Technische Universität München
Lehrstuhl für Logik und Verifikation

# Automatic Proofs and Refutations for Higher-Order Logic

**Jasmin Christian Blanchette**

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. Dr. Helmut Seidl |
| Prüfer der Dissertation: |  |  |
|  | 1. | Univ.-Prof. Tobias Nipkow, Ph.D. |
|  | 2. | Prof. Koen Claessen, Ph.D.<br>Technische Hochschule Chalmers, Schweden |

Die Dissertation wurde am 02.03.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.05.2012 angenommen.

# Abstract

This thesis describes work on two components of the interactive theorem prover Isabelle/HOL that generate proofs and counterexamples for higher-order conjectures by harnessing external first-order reasoners.

Our primary contribution is the development of Nitpick, a counterexample generator that builds on a first-order relational model finder based on a Boolean satisfiability (SAT) solver. Nitpick supports (co)inductive predicates and datatypes as well as (co)recursive functions. A novel aspect of this work is the use of a monotonicity inference to prune the search space and to soundly interpret infinite types with finite sets, leading to considerable speed and precision improvements. In a case study, Nitpick was successfully applied to an Isabelle formalization of the C++ memory model.

Our second main contribution is the further development of the Sledgehammer proof tool. This tool heuristically selects facts relevant to the conjecture to prove, delegates the problem to first-order resolution provers, and, if a prover is successful, outputs a command that reconstructs the proof in Isabelle. We extended Sledgehammer to invoke satisfiability modulo theories (SMT) solvers as well, exploiting the existing relevance filter and parallel architecture. To address longstanding user complaints, we developed a family of sound, complete, and efficient encodings of polymorphic types in untyped first-order logic and looked into ways to make proof reconstruction faster and more reliable.

# Acknowledgment

I first want to express my gratitude to Tobias Nipkow, who supervised this thesis. He guided me toward two fruitful and complementary research areas that perfectly matched my interests. His clear thinking and vision has helped me focus on the needs of Isabelle users, and his bug reports have led to many improvements to both Nitpick and Sledgehammer.

To my delight, Koen Claessen accepted the invitation to referee this thesis. His group's work on first-order provers and model finders is closely connected to my research. During my three-week stay in Gothenburg, he shared many insights about finite, infinite, and nonstandard model finding, and many other things. He took the trouble to read a draft of this thesis and made dozens of suggestions.

I had the pleasure to work with the following former, current, and future members of the Isabelle group in Munich: Clemens Ballarin, Stefan Berghofer, Sascha Böhme, Lukas Bulwahn, Florian Haftmann, Johannes Hölzl, Brian Huffman, Cezary Kaliszyk, Alexander Krauss, Ondřej Kunčar, Peter Lammich, Lars Noschinski, Andrei Popescu, Dmitriy Traytel, Thomas Türk, Christian Urban, and Makarius Wenzel.

I gratefully thank my girlfriend, Anja Palatzke, as well as my friend Trenton Schulz for their unfailing encouragement and support in this project. Despite rustiness with formal methods, my friend Mark Summerfield read through the manuscript. His suggestions considerably improved the quality of English throughout. My friends and colleagues Sascha Böhme and Tjark Weber also commented on drafts of this thesis, for which I am very thankful.

Much of the text of this thesis has appeared before in scientific papers. The following people provided comments that led to textual improvements to the papers: Stefan Berghofer, Sascha Böhme, Chad Brown, Lukas Bulwahn, Pascal Fontaine, Marcelo Frias, Florian Haftmann, Paul Jackson, Alexander Krauss, Peter Lammich, Rustan Leino, Ann Lillieström, Andreas Lochbihler, Tobias Nipkow, Andrei Popescu, Peter Sewell, Mark Summerfield, Geoff Sutcliffe, Emina Torlak, Tjark Weber, and numerous anonymous reviewers.

Tjark Weber created the Refute tool that guided Nitpick's design. As a user of Nitpick, he provided feedback on the tool and tried it out on a formalization of the C++ memory model he was developing together with Mark Batty, Scott Owens, Susmit Sarkar, and Peter Sewell, paving the way for this thesis's main case study.

Emina Torlak developed the wonderful Kodkod model finder upon which Nitpick is built. During the last three years, she frequently provided advice and shed some light onto the inner workings of Kodkod.

Alexander Krauss codeveloped the monotonicity calculi presented in this thesis. His fixation on simplicity and generality benefited both the calculi and their soundness proofs. He is the one to thank for the pleasing type-system flavor of the calculi.

Andreas Lochbihler, Denis Lohner, and Daniel Wasserrab were among the first users of Nitpick and provided much helpful feedback. Geoff Sutcliffe installed Nitpick on his Miami computer farm and runs it regularly against other model finders; in response to my request, he introduced a higher-order refutation division to the annual CADE ATP System Competition (CASC).

On the Sledgehammer front, my first acknowledgment is of Lawrence Paulson, whose team developed the first version of the tool. I am particularly grateful for the numerous email discussions and the joint papers.

Sascha Böhme did most of the hard work necessary to extend Sledgehammer with SMT solvers, and Nik Sultana helped me integrate the higher-order automatic provers LEO-II and Satallax as additional backends. Sascha also developed the $\lambda$-lifting procedure and the monomorphizer that are now part of the interface with resolution provers. Charles Francis, financed by the Google Summer of Code program, implemented the proof redirection algorithm described here.

Following on the work with Alexander Krauss on monotonicity, Koen Claessen, Ann Lillieström, and Nicholas Smallbone showed how to exploit monotonicity to encode types in an untyped logic. Nicholas was gracious enough to let me implement some of his unpublished ideas, which now lie at the heart of Sledgehammer's sound, complete, and efficient type encodings.

The authors of the automatic provers upon which Sledgehammer is built helped in various ways. Stephan Schulz extended E with two new weight functions at my request. Daniel Wand and Christoph Weidenbach are working on several SPASS features tailored to Sledgehammer's needs. Kryštof Hoder and Andrei Voronkov were quick to address the few issues I encountered with Vampire. Nikolaj Bjørner and Leonardo de Moura promptly fixed a critical bug in Z3's proof generator. Christoph Benzmüller added an option to LEO-II to control the proof output's level of detail, and Chad Brown added an option to Satallax to output the unsatisfiable core; both suggested alternative translations for $\lambda$-abstractions, which I am looking forward to trying out. Joe Hurd helped me upgrade to the latest Metis and kindly applied Isabelle-motivated patches to his repository.

Geoff Sutcliffe, whose involvement with Nitpick was acknowledged above, also deserves credit in the context of Sledgehammer. He maintains the SystemOnTPTP service, which I found invaluable for experimenting with hard-to-install provers. Apart from occasional reminders about my "hammering the service quite hard," he showed himself truly service-minded. He also introduced a problem category consisting exclusively of Sledgehammer-generated problems as part of CASC, a step that could lead to new prover versions that are better suited to Isabelle. Finally, I want to thank the other participants of the Polymorphic TPTP TFF mailing list, especially François Bobot, Chad Brown, Viktor Kuncak, Andrei Paskevich, Florian Rabe, Philipp Rümmer, Stephan Schulz, and Josef Urban.

My research was financed by the Deutsche Forschungsgemeinschaft project *Quis Custodiet* (grants Ni 491/11-1 and Ni 491/11-2).

# Contents

When we're faced with a "prove or disprove,"
we're usually better off trying first to disprove with
a counterexample, for two reasons: A disproof is
potentially easier (we need just one counterexample);
and nitpicking arouses our creative juices.

— R. Graham, D. Knuth, and O. Patashnik (1988)

# Chapter 1

# Introduction

This thesis describes work on two components of the interactive theorem prover Isabelle/HOL—Sledgehammer and Nitpick—that generate proofs and counterexamples for higher-order conjectures by harnessing external first-order reasoners.

