 2002. Springer.

[12] Serge Autexier and Carsten Schürmann. Disproving false conjectures. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings*, volume 2850 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2003.

[13] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 – May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[14] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11-12, 2006, Proceedings*, volume 4108 of *Lecture Notes in Artificial Intelligence*, pages 31–43. Springer, 2006.

[15] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, Boston, Massachusetts, USA, July 2004.

[16] Clark Barrett, Sergey Berezin, and David L. Dill. A proof-producing Boolean search engine. In *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2003)*, Miami, Florida, USA, July 2003.

[17] David Basin and Stefan Friedrich. Combining WS1S and HOL. In Dov M. Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[18] David Basin, Séan Matthews, and Luca Viganò. A modular presentation of modal logics in a logical framework. In *Isabelle Users Workshop – Cambridge, England, September 18-19, 1995, Proceedings*, September 1995.

[19] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, September 2004.

[20] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology – EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 1996.

[21] Belaid Benhamou and Laurent Henocque. Finite model search for equational theories (FMSET). In Jacques Calmet and Jan Plaza, editors, *Artificial Intelligence and Symbolic Computation, International Conference, AISC'98, Plattsburg, New York, USA, September 1618, 1998, Proceedings*, volume 1476 of *Lecture Notes in Artificial Intelligence*, pages 84–93. Springer, 1998.

[22] Stefan Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL. Master's thesis, Institut für Informatik, Technische Universität München, 1998.

[23] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics – 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 2000.

[24] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 230–239. IEEE Computer Society, 2004. Invited paper.

[25] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL – lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1999.

[26] Paul Bernays and Moses Schönfinkel. Zum Entscheidungsproblem der mathematischen Logik. *Mathematische Annalen*, 99(1):342–372, 1928.

[27] Daniel Le Berre and Laurent Simon, editors. *The SAT 2005 competitions and evaluations*, volume 2 of *Journal on Satisfiability, Boolean Modeling and Computation*. IOS Press, 2005.

[28] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation 16 August 2006, edited in place 29 September 2006*, 2006. Available from `http://www.w3.org/TR/2006/REC-xml11-20060816/`.

[29] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing – 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2005.

[30] Achim D. Brucker and Burkhart Wolff. Interactive testing with HOL-TestGen. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Software Testing – 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2006.

[31] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. To appear at TPHOLs 2008.

[32] Georg Cantor. Über eine elementare Frage der Mannigfaltigkeitslehre. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 1:75–78, 1891.

[33] Constantin Carathéodory. Über den Variabilitätsbereich der Fourierschen Konstanten von positiven harmonischen Funktionen. *Rendiconti del Circolo Matematico di Palermo*, 32:193–217, 1911.

[34] Serenella Cerrito and Marta Cialdea Mayer. Using linear temporal logic to model and solve planning problems. In Fausto Giunchiglia, editor, *Artificial Intelligence: Methodology, Systems, and Applications, 8th International Conference, AIMSA'98, Sozopol, Bulgaria, September 2123, 1998, Proceedings*, volume 1480 of *Lecture Notes in Artificial Intelligence*, pages 141–152. Springer, 1998.

[35] Amine Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra.* PhD thesis, Institut für Informatik, Technische Universität München, Germany, January 2008. Preliminary version, submitted.

[36] Amine Chaieb and Tobias Nipkow. Verifying and reflecting quantifier elimination for Presburger arithmetic. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 367–380. Springer, 2005.

[37] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

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

[39] Koen Claessen. Equinox, a new theorem prover for full first-order logic with equality. Presentation at Dagstuhl Seminar 05431 on Deduction and Applications, October 2005.

[40] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, volume 35(9) of *SIGPLAN Notices*, pages 268–279. ACM, September 2000.

[41] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[42] Ernie Cohen. Separation and reduction. In Roland Carl Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction – 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 2000.