## 1.1 Motivation

Anecdotal evidence suggests that most "theorems" initially given to an interactive theorem prover do not hold, typically because of a typo or a missing assumption, but sometimes because of a fundamental flaw. Andreas Lochbihler presented a beautiful example of this at TPHOLs 2009 [113]. Having just proved the lemma

$$\{a,b\} = \{x,y\} \longleftrightarrow a = x \wedge b = y \;\vee\; a = y \wedge b = x$$

with an automatic proof tactic, we would naturally expect the generalization to sets of three variables to be provable as well:

$$\begin{aligned}\{a,b,c\} = \{x,y,z\} \longleftrightarrow\; & a = x \wedge b = y \wedge c = z \;\vee\; a = x \wedge b = z \wedge c = y \;\vee \\ & a = y \wedge b = x \wedge c = z \;\vee\; a = y \wedge b = z \wedge c = x \;\vee \\ & a = z \wedge b = x \wedge c = y \;\vee\; a = z \wedge b = y \wedge c = x\end{aligned}$$

We can waste much time and effort trying various proof tactics before noticing that there is a flaw in our thinking: If $a = b = x \neq c = y = z$, the left-hand side is true but the right-hand side is false.

As useful as they might be, automatic proof tools are mostly helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy.

To spare users the Sisyphean task of trying to prove non-theorems, most modern proof assistants include counterexample generators to debug putative theorems or specific subgoals in an interactive proof. For some years, Isabelle/HOL [140] has included two such tools:

- Refute [198] systematically searches for finite countermodels of a formula through a reduction to SAT (propositional satisfiability).
- Quickcheck [21] combines Isabelle's code generator with random testing, in the style of QuickCheck for Haskell [59]. It analyses inductive definitions to

generate values that satisfy them by construction [49] and has recently been extended with exhaustive testing and narrowing [30, §4].

Their areas of applicability are somewhat disjoint: Quickcheck excels at inductive datatypes but is restricted to an executable fragment of higher-order logic (which excludes unbounded quantifiers ranging over infinite types as well as underspecification) and may loop endlessly on inductive predicates. Refute does not impose restrictions on the form of the formulas, but this comes at the cost of frequent spurious counterexamples; inductive datatypes and predicates are mostly out of reach due to combinatorial explosion.

In the first-order world, the Alloy Analyzer testing tool [94] has achieved considerable popularity and features in several case studies. Its backend, the finite model finder Kodkod [187], reduces relational logic problems to SAT. Alloy's success inspired us to develop a new counterexample generator for Isabelle, called Nitpick.[1] It employs Kodkod as its backend, thereby benefiting from Kodkod's optimizations and its rich relational logic. Nitpick searches for finite fragments (substructures) of infinite countermodels, soundly approximating problematic constructs.

A failure to falsify the conjecture corroborates it and suggests that it might be a theorem. At this point, the actual proving begins. In the tradition of LCF-style provers [83], Isabelle has long emphasized *tactics*—functions written in Standard ML [129] that operate on the proof state via a trusted inference kernel. Tactics discharge a proof goal directly or, more often, break it down into one or more subgoals that must then be tackled by other tactics. In the last decade, the structured Isar language [139,202] has displaced ML as the language of choice for Isabelle proofs, but the most important ML tactics are still available as Isar proof methods.

Much effort has been devoted to developing general-purpose proof methods (or tactics) that work equally well on all object logics supported by Isabelle, notably higher-order logic (HOL) [82] and Zermelo–Fraenkel set theory (ZF) [145,146]. The most important methods are the simplifier [134], which rewrites the goal using equations as oriented rewrite rules, and the tableau prover [147]. These are complemented by specialized decision procedures, especially for arithmetic [53]. For the users of an interactive theorem prover, one of the main challenges is to find out which proof methods to use and which arguments to specify.

Although proof methods are still the mainstay of Isabelle proofs, the last few years have seen the emergence of advisory proof tools that work outside the LCF-style inference kernel. Some of these tools are very simple and yet surprisingly effective; for example, one searches Isabelle's libraries for a lemma that can prove the current goal directly, and another tries common proof methods.

The most important proof tool besides the simplifier and the tableau prover is probably Sledgehammer [152], a subsystem that harnesses first-order resolution provers. Given a conjecture, it heuristically selects a few hundred facts (lemmas, definitions, or axioms) from Isabelle's libraries, translates them to first-order logic along with the conjecture, and delegates the proof search to the external provers.

[1] The name Nitpick is shamelessly appropriated from Alloy's venerable precursor [93].

Sledgehammer boasts a fairly high success rate on goals that cannot be discharged directly by standard proof methods: In a study involving older proof documents, Sledgehammer could handle 34% of the more difficult goals arising in those proofs [44, §6]. The tool also works well in combination with the structured Isar language: The new way of teaching Isabelle is to let students think up intermediate properties and rely on Sledgehammer to fill in the gaps, rather than teach them low-level tactics and have them memorize lemma libraries [152, §4].

Sledgehammer was originally developed by Lawrence Paulson and his team in Cambridge, UK. The Isabelle team in Munich eventually took over its maintenance and further development. Once the more pressing technical issues had been sorted out, we became very much interested in increasing the success rate and identified three main avenues to achieve this: Embrace other classes of automatic provers besides first-order resolution provers. Address the soundness issues in the problem preparation phase. Make proof reconstruction in Isabelle faster and more reliable.

## 1.2 Contributions

The primary contribution of this thesis is the development of the counterexample generator Nitpick. The basic translation from Isabelle's higher-order logic to Kodkod's first-order relational logic is conceptually simple; however, common Isabelle idioms, such as (co)inductive predicates and datatypes and (co)recursive functions, are treated specially to ensure efficient SAT solving. Experimental results on Isabelle theories and the TPTP library indicate that Nitpick generates more counterexamples than other model finders for higher-order logic, without a priori restrictions on the form of the conjecture. In a case study, we applied the tool to an Isabelle formalization of the C++ memory model.

Our second main contribution is the further development of the Sledgehammer proof tool. Complementing the resolution provers with satisfiability modulo theories (SMT) solvers, which combine a satisfiability solver with decision procedures for first-order theories, adds considerable power to an already powerful tool. Previous schemes for encoding Isabelle's rich type information in untyped first-order logic were either unsound or too inefficient; although Isabelle certifies external proofs, unsound proofs are annoying and may conceal sound proofs. Our new approach encodes polymorphic types in a sound, complete, and efficient way. Finally, we devised an algorithm for translating the resolution proofs found by external provers into direct Isar proofs, to make them more attractive to Isabelle users.

A leitmotif of our work is the reliance on type-monotonicity inferences. Monotonicity naturally occurs in the context of finite model finding, where it helps prune the search space and allows us to soundly interpret infinite types with finite sets. Our Gothenburg colleagues discovered that monotonicity can also be exploited to encode types soundly and efficiently [61], and Sledgehammer's new type encodings build on their ideas. The development of the monotonicity inferences and the underlying theory, together with their application in Nitpick and Sledgehammer, surely constitutes the most serendipitous aspect of our research.

## 1.3 Publications

Most of the contributions described here have been presented at conferences and workshops, and two of the conference papers have subsequently been extended and accepted in journals. This thesis builds on the following papers:

1. J. C. B. and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.
2. J. C. B., T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *PPDP 2011*, pages 113–124. ACM Press, 2011.
3. J. C. B., S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNAI*, pages 207–221. Springer, 2011.
4. J. C. B., L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNAI*, pages 12–27. Springer, 2011.
5. J. C. B. and A. Krauss. Monotonicity inference for higher-order formulas. *J. Autom. Reasoning*, 47(4):369–398, 2011.
6. J. C. B. Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. To appear in *Softw. Qual. J.*
7. J. C. B., S. Böhme, and N. Smallbone. Monotonicity or how to encode polymorphic types safely and efficiently. Submitted.

Joint work is presented here with the coauthors' permission. The following papers were also written as part of the Ph.D. but are beyond the scope of this thesis:

8. J. C. B. Proof pearl: Mechanizing the textbook proof of Huffman's algorithm in Isabelle/HOL. *J. Autom. Reasoning*, 43(1):1–18, 2009.
9. J. C. B. and K. Claessen. Generating counterexamples for structural inductions by exploiting nonstandard models. In C. G. Fermüller and A. Voronkov, editors, *LPAR-17*, number 6397 in LNAI, pages 117–141. Springer, 2010.
10. L. C. Paulson and J. C. B. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, E. Ternovska, and S. Schulz, editors, *IWIL-2010*, 2010.
11. D. Traytel, A. Popescu, and J. C. B. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. Submitted.
12. J. C. B. and A. Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. Submitted.
13. J. C. B., A. Popescu, D. Wand, and C. Weidenbach. More SPASS with Isabelle—Superposition with hard sorts and configurable simplification. Submitted.

## 1.4 Structure of This Thesis

Although our title puts Proofs before Refutations in line with Imre Lakatos's celebrated essay [108], our presentation follows Graham, Knuth, and Patashnik's advice and makes the nitpicking that arouses our creative juices precede the heavy proof hammering. In more concrete terms:

- Chapter 2 briefly introduces the syntax and semantics of higher-order logic as well as the main definitional principles offered by Isabelle/HOL.
- Chapter 3 describes Nitpick's translation to the relational logic understood by Kodkod and demonstrates the tool on realistic examples.
- Chapter 4 presents the monotonicity inference calculi together with soundness proofs.
- Chapter 5 applies Nitpick to a formalization of the C++ memory model.
- Chapter 6 describes various enhancements to Sledgehammer and evaluates them on a large benchmark suite.
- Chapter 7 summarizes our results and gives directions for future work.

Notwithstanding Graham et al., Chapters 3 to 6 can be read in any order. Related work is considered at the end of each chapter. The raw test data for the various evaluations are available at `http://www21.in.tum.de/~blanchet/phd-data.tgz`.

## 1.5 A Note on the Proofs

Pen-and-paper proofs are included in the body of this thesis. Although mechanized Isabelle proofs would be desirable for the entirety of our metatheory, before this becomes cost-effective Nitpick and Sledgehammer would have to be much more powerful: Their increased power would make such an effort more tractable and simultaneously justify the time (and taxpayer-money) expenditure.

Nitpick's translation to relational logic in Chapter 3 is supported by short proof sketches. Part of the reason is that finite model finding for HOL is well understood nowadays; indeed, a very thorough soundness proof of a translation from HOL to SAT can be found in Tjark Weber's Ph.D. thesis [198, ch. 2]. Nitpick is also easy to test, by checking that it finds no counterexamples to the tens of thousands of lemmas in the standard Isabelle libraries and the *Archive of Formal Proofs* [102]. In contrast, the monotonicity work in Chapter 4 is more difficult to test effectively, and its abstract flavor calls for very detailed proofs; accordingly, much of the chapter is devoted to proving our monotonicity inference calculi sound.

Most of the enhancements to Sledgehammer described in Chapter 6 require at most easy or well-known arguments, which are omitted. Assuming Isabelle's inference kernel is sound, any unsoundness in the translation would be caught during proof reconstruction. We make an exception for the application of monotonicity to encode HOL types, which is novel and interesting enough in its own right to deserve a detailed proof of correctness.

> HOL is weaker than ZF set theory but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF.
>
> — Lawrence C. Paulson (1993)

# Chapter 2

# Isabelle/HOL

Isabelle [140] is a generic interactive theorem prover whose built-in metalogic is an intuitionistic fragment of higher-order logic [4, 56, 82]. The HOL object logic provides a more elaborate version of higher-order logic, complete with the familiar connectives and quantifiers. Isabelle/HOL provides various definitional principles for introducing new types and constants safely. This chapter provides a brief introduction to Isabelle/HOL, focusing on the features that are needed in this thesis and taking some liberties with the official syntax to lighten the presentation.

## 2.1 The Metalogic

**Definition 2.1 (Syntax).** The *types* and *terms* of Isabelle are that of the simply typed $\lambda$-calculus, augmented with constants, $n$-ary type constructors, and ML-style polymorphism.

$$
\begin{array}{lll@{\qquad\qquad}lll}
\text{Types:} & & & \text{Terms:} & & \\
\sigma ::= & \alpha & \text{type variable} & t ::= & x^{\sigma} & \text{variable} \\
\mid & \bar{\sigma}\,\kappa & \text{type constructor} & \mid & \mathsf{c}^{\sigma} & \text{constant} \\
 & & & \mid & \lambda x^{\sigma}.\ t & \lambda\text{-abstraction} \\
 & & & \mid & t\ t' & \text{application}
\end{array}
$$

We reserve the Greek letters $\alpha$, $\beta$, $\gamma$ for type variables, $\sigma$, $\tau$, $\upsilon$ for types, $x$, $y$, $z$ for term variables, and $t$, $u$ for terms, although we will also use many other letters for terms. Lists of zero or more instances of a syntactic quantity are indicated by a bar (e.g., $\bar{\sigma}$). Type constructors are written in ML-style postfix notation (e.g., $\alpha$ *list*, $(\alpha, \beta)$ *map*). We set constants in sans serif to distinguish them from variables. Function application expects no parentheses around the argument list and no commas between the arguments, as in $f\ x\ y$. Syntactic sugar provides an infix syntax for common operators, such as $x = y$ and $x + y$.

The standard semantics interprets the type *prop* of propositions (formulas) and the type of functions $\alpha \rightarrow \beta$. The function arrow associates to the right, reflecting the left-associativity of application. We assume throughout that terms are well-typed using the following (simplified) typing rules:

$$
\frac{}{\vdash x^{\sigma} : \sigma} \qquad \frac{}{\vdash \mathsf{c}^{\sigma} : \sigma} \qquad \frac{\vdash t : \tau}{\vdash \lambda x^{\sigma}.\ t : \sigma \rightarrow \tau} \qquad \frac{\vdash t : \sigma \rightarrow \tau \quad \vdash u : \sigma}{\vdash t\ u : \tau}
$$

In keeping with the logic's higher-order nature, variables can have function types. We write $x$ and $\mathsf{c}$ rather than $x^{\sigma}$ and $\mathsf{c}^{\sigma}$ when the type $\sigma$ is irrelevant or clear from the context; conversely, we write $t^{\sigma}$ to indicate that an arbitrary term $t$ has type $\sigma$.

The metalogical operators are

| | |
|---|---|
| $\Longrightarrow^{prop\to prop\to prop}$ | implication |
| $\bigwedge^{(\alpha\to prop)\to prop}$ | universal quantification |
| $\equiv^{\alpha\to\alpha\to prop}$ | equality |

We write $\bigwedge x.\ t$ as an abbreviation for $\bigwedge\ (\lambda x.\ t)$ and similarly for the other binders introduced later. The operators $\Longrightarrow$ and $\equiv$ are written infix. In addition to the usual equality rules (symmetry, reflexivity, transitivity, and substitutivity), the $\alpha$-renaming, $\beta$-reduction, and $\eta$-expansion rules from the $\lambda$-calculus also apply:

$$\frac{}{(\lambda x.\ t[x]) \equiv (\lambda y.\ t[y])}\ \alpha \qquad \frac{}{(\lambda x.\ t[x])\ u \equiv t[u]}\ \beta \qquad \frac{}{t^{\sigma\to\tau} \equiv (\lambda x^{\sigma}.\ t\ x)}\ \eta$$

Proviso for $\eta$: $x^{\sigma}$ does not appear free in $t$.