[43] Simon Colton and Alison Pease. The TM system for repairing non-theorems. In Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle, Silvio Ranise, and Cesare Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 87–101. Elsevier, July 2005.

[44] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[45] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[46] Luca de Alfaro and Thomas A. Henzinger. Concurrent omega-regular games. In *15th Annual IEEE Symposium on Logic in Computer Science, 26-29 June 2000, Santa Barbara, California, USA*, pages 141–154. IEEE Computer Society, 2000.

[47] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[48] Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning – Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 303–317, 2006.

[49] Giovanni di Lorenzo, editor. *Die Zeit*. Zeitverlag Gerd Bucerius GmbH & Co. KG, Hamburg, Germany, 2005.

[50] DIMACS satisfiability suggested format, 1993. Available from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc`.

[51] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, 1981.

[52] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – 8th International Conference, SAT 2005, St Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[53] Niklas Eén and Niklas Sörensson. MiniSat-p-v1.14 – A proof-logging version of MiniSat, September 2006. Available from `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`.

[54] Carsten Sinz et al. SAT-Race 2006 – results, August 2006. Available from `http://fmv.jku.at/sat-race-2006/results.html`.

[55] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible Sudoku grids, June 2005. Available from `http://www.shef.ac.uk/~pm1afj/sudoku/`.

[56] The SML/NJ Fellowship. Standard ML of New Jersey, June 2007. Available from `http://www.smlnj.org/`.

[57] Melvin Fitting. Kleene's three valued logics and their children. *Fundamenta Informaticae*, 20(1–3):113–131, 1994.

[58] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[59] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[60] C. Goller, O. Ibens, R. Letz, K. Mayr, M. Moser, J. Schumann, and J. Steinbach. The model elimination provers SETHEO and E-SETHEO. *Journal of Automated Reasoning*, 18(2):237–246, 1997.

[61] M. J. C. Gordon. From LCF to HOL: A short history. In G. Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*. MIT Press, 2000.

[62] M. J. C. Gordon. HolSatLib documentation, version 1.0b, June 2001. Available from `http://www.cl.cam.ac.uk/~mjcg/HolSatLib/HolSatLib.html`.

[63] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[64] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems Series*, pages 49–70. Elsevier, 1994.

[65] Michael J. C. Gordon. Set theory, higher order logic or both? In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics – 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 191–201. Springer, 1996. Invited paper.

[66] Reiner Hähnle. Tableaux and related methods. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier and MIT Press, 2001.

[67] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[68] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.

[69] John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 1996.

[70] Wilfrid Hodges. *Model Theory*. Cambridge University Press, 1993.

[71] Wilfrid Hodges. First-order model theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Center for the Study of Language and Information, Stanford University, Summer 2005. Available from `http://plato.stanford.edu/archives/sum2005/entries/modeltheory-fo/`.

[72] Gerard J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, September 2003.

[73] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000. Available from `http://www.satlib.org/`.

[74] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, Nice, France, September 1999. Springer.

[75] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Copenhagen, Denmark, July 2002. Springer.

[76] Clément Hurlin. Proof reconstruction for first-order logic and set-theoretical constructions. In Stephan Merz and Tobias Nipkow, editors, *Sixth International Workshop on Automated Verification of Critical Systems (AVoCS 2006) – Preliminary Proceedings*, pages 157–162, 2006.

[77] Clément Hurlin, Amine Chaieb, Pascal Fontaine, Stephan Merz, and Tjark Weber. Practical proof reconstruction for first-order logic and set-theoretical constructions. In Lucas Dixon and Moa Johansson, editors, *Proceedings of the Isabelle Workshop 2007*, pages 2–13, Bremen, Germany, July 2007.

[78] Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1003.1c-1995*, 1995.

[79] International Organization for Standardization. *Information technology – Syntactic metalanguage – Extended BNF*, 1996. ISO/IEC 14977:1996(E).

[80] International Organization for Standardization. *Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2*, 2005. ISO/IEC 19501:2005.