For both types and terms, Isabelle distinguishes between two kinds of free variable. *Schematic variables* can be instantiated arbitrarily, whereas *nonschematic variables* represent fixed but unknown terms or types. Although formulas can involve both kinds of variable simultaneously, this is rarely necessary: When stating a conjecture and proving it, the type and term variables are normally fixed, and once it is proved, they become schematics so that users of the lemma can instantiate them as they wish. In this thesis, we restrict ourselves to this use case and let the context (lemma or conjecture) determine whether the variables are schematic.

## 2.2 The HOL Object Logic

HOL is the most widely used instance of Isabelle and is the only object logic supported by Nitpick and Sledgehammer. It provides classical higher-order logic [4, 56, 82] with ML-style rank-1 polymorphism, extended with Haskell-style axiomatic type classes [135, 201].

HOL axiomatizes a type *bool* of Booleans, which we abbreviate to $o$ (omicron). It defines the constants $\mathsf{False}^{o}$, $\mathsf{True}^{o}$, and $=^{\alpha\to\alpha\to o}$ (with $\longleftrightarrow^{o\to o\to o}$ as surface syntax), the connectives $\neg^{o\to o}$, $\wedge^{o\to o\to o}$, $\vee^{o\to o\to o}$, and $\longrightarrow^{o\to o\to o}$, the quantifiers $\forall^{(\alpha\to o)\to o}$ and $\exists^{(\alpha\to o)\to o}$, and the conditional expression 'if then else'$^{o\to\alpha\to\alpha\to\alpha}$. Equality on functions is extensional:

$$(\bigwedge x.\ f\ x = g\ x) \Longrightarrow f = g$$

HOL also provides the definite and indefinite description operators $\iota^{(\alpha\to o)\to\alpha}$ and $\varepsilon^{(\alpha\to o)\to\alpha}$ axiomatized by

$$(\iota x.\ x = a) = a \qquad\qquad P\ x \Longrightarrow P\ (\varepsilon\ P)$$

The $\varepsilon$ operator is often called Hilbert choice. Both operators are binders; $\varepsilon\ P$ can be $\eta$-expanded into $\varepsilon\ (\lambda x.\ P\ x)$, i.e., $\varepsilon x.\ P\ x$. HOL types are inhabited (nonempty), and we can obtain an arbitrary element of a type $\sigma$ using either $\varepsilon x^{\sigma}.\ \mathsf{True}$ or the unspecified constant $\mathsf{undefined}^{\sigma}$.

HOL is embedded into the metalogic using $\mathsf{Trueprop}^{o\to prop}$. Isabelle's parser inserts it as appropriate and the output routine ignores it, so that users are hardly ever exposed to it. We follow this convention and write, for example, $\mathsf{False} \Longrightarrow \mathsf{True}$ rather than $\mathsf{Trueprop\ False} \Longrightarrow \mathsf{Trueprop\ True}$.

There is considerable overlap between the metalogic and the HOL object logic. Many Isabelle tools must distinguish the two layers, but for Nitpick and Sledgehammer this is hardly relevant: Both tools effectively treat *prop* the same as $o$, $\Longrightarrow$ the same as $\longrightarrow$, $\bigwedge$ the same as $\forall$, and $\equiv$ the same as $=$. Some properties can only be expressed in the metalogic, but these are of foundational interest and play no role here. We preserve the distinction between the two levels in the examples, to avoid distracting the trained Isabelle eye, but otherwise assume formulas are expressed purely in the object logic. Readers unfamiliar with Isabelle are encouraged to ignore the distinction when reading this thesis.

The standard semantics of HOL is given in terms of Zermelo–Fraenkel set theory with the axiom of choice (ZFC) [82, ch. 15]. The proofs in Sections 3 and 4 require only a monomorphic fragment of HOL, in which the only type variables are nonschematic. To simplify these proofs, we depart from the HOL tradition and treat polymorphism in the metalanguage: Polymorphic constants such as equality are expressed as collections of constants, one for each type.

Types and terms are interpreted relative to a scope that fixes the interpretation of types. Scopes are also called "type environments." Our terminology here is consistent with Jackson [95].

**Definition 2.2 (Scope).** A *scope* $S$ is a function from (nonschematic) types to nonempty sets (the *domains*).

**Definition 2.3 (Interpretation of Types).** The *interpretation* $[\![\sigma]\!]_S$ of a type $\sigma$ in a scope $S$ is defined recursively by the equations

$$[\![o]\!]_S = \{\bot, \top\} \qquad [\![\sigma \to \tau]\!]_S = [\![\sigma]\!]_S \to [\![\tau]\!]_S \qquad [\![\bar{\sigma}\,\kappa]\!]_S = S(\bar{\sigma}\,\kappa)$$

where $\bot \neq \top$ and $A \to B$ denotes the set of (total) functions from $A$ to $B$.[1]

**Definition 2.4 (Model).** A *constant model* is a scope-indexed family of functions $M_S$ that map each constant $\mathsf{c}^\sigma$ to a value $M_S(\mathsf{c}) \in [\![\sigma]\!]_S$. A constant model is *standard* if it gives the usual interpretation to equality and implication. A *variable assignment* $V$ for a scope $S$ is a function that maps each variable $x^\sigma$ to a value $V(x) \in [\![\sigma]\!]_S$. A *model* for $S$ is a triple $\mathcal{M} = (S, V, M)$, where $V$ is a variable assignment for $S$ and $M$ is a constant model.

**Definition 2.5 (Interpretation of Terms).** Let $\mathcal{M} = (S, V, M)$ be a model. The *interpretation* $[\![t]\!]_{\mathcal{M}}$ of a term $t$ in $\mathcal{M}$ is defined recursively by the equations

$$\begin{aligned} [\![x]\!]_{(S,V,M)} &= V(x) & [\![t\ u]\!]_{(S,V,M)} &= [\![t]\!]_{(S,V,M)}\big([\![u]\!]_{(S,V,M)}\big) \\ [\![\mathsf{c}]\!]_{(S,V,M)} &= M_S(\mathsf{c}) & [\![\lambda x^\sigma.\ t]\!]_{(S,V,M)} &= a \in [\![\sigma]\!]_S \mapsto [\![t]\!]_{(S,V[x\mapsto a],M)} \end{aligned}$$

[1] Metatheoretic functions here and elsewhere in this thesis are defined using sequential pattern matching. As a result, the second equation is preferred over the third when both are applicable.

where $a \in A \mapsto f(a)$ denotes the function with domain $A$ that maps each $a \in A$ to $f(a)$. If $t$ is a formula and $[\![t]\!]_{\mathcal{M}} = \top$, we say that $\mathcal{M}$ is a *model* of $t$, written $\mathcal{M} \vDash t$. A formula is *satisfiable* for scope $S$ if it has a model for $S$.

The HOL connectives and quantifiers can be defined in terms of equality and implication. Definitions also cater for set-theoretic notations.[1]

**Definition 2.6 (Logical Constants).**

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{True} & \equiv & (\lambda x^{o}.\; x) = (\lambda x.\; x) & p \wedge q & \equiv & \neg\,(p \longrightarrow \neg\, q) \\
\mathsf{False} & \equiv & (\lambda x^{o}.\; x) = (\lambda x.\; \mathsf{True}) & p \vee q & \equiv & \neg\, p \longrightarrow q \\
\neg\, p & \equiv & p \longrightarrow \mathsf{False} & \forall x^{\sigma}.\; p & \equiv & (\lambda x.\; p) = (\lambda x.\; \mathsf{True}) \\
p \neq q & \equiv & \neg\,(p = q) & \exists x^{\sigma}.\; p & \equiv & \neg\, \forall x.\; \neg\, p
\end{array}
$$

**Definition 2.7 (Set Constants).**

$$
\begin{array}{rcl@{\qquad}rcl}
\emptyset & \equiv & \lambda x.\; \mathsf{False} & s - t & \equiv & \lambda x.\; s\; x \wedge \neg\, t\; x \\
\mathsf{UNIV} & \equiv & \lambda x.\; \mathsf{True} & x \in s & \equiv & s\; x \\
s \cap t & \equiv & \lambda x.\; s\; x \wedge t\; x & \mathsf{insert}\; x\; s & \equiv & (\lambda y.\; y = x) \cup s \\
s \cup t & \equiv & \lambda x.\; s\; x \vee t\; x & & &
\end{array}
$$

The constants $\emptyset$ and $\mathsf{insert}$ can be seen as (non-free) constructors for finite sets. Following tradition, we write $\{x_1, \ldots, x_m\}$ instead of $\mathsf{insert}\; x_1\; (\ldots (\mathsf{insert}\; x_m\; \emptyset) \ldots)$.

## 2.3 Definitional Principles

Isabelle/HOL provides a rich array of ways to introduce new type constructors and constants. Types and constants can be specified axiomatically, but this is generally avoided because it can easily lead to inconsistent specifications.

**Type Declarations.** The type declaration command

**typedecl** $\bar{\alpha}\; \kappa$

introduces a new uninterpreted type constructor $\kappa$ whose arity is given by the number of type variables $\bar{\alpha}$ [140, §8.5.1]. The type $\bar{\alpha}\; \kappa$ is completely unspecified except that each of its ground instances is inhabited.

**Type Definitions.** A more interesting mechanism for expanding the logic is to introduce new types as isomorphic to subsets of existing types. The syntax is

**typedef** $\bar{\alpha}\; \kappa \;=\; t^{\sigma \to o}$

where $t$ must be a provably nonempty set for any instantiation of $\bar{\alpha}$ [140, §8.5.2]. For example, assuming a type $nat$ of natural numbers with $0, 1, 2, \ldots,$ the command

**typedef** $three \;=\; \{0^{nat}, 1, 2\}$

[1] Isabelle versions 2007 to 2011-1 identify sets and predicates, as we do here: $\alpha\; set$ is surface syntax for $\alpha \to o$. Future versions are expected to reintroduce the distinction between $\alpha\; set$ and $\alpha \to o$.

introduces a new type with exactly three values. The bijection between the new type *three* and the subset $\{0, 1, 2\}$ of $\mathsf{UNIV}^{nat \to o}$ is accessible as $\mathsf{Rep\_three}^{three \to nat}$ and $\mathsf{Abs\_three}^{nat \to three}$.

**Simple Definitions.** A simple definition

$\textbf{definition}\ \mathsf{c}^{\sigma}\ \textbf{where}\ \mathsf{c}\ \bar{x} = t$

introduces a constant as equal to another term [140, §2.7.2]. Behind the scenes, the definition introduces the axiom $\mathsf{c} \equiv (\lambda \bar{x}.\ t)$. Isabelle checks the following provisos to ensure that the definition is conservative [201]: The constant $\mathsf{c}$ is fresh, the variables $\bar{x}$ are distinct, and the right-hand side $t$ does not refer to any other free variables than $\bar{x}$, to any undefined constants or $\mathsf{c}$, or to any type variables not occurring in $\sigma$. An example follows:

$\textbf{definition}\ \mathsf{K}^{\alpha \to \beta \to \alpha}\ \textbf{where}\ \mathsf{K}\ x\ y = x$

**(Co)inductive Predicates.** The **inductive** and **coinductive** commands define inductive and coinductive predicates specified by their introduction rules [148]:

$[\textbf{co}]\textbf{inductive}\ \mathsf{p}^{\sigma}\ \textbf{where}$
$Q_{11} \Longrightarrow \cdots \Longrightarrow Q_{1\ell_1} \Longrightarrow \mathsf{p}\ \bar{t}_1$
$\vdots$
$Q_{n1} \Longrightarrow \cdots \Longrightarrow Q_{n\ell_n} \Longrightarrow \mathsf{p}\ \bar{t}_n$

Among the provisos, the constant $\mathsf{p}$ must be fresh, and each $Q_{ij}$ is either a side condition that does not refer to $\mathsf{p}$ or is of the form $\mathsf{p}^{\sigma}\ \bar{u}$, where the arguments $\bar{u}$ do not refer to $\mathsf{p}$. The introduction rules may involve any number of free variables $\bar{y}$. If the **inductive** or **coinductive** keyword is replaced by **inductive_set** or **coinductive_set**, the applications of $\mathsf{p}$ must use the set membership operator (e.g., $u \in \mathsf{p}$).

The syntactic restrictions on the rules ensure monotonicity; by the Knaster–Tarski theorem, the fixed-point equation

$$\mathsf{p} = \big(\lambda \bar{x}.\ \exists \bar{y}.\ \textstyle\bigvee_{j=1}^{n} \bar{x} = \bar{t}_j \wedge Q_{j1} \wedge \cdots \wedge Q_{j\ell_j}\big)$$

admits a least and a greatest solution [86, 148]. Inductive definitions provide the least fixed point, and coinductive definitions provide the greatest fixed point. Internally, (co)inductive definitions are reduced to simple definitions.

Let us consider an example. Assuming a type *nat* of natural numbers generated freely by $0^{nat}$ and $\mathsf{Suc}^{nat \to nat}$, the following definition introduces the predicate even of even numbers:

$\textbf{inductive}\ \mathsf{even}^{nat \to o}\ \textbf{where}$
$\mathsf{even}\ 0$
$\mathsf{even}\ n \Longrightarrow \mathsf{even}\ (\mathsf{Suc}\ (\mathsf{Suc}\ n))$

The associated fixed-point equation is

$$\mathsf{even} = (\lambda x.\ \exists n.\ x = 0 \vee (x = \mathsf{Suc}\ (\mathsf{Suc}\ n) \wedge \mathsf{even}\ n))$$

which we can also express as $\mathsf{even} = F\ \mathsf{even}$, where

$$F = (\lambda f\ x.\ \exists n.\ x = 0 \lor (x = \mathsf{Suc}\ (\mathsf{Suc}\ n) \land f\ n))$$

Isabelle/HOL also supports mutual definitions, such as the following:

$\textbf{inductive}\ \mathsf{even}^{nat \to o}\ \textbf{and}\ \mathsf{odd}^{nat \to o}\ \textbf{where}$
$\mathsf{even}\ 0$
$\mathsf{even}\ n \Longrightarrow \mathsf{odd}\ (\mathsf{Suc}\ n)$
$\mathsf{odd}\ n \Longrightarrow \mathsf{even}\ (\mathsf{Suc}\ n)$

In general, mutual definitions for $\mathsf{p}_1, \ldots, \mathsf{p}_m$ can be reduced to a single predicate $\mathsf{q}$ whose domain is the disjoint sum of the domains of each $\mathsf{p}_i$ [148]. Assuming $\mathsf{Inl}^{\alpha \to \alpha + \beta}$ and $\mathsf{Inr}^{\beta \to \alpha + \beta}$ are the disjoint sum constructors, this definition of even and odd is reducible to

$\textbf{inductive}\ \mathsf{even\_odd}^{nat + nat \to o}\ \textbf{where}$
$\mathsf{even\_odd}\ (\mathsf{Inl}\ 0)$
$\mathsf{even\_odd}\ (\mathsf{Inl}\ n) \Longrightarrow \mathsf{even\_odd}\ (\mathsf{Inr}\ (\mathsf{Suc}\ n))$
$\mathsf{even\_odd}\ (\mathsf{Inr}\ n) \Longrightarrow \mathsf{even\_odd}\ (\mathsf{Inl}\ (\mathsf{Suc}\ n))$