[81] Daniel Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, pages 130–139, San Diego, November 2000.

[82] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[83] Paul Jackson and Daniel Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2004, APES Research Group, March 2004.

[84] Thomas Jech. *Set Theory*. Springer Monographs in Mathematics. Springer, 3rd millennium edition, 2003.

[85] Jan Jürjens. Sound methods and effective tools for model-based security engineering with UML. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), May 15-21, 2005, St. Louis, Missouri, USA*, pages 322–331. ACM, 2005.

[86] Jan Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), September 18-22, 2006, Tokyo, Japan*, pages 167–176. IEEE Computer Society, 2006.

[87] Sara Kalvala and Valeria de Paiva. Linear logic in isabelle. In *Isabelle Users Workshop – Cambridge, England, September 18-19, 1995, Proceedings*, September 1995.

[88] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

[89] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[90] Stefan Klingenbeck. *Counter Examples in Semantic Tableaux*. PhD thesis, Institute for Logic, Complexity and Deduction Systems, University of Karlsruhe, Karlsruhe, Germany, 1996.

[91] Donald E. Knuth. Mathematics and computer science: Coping with finiteness. Advances in our ability to compute are bringing us substantially closer to ultimate limitations. *Science*, 194(4271):1235–1242, December 1976.

[92] Karsten Konrad. *Model Generation for Natural Language Interpretation and Analysis*. PhD thesis, Technische Fakultät, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[93] Dexter Kozen. On Kleene algebras and closed semirings. In Branislav Rovan, editor, *Mathematical Foundations of Computer Science 1990 – Banská Bystrica, Czechoslovakia, August 27-31, 1990, Proceedings*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 1990.

[94] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1996.

[95] Gihwon Kwon and Himanshu Jain. Optimized CNF encoding for Sudoku puzzles. In Miki Hermann and Andrei Voronkov, editors, *LPAR-13, The 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings*, November 2006.

[96] Christina Lindenberg and Kai Wirt. SHA1, RSA, PSS and more. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sourceforge.net/entries/RSAPSS.shtml`, May 2005. Formal proof development.

[97] DeadMan's Handle Ltd. Sudoku solver, September 2005. Available from `http://www.sudoku-solver.com/`.

[98] David Matthews. Poly/ML 5.0 release, December 2006. Available from `http://sourceforge.net/project/showfiles.php?group_id=148318&package_id=163589&release_id=470957`.

[99] David Matthews. Poly/ML home page, February 2007. Available from `http://www.polyml.org/`.

[100] David Matthews. The Thread structure and signature, November 2007. Available from `http://www.polyml.org/docs/Threads.html`.

[101] John Matthews. ASCII proof traces for MiniSat. Personal communication, August 2006.

[102] William McCune. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problems. Technical Report ANL/MCS-TM-194, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1994.

[103] William McCune. MACE 2.0 reference manual and guide. Technical Report ANL/MCS-TM-249, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, May 2001.

[104] William McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-264, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 2003.

[105] William McCune. Otter 3.3 reference manual. Technical Report ANL/MCS-TM-263, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 2003.

[106] William McCune. Prover9 manual, version 2008-04a, April 2008. Available from `http://www.cs.unm.edu/~mccune/prover9/manual/2008-04A/`.

[107] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer Monographs in Computer Science. Springer, 2005.

[108] Annabelle McIver and Tjark Weber. Towards automated proof support for probabilistic distributed systems. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 534–548. Springer, December 2005.

[109] Andreas Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level. In David A. McAllester, editor, *Automated Deduction – CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 460–464. Springer, 2000.

[110] Andreas Meier and Volker Sorge. Applying SAT solving in classification of finite algebras. *Journal of Automated Reasoning*, 35(1–3):201–235, October 2005.

[111] Jia Meng. Integration of interactive and automatic provers. In Manuel Carro and Jesus Correas, editors, *Second CologNet Workshop on Implementation Technology for Computational Logic Systems, FME 2003*, September 2003.