$\textbf{definition}\ \mathsf{even}^{nat \to o}\ \textbf{where}\ \mathsf{even}\ n = \mathsf{even\_odd}\ (\mathsf{Inl}\ n)$
$\textbf{definition}\ \mathsf{odd}^{nat \to o}\ \textbf{where}\ \mathsf{odd}\ n = \mathsf{even\_odd}\ (\mathsf{Inr}\ n)$

To enhance readability, we sometimes present the introduction rules of a predicate in the customary rule format, with a simple horizontal bar for inductive predicates and a double bar for coinductive predicates. For example:

$$\frac{}{\mathsf{even}\ 0} \qquad \frac{\mathsf{even}\ n}{\mathsf{odd}\ (\mathsf{Suc}\ n)} \qquad \frac{\mathsf{odd}\ n}{\mathsf{even}\ (\mathsf{Suc}\ n)}$$

**(Co)inductive Datatypes.** The **datatype** command defines mutually recursive inductive datatypes specified by their constructors [22]:

$$\begin{aligned} \textbf{datatype}\ \bar\alpha\,\kappa_1 &= \mathsf{C}_{11}\ \bar\sigma_{11} \mid \cdots \mid \mathsf{C}_{1\ell_1}\ \bar\sigma_{1\ell_1} \\ \textbf{and}\ \ldots & \\ \textbf{and}\ \bar\alpha\,\kappa_n &= \mathsf{C}_{n1}\ \bar\sigma_{n1} \mid \cdots \mid \mathsf{C}_{n\ell_n}\ \bar\sigma_{n\ell_n} \end{aligned}$$

The defined types $\bar\alpha\,\kappa_i$ are parameterized by a list of distinct type variables $\bar\alpha$, providing type polymorphism. Each constructor $\mathsf{C}_{ij}$ has type $\bar\sigma_{ij} \to \bar\alpha\,\kappa_i$. The arguments $\bar\alpha$ must be the same for all the type constructors $\kappa_i$. The type constructors $\kappa_i$ and the constructor constants $\mathsf{C}_{ij}$ must be fresh and distinct, the type parameters $\bar\alpha$ must be distinct, and the argument types $\bar\sigma_{ij}$ may not refer to other type variables than $\bar\alpha$ (but may refer to the types $\bar\alpha\,\kappa_i$ being defined).

Types introduced by the **datatype** command roughly correspond to Standard ML datatypes [84,123]. A corresponding **codatatype** command, which allows infinite objects like Haskell **data** [154], is under development [189]. For the moment, Isabelle also provides a "lazy list" library that defines a coinductive datatype of lists without the help of a definitional package. Since Nitpick supports lazy lists and lets users register custom codatatypes, we find it convenient in this thesis to entertain the fiction that Isabelle already provides a **codatatype** command.

The (**co**)**datatype** commands can be used to define natural numbers, pairs, finite lists, and possibly infinite lazy lists as follows:

**datatype** $\mathit{nat} = 0 \mid \mathsf{Suc}\ \mathit{nat}$
**datatype** $\alpha \times \beta = \mathsf{Pair}\ \alpha\ \beta$
**datatype** $\alpha\ \mathit{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ \mathit{list})$
**codatatype** $\alpha\ \mathit{llist} = \mathsf{LNil} \mid \mathsf{LCons}\ \alpha\ (\alpha\ \mathit{llist})$

Mutually recursive trees and forests (lists of trees) can be defined just as easily:

**datatype** $\alpha\ \mathit{tree} = \mathsf{Empty} \mid \mathsf{Node}\ \alpha\ (\alpha\ \mathit{forest})$
  **and** $\alpha\ \mathit{forest} = \mathsf{FNil} \mid \mathsf{FCons}\ (\alpha\ \mathit{tree})\ (\alpha\ \mathit{forest})$

Defining a (co)datatype introduces the appropriate axioms for the constructors [148]. It also introduces a case combinator and syntactic sugar

$$\text{case } t \text{ of } \mathsf{C}_{i1}\ \bar{x}_1 \Rightarrow u_1 \mid \ldots \mid \mathsf{C}_{i\ell_i}\ \bar{x}_{\ell_i} \Rightarrow u_{\ell_i}$$

such that the following equation holds for $j \in \{1, \ldots, \ell_i\}$:

$$(\text{case } \mathsf{C}_{ij}\ \bar{x}_j \text{ of } \mathsf{C}_{i1}\ \bar{x}_1 \Rightarrow u_1 \mid \ldots \mid \mathsf{C}_{i\ell_i}\ \bar{x}_{\ell_i} \Rightarrow u_{\ell_i}) = u_j$$

**(Co)recursive Functions.** The **primrec** command defines primitive recursive functions on inductive datatypes [22]:

**primrec** $\mathsf{f}_1^{\sigma_1}$ **and** ... **and** $\mathsf{f}_n^{\sigma_n}$ **where**
$\mathsf{f}_1\ \bar{x}_{11}\ (\mathsf{C}_{11}\ \bar{y}_{11})\ \bar{z}_{11} = t_{11} \quad \ldots \quad \mathsf{f}_1\ \bar{x}_{1\ell_1}\ (\mathsf{C}_{1\ell_1}\ \bar{y}_{1\ell_1})\ \bar{z}_{1\ell_1} = t_{1\ell_1}$
$\vdots$
$\mathsf{f}_n\ \bar{x}_{n1}\ (\mathsf{C}_{n1}\ \bar{y}_{n1})\ \bar{z}_{n1} = t_{n1} \quad \ldots \quad \mathsf{f}_n\ \bar{x}_{n\ell_n}\ (\mathsf{C}_{n\ell_n}\ \bar{y}_{n\ell_n})\ \bar{z}_{n\ell_n} = t_{n\ell_n}$

The main proviso is that the middle argument of any recursive call in the $t_{ij}$'s must be one of the $\bar{y}_{ij}$'s. This ensures that each recursive call peels off one constructor from the argument and hence that the recursion is well-founded. Isabelle also provides **fun** and **function** commands for more general forms of recursion [106].

Like coinductive datatypes, corecursive functions are not directly supported by Isabelle, but Nitpick can handle them if the user registers them appropriately. Their postulated concrete syntax, inspired by the lazy list corecursion combinator [80, 114], follows a rather different schema, with a single equation per function $\mathsf{f}_i$ that must return type $\bar{\alpha}\ \kappa_i$:

**coprimrec** $\mathsf{f}_1^{\sigma_1}$ **and** ... **and** $\mathsf{f}_n^{\sigma_n}$ **where**
$\mathsf{f}_1\ \bar{y}_1 \ = \ \text{if } Q_{11} \text{ then } t_{11} \text{ else if } Q_{12} \text{ then } \ldots \text{ else } t_{1\ell_1}$
$\vdots$
$\mathsf{f}_n\ \bar{y}_n \ = \ \text{if } Q_{n1} \text{ then } t_{n1} \text{ else if } Q_{n2} \text{ then } \ldots \text{ else } t_{n\ell_n}$

Provisos: The constants $\mathsf{f}_i$ are fresh and distinct, the variables $\bar{y}_i$ are distinct, the right-hand sides involve no other variables than $\bar{y}_i$, no corecursive calls occur in the conditions $Q_{ij}$, and either $t_{ij}$ does not involve any corecursive calls or it has the form $\mathsf{C}_{ij}\ \bar{u}_{ij}$ for some codatatype constructor $\mathsf{C}_{ij}$.[1] The syntax can be relaxed to allow a 'case' expression instead of a sequence of conditionals; what matters is that

[1] Some authors formulate corecursion in terms of selectors instead of constructors [96].

corecursive calls are protected by constructors, to ensure that the functions $\mathsf{f}_i$ are productive and hence well-defined.

The following examples define concatenation for inductive and coinductive lists:

$$
\begin{array}{l}
\textbf{primrec } @^{\alpha\ list \to \alpha\ list \to \alpha\ list} \textbf{ where} \\
\mathsf{Nil}\ @\ zs = zs \\
\mathsf{Cons}\ y\ ys\ @\ zs = \mathsf{Cons}\ y\ (ys\ @\ zs)
\end{array}
$$

$$
\begin{array}{l}
\textbf{coprimrec } @_{\mathsf{L}}{}^{\alpha\ llist \to \alpha\ llist \to \alpha\ llist} \textbf{ where} \\
ys\ @_{\mathsf{L}}\ zs \;=\; \text{case } ys \text{ of} \\
\qquad\qquad\quad \mathsf{LNil} \Rightarrow zs \\
\qquad\qquad\quad \mid \mathsf{LCons}\ y\ ys' \Rightarrow \mathsf{LCons}\ y\ (ys'\ @_{\mathsf{L}}\ zs)
\end{array}
$$

At the moment you find an error, your brain may disappear because of the Heisenberg uncertainty principle, and be replaced by a new brain that thinks the proof is correct.

— Leonid A. Levin (1993)

# Chapter 3

# Counterexample Generation Using a Relational Model Finder

The Alloy Analyzer [94] is a lightweight checking tool that searches for counterexamples to claimed properties of specifications. It has achieved considerable popularity since its introduction a decade ago. Its specification language, Alloy, is a modeling language designed around a first-order relational logic. The Analyzer is based on Kodkod [187], a Java library that translates relational problems to propositional logic and invokes a SAT solver (usually MiniSat [74]).

Alloy's success inspired Tjark Weber to develop Refute [198], a SAT-based higher-order model finder for Isabelle/HOL. Refute excels at formulas with small finite models, such as those from the TPTP benchmark suite [181], and features in several case studies [98, 120, 197]. However, it suffers from soundness and scalability issues that severely impair its usefulness: Inductive datatypes and (co)inductive predicates are mostly out of its reach due to the combinatorial explosion, and conjectures involving datatypes often give rise to spurious countermodels.

Our Nitpick tool succeeds Refute and reuses many of its ideas: The translation from HOL is parameterized by the cardinalities of the types occurring in the problem; Nitpick systematically enumerates the cardinalities, so that if the conjecture has a finite countermodel, the tool eventually finds it, unless it runs out of resources. But unlike Refute, Nitpick employs Kodkod as its backend, thereby benefiting from Kodkod's rich relational logic (Section 3.1) and its optimizations. The basic translation from HOL is conceptually simple (Section 3.2). For common idioms such as (co)inductive datatypes and predicates, we departed from Refute and adopted more appropriate translation schemes that scale better while being sound (Section 3.3). Nitpick benefits from many novel optimizations that greatly improve its performance, especially in the presence of higher-order constructs (Section 3.4); one optimization, monotonicity inference, is treated separately in Chapter 4.

We present five small case studies: a context-free grammar, AA trees, a security type system, a hotel key card system, and lazy lists (Section 3.5). A larger case study, in which we apply Nitpick to a formalization of the C++ memory model, is treated in Chapter 5. We also evaluate Nitpick and its two rivals, Quickcheck [21, 30, 49] and Refute, on a large benchmark suite of mutated theorems (Section 3.6).

## 3.1 First-Order Relational Logic

Kodkod's logic, first-order relational logic (FORL), combines elements from first-order logic and relational calculus, to which it adds the transitive closure operator [187]. FORL terms range over *relations*—sets of tuples drawn from a finite universe of uninterpreted atoms. Relations can be of arbitrary arities. The logic is untyped, but each term denotes a relation of a fixed arity. Nitpick's translation relies on the following FORL fragment.[1]

Terms:

| | | |
|---|---|---|
| $r$ ::= | NONE | empty set |
| | $\mid$ IDEN | identity relation |
| | $\mid \mathsf{a}_n$ | atom |
| | $\mid x$ | variable |
| | $\mid r^+$ | transitive closure |
| | $\mid \pi^n_n(r)$ | projection |
| | $\mid r_1 . r_2$ | dot-join |
| | $\mid r_1 \times r_2$ | Cartesian product |
| | $\mid r_1 \cup r_2$ | union |
| | $\mid r_1 - r_2$ | difference |
| | $\mid r_1 \mathrel{++} r_2$ | override |
| | $\mid \{\langle x_1 \in r_1, \ldots, x_m \in r_m\rangle \mid \varphi\}$ | comprehension |
| | $\mid$ IF $\varphi$ THEN $r_1$ ELSE $r_2$ | conditional |

Formulas:

| | | |
|---|---|---|
| $\varphi$ ::= | FALSE | falsity |
| | $\mid$ TRUE | truth |
| | $\mid m\ r$ | multiplicity constraint |
| | $\mid r_1 = r_2$ | equality |
| | $\mid r_1 \subseteq r_2$ | inclusion |
| | $\mid \neg\, \varphi$ | negation |
| | $\mid \varphi_1 \wedge \varphi_2$ | conjunction |
| | $\mid \forall x \in r\colon \varphi$ | universal quantification |

Miscellaneous:

| | | |
|---|---|---|
| $m$ ::= | NO $\mid$ LONE $\mid$ ONE $\mid$ SOME | multiplicities |
| $n$ ::= | $1 \mid 2 \mid \cdots$ | positive integers |

The universe of discourse is $\mathcal{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$, where each $\mathsf{a}_j$ is a distinct uninterpreted atom. Atoms and $n$-tuples are identified with singleton sets and singleton $n$-ary relations, respectively; thus, the term $\mathsf{a}_1$ denotes the set $\{\mathsf{a}_1\}$ (or $\{\langle \mathsf{a}_1\rangle\}$), and $\mathsf{a}_1 \times \mathsf{a}_2$ denotes the binary relation $\{\langle \mathsf{a}_1, \mathsf{a}_2\rangle\}$. Bound variables range over the tuples in a relation; thus, $\forall x \in (\mathsf{a}_1 \cup \mathsf{a}_2) \times \mathsf{a}_3\colon \varphi[x]$ is equivalent to $\varphi[\mathsf{a}_1 \times \mathsf{a}_3] \wedge \varphi[\mathsf{a}_2 \times \mathsf{a}_3]$. Although they are not listed above, we sometimes make use of $\vee$ (disjunction), $\longrightarrow$ (implication), $^*$ (reflexive transitive closure), and $\cap$ (intersection).

[1] Our syntax for FORL is inspired by Alloy but closer to standard logical notation; for example, we write $\forall x \in r\colon \varphi$ where Alloy would require `all [x : r] |` $\varphi$. To simplify the presentation, we also relax some arity restrictions on override, comprehension, and universal quantification. It is easy to encode our relaxed operators in Kodkod's slightly more restrictive logic.

The constraint NO $r$ states that $r$ is the empty relation, ONE $r$ states that $r$ is a singleton, LONE $r \iff$ NO $r \lor$ ONE $r$, and SOME $r \iff \neg$ NO $r$. The override, projection, and dot-join operators are also unconventional. Their semantics is given below:

$$
\begin{aligned}
[\![r \mathbin{++} s]\!] &= [\![s]\!] \cup \{\langle r_1,\ldots,r_m\rangle \mid \langle r_1,\ldots,r_m\rangle \in [\![r]\!] \land \nexists t.\ \langle r_1,\ldots,r_{m-1},t\rangle \in [\![s]\!]\}\\
[\![\pi_i^k(r)]\!] &= \{\langle r_i,\ldots,r_{i+k-1}\rangle \mid \langle r_1,\ldots,r_m\rangle \in [\![r]\!]\}\\
[\![r.s]\!] &= \{\langle r_1,\ldots,r_{m-1},s_2,\ldots,s_n\rangle \mid \exists t.\ \langle r_1,\ldots,r_{m-1},t\rangle \in [\![r]\!] \land \langle t,s_2,\ldots,s_n\rangle \in [\![s]\!]\}
\end{aligned}
$$

We write $\pi_i(r)$ for $\pi_i^1(r)$. If $r$ and $s$ are partial functions, the override $r \mathbin{++} s$ is the partial function that agrees with $s$ where $s$ is defined and with $r$ elsewhere.