[112] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning: Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 372–384. Springer, 2004.

[113] Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 70–80, 2006.

[114] Robin Milner. *A Calculus of Communicating Systems*. Springer, October 1980.

[115] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, May 1997.

[116] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[117] Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, 2nd edition, 1994.

[118] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[119] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, June 2001.

[120] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[121] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics – 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 October 1, 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366. Springer, 1998.

[122] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[123] Tobias Nipkow. Structured proofs in Isar/HOL. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 2002.

[124] Tobias Nipkow. Linear quantifier elimination. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning (IJCAR 2008)*, volume ? of *Lecture Notes in Computer Science*, pages ?–? Springer, 2008.

[125] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[126] Michael Norrish and Konrad Slind. *The HOL System Description*, January 2007. Available from `http://hol.sourceforge.net/documentation.html`.

[127] Steven Obua. Checking conservativity of overloaded definitions in higher-order logic. In Frank Pfenning, editor, *Term Rewriting and Applications – 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2006.

[128] Steven Obua. Partizan games in Isabelle/HOLZF. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *Theoretical Aspects of Computing – ICTAC 2006, Third International Colloquium, Tunis, Tunisia, November 20-24, 2006, Proceedings*, volume 4281 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2006.

[129] Larry Paulson and Markus Wenzel. The Isabelle reference manual, October 2005. Available from `http://isabelle.in.tum.de/dist/Isabelle/doc/ref.pdf`.

[130] Larry Paulson and Markus Wenzel. Isabelle/FOL — first-order logic, October 2005. Available from `http://isabelle.in.tum.de/dist/library/FOL/document.pdf`.

[131] Lawrence C. Paulson. Set theory for verification: I. from foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.

[132] Lawrence C. Paulson. Set theory for verification: II. induction and recursion. Technical Report 312, University of Cambridge Computer Laboratory, 1993.

[133] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Automated Deduction – CADE-12 – 12th International Conference on Automated Deduction, Nancy, France, June 26 July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 1994.

[134] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[135] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[136] Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–83, 1999.

[137] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):332–351, 1999.

[138] Amir Pnueli, Yoav Rodeh, Ofer Strichman, and Michael Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):279–293, 2002.

[139] Erik Reeber and Warren A. Hunt, Jr. A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning – Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 453–467, 2006.

[140] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, Germany, 1994.

[141] Franz Regensburger. HOLCF: Higher order logic of computable functions. In E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications – 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 1995.

[142] R. Rivest, A. Shamir, and L. Adleman. On digital signatures and public key cryptosystems. Technical Report 82, MIT Laboratory for Computer Science, April 1977.

[143] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[144] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Communications of the ACM*, 12(1):23–41, 1965.

[145] RSA Laboratories. *PKCS #1: RSA Cryptography Standard Version 2.1*, June 2002.

[146] Robert Sedgewick. *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison Wesley Professional, 3rd edition, September 1997.

[147] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1995. Available as Technical Report MIT/LCS/TR-676.

[148] Natarajan Shankar. Automated verification using deduction, exploration, and abstraction. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*. Springer, 2003.

[149] John K. Slaney. FINDER: Finite domain enumerator – system description. In Alan Bundy, editor, *Automated Deduction – CADE-12 – 12th International Conference on Automated Deduction, Nancy, France, June 26 July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 798–801. Springer, 1994.

[150] John K. Slaney and Timothy Surendonk. Combining finite model generation with theorem proving: Problems and prospects. In Franz Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems, First International Workshop, FroCos '96, Munich, Germany, March 26-29, 1996, Proceedings*, volume 3 of *Applied Logic Series*, pages 141–155. Kluwer Academic Publishers, 1996.

[151] Neil J. A. Sloane. The on-line encyclopedia of integer sequences: A014221, December 2007. Available from `http://www.research.att.com/~njas/sequences/A014221`.

[152] Graham Steel, Alan Bundy, and Ewen Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2), 2002.

[153] Ernst Steinitz. Bedingt konvergente Reihen und konvexe Systeme, I–IV. *Journal für reine und angewandte Mathematik*, 143:128–175, 1913.

[154] Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design – 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, volume 2517 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2002.

[155] Geoff Sutcliffe and Christian Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. Available from `http://www.cs.miami.edu/~tptp/`.

[156] Tanel Tammet. Finite model building: improvements and comparisons. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[157] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[158] Robert James Thomson, editor. *The Times*. Times Newspapers Ltd., London, UK, 2005.

[159] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[160] B. A. Trachtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950.

[161] G. Tseitin. On the complexity of derivation in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1970.

[162] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation Of Reasoning: Classical Papers On Computational Logic, Vol. II, 1967–1970*, pages 466–483. Springer, 1983. Also in *Structures in Constructive Mathematics and Mathematical Logic Part II*, ed. A. O. Slisenko, 1968, pp. 115–125.

[163] Pete Wake. Sudoku solver by logic, September 2005. Available from `http://www.sudokusolver.co.uk/`.

[164] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly Media, Inc., 3rd edition, July 2000.

[165] Tjark Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. In Joe Hurd, Edward Smith, and Ashish Darbari, editors, *Theorem Proving in Higher Order Logics – 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005, Emerging Trends Proceedings*, pages 180–189, Oxford, UK, August 2005. Oxford University Computing Laboratory, Programming Research Group. Research Report PRG-RR-05-02.

[166] Tjark Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In Chris Benzmüller, Bernd Fischer, and Geoff Sutcliffe, editors, *Proceedings of the 6th International Workshop on the Implementation of Logics*, volume 212 of *CEUR Workshop Proceedings*, pages 44–62, November 2006.

[167] Tjark Weber. Integrating a SAT solver with an LCF-style theorem prover. In Alessandro Armando and Alessandro Cimatti, editors, *Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 67–78. Elsevier, January 2006.

[168] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, July 2007. To appear.

[169] Christoph Weidenbach, Bijan Afshordel, Uwe Brahm, Christian Cohrs, Thorsten Engel, Enno Keen, Christian Theobalt, and Dalibor Topic. System description: SPASS version 1.0.0. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–318. Springer, 1999.

[170] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics – 10th International Conference, TPHOLs '97, Murray Hill, NJ, USA, August 1922, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.

[171] Markus Wenzel. *Isabelle/Isar— a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2002.

[172] Wikipedia. Sudoku – Wikipedia, the free encyclopedia, September 2005. Available from `http://en.wikipedia.org/wiki/Sudoku`.

[173] Wikipedia. 2-3 tree – Wikipedia, the free encyclopedia, October 2007. Available from `http://en.wikipedia.org/w/index.php?title=2-3_tree&oldid=167942797`.

[174] Wikipedia. Markov decision process – Wikipedia, the free encyclopedia, March 2008. Available from `http://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=193875137`.

[175] Steve Winker. Generation and verification of finite models and counterexamples using an automated theorem prover answering two open questions. *Journal of the ACM*, 29(2):273–284, April 1982.

[176] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. In *IPSJ SIG Notes 2002-AL-87-2*. IPSJ, 2002.

[177] Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In Morgan Kaufmann, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal, Québec, Canada, August 20-25, 1995, Volume 1*, pages 298–303, 1995.

[178] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In Andrei Voronkov, editor, *Proceedings of the 8th International Conference on Computer Aided Deduction (CADE 2002)*, volume 2392 of *Lecture Notes in Computer Science*. Springer, 2002.

[179] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE 2003)*, pages 10880–10885. IEEE Computer Society, 2003.

[180] Irina Zhitomirskaja. Werkzeuggestützte modellbasierte Sicherheitsanalyse: RSAPSS Signaturverfahren. Master's thesis, Institut für Informatik, Technische Universität München, Germany, October 2005.