The dot-join operator admits three important special cases. Let $s$ be unary and $r$, $r'$ be binary relations. The expression $s.r$ gives the direct image of the set $s$ under $r$; if $s$ is a singleton and $r$ a function, it coincides with the function application $r(s)$. Analogously, $r.s$ gives the inverse image of $s$ under $r$. Finally, $r.r'$ expresses the relational composition $r \circ r'$. To pass an $n$-tuple $s$ to a function $r$, we write $\langle s\rangle.r$, which stands for the $n$-fold dot-join $\pi_n(s).(\ldots.(\pi_1(s).r)\ldots)$; similarly, the abbreviation $r.\langle s\rangle$ stands for $(\ldots(r.\pi_n(s)).\ldots).\pi_1(s)$.

The relational operators often make it possible to express first-order problems concisely. The following FORL problem vainly tries to fit 30 pigeons into 29 holes:

$$
\begin{aligned}
&\textbf{var}\ \mathit{pigeons} = \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\}\\
&\textbf{var}\ \mathit{holes} = \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}\\
&\textbf{var}\ \emptyset \subseteq \mathit{nest} \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\} \times \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}\\
&\textbf{solve}\ (\forall p \in \mathit{pigeons}\colon \text{ONE}\ p.\mathit{nest}) \land (\forall h \in \mathit{holes}\colon \text{LONE}\ \mathit{nest}.h)
\end{aligned}
$$

The variables *pigeons* and *holes* are given fixed values, whereas *nest* is specified with a lower and an upper bound. Variable declarations are an extralogical way of specifying type constraints and partial solutions [187]. They also indirectly specify the variables' arities, which in turn determine the arities of all the terms.

The constraint ONE $p.\mathit{nest}$ states that pigeon $p$ is in relation with exactly one hole, and LONE $\mathit{nest}.h$ states that hole $h$ is in relation with at most one pigeon. Taken as a whole, the formula states that *nest* is an injective function. It is, of course, not satisfiable, a fact that Kodkod can establish in less than a second.

When reducing FORL to SAT, each $n$-ary relational variable $y$ is in principle translated to an $|\mathcal{A}|^n$ array of propositional variables $V[i_1,\ldots,i_n]$, with $V[i_1,\ldots,i_n] \iff \langle \mathsf{a}_{i_1},\ldots,\mathsf{a}_{i_n}\rangle \in y$. Most relational operations can be coded efficiently; for example, $\cup$ is simply $\lor$. The quantified formula $\forall r \in s\colon \varphi[r]$ is treated as $\bigwedge_{j=1}^{n} t_j \subseteq s \longrightarrow \varphi[t_j]$, where the $t_j$'s are the tuples that may belong to $s$. Transitive closure is simply unrolled to saturation, which is possible since all cardinalities are finite. Of course, the actual translation performed by Kodkod is considerably more sophisticated [187].

## 3.2 Basic Translations

Nitpick employs Kodkod to find a finite model (a satisfying assignment to the free variables and constants) of $A \land \neg C$, where $A$ is the conjunction of all relevant axioms and $C$ is the conjecture. The translation of a formula from HOL to FORL is parameterized by the cardinalities of the types occurring in it, provided as a function $|\ |$ from nonschematic types to cardinalities obeying

$$|\sigma| \geq 1 \qquad |o| = 2 \qquad |\sigma \to \tau| = |\tau|^{|\sigma|} \qquad |\sigma \times \tau| = |\sigma| \cdot |\tau|$$

In this chapter, we call such a function a *scope*.[1] Like Refute, Nitpick enumerates the possible cardinalities for each type, so that if a formula has a finite counterexample, the tool will eventually find it, unless it runs out of resources [198, §2.4.2]. Both tools reject goals involving schematic type variables, since refuting such goals requires considering infinitely many ground instantiations of the type variables.

To exhaust all models up to a cardinality bound $k$ for $n$ atomic types (types other than $o$, $\to$, and $\times$), a model finder must a priori iterate through $k^n$ combinations of cardinalities and consider all models for each of these combinations. This can be made more efficient if some of the types in the problem are monotonic (Chapter 4). Another option is to avoid hard-coding the exact cardinalities in the translation and let the SAT solver try all cardinalities up to a given bound; this is the Alloy Analyzer's normal mode of operation [94, §4.6].

### 3.2.1 A Sound and Complete Translation

We start by presenting a sound and complete translation from HOL to FORL, excluding the definitional principles. The translation is limited to finite domains, but for these it is sound and complete. It primarily serves as a stepping stone toward the more sophisticated translations of Sections 3.2.2 and 3.3, which are closer to Nitpick's actual implementation.

SAT solvers (Kodkod's backends) are particularly sensitive to the encoding of problems. Whenever practicable, HOL constants should be mapped to their FORL equivalents, rather than expanded to their definitions. This is especially true for the transitive closure $r^+$, which HOL defines (roughly) as

$$\mathsf{lfp}\ (\lambda R\ (x, y).\ (x, y) \in r \vee (\exists u.\ (x, u) \in R \wedge (u, y) \in r))$$

where $\mathsf{lfp}\ F$ is the least $R$ such that $F\ R \subseteq R$ if such an $R$ exists. The translation treats the following HOL constants specially:

| | | | |
|---|---|---|---|
| $\mathsf{False}^{o}$ | falsity | $\mathsf{insert}^{\alpha\to(\alpha\to o)\to\alpha\to o}$ | element insertion |
| $\mathsf{True}^{o}$ | truth | $\cup^{(\alpha\to o)\to(\alpha\to o)\to\alpha\to o}$ | union |
| $=^{\alpha\to\alpha\to o}$ | equality | $-^{(\alpha\to o)\to(\alpha\to o)\to\alpha\to o}$ | set difference |
| $\neg^{o\to o}$ | negation | $(\,,\,)^{\alpha\to\beta\to\alpha\times\beta}$ | pair constructor |
| $\wedge^{o\to o\to o}$ | conjunction | $\mathsf{fst}^{\alpha\times\beta\to\alpha}$ | first projection |
| $\forall^{(\alpha\to o)\to o}$ | universal quantifier | $\mathsf{snd}^{\alpha\times\beta\to\beta}$ | second projection |
| $\emptyset^{\alpha\to o}$ | empty set | $(\ )^{+\ (\alpha\times\alpha\to o)\to\alpha\times\alpha\to o}$ | transitive closure |
| $\mathsf{UNIV}^{\alpha\to o}$ | universal set | | |

Disjunction, equivalence, and existential quantification can be seen as abbreviations. Other constants are treated as if they were free variables of the problem, constrained by their (definitional or axiomatic) specification.

[1] In Chapter 2, we defined a scope as a function to nonempty sets (Definition 2.2). Here we abstract away the set elements, which are irrelevant [198, §2.3.2], and consider only finite cardinalities.

In general, an $n$-ary HOL function from and to atomic types can be encoded in FORL as an $(n+1)$-ary relation accompanied by a constraint. For example, given the scope $|\alpha| = 2$ and $|\beta| = 3$, the HOL conjecture $\forall x^{\alpha}.\ \exists y^{\beta}.\ f\,x = y$ corresponds to the (negated) FORL problem

$$\begin{aligned}&\textbf{var}\ \emptyset \subseteq f \subseteq \{\mathsf{a}_1, \mathsf{a}_2\} \times \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\}\\ &\textbf{solve}\ (\forall x \in \mathsf{a}_1 \cup \mathsf{a}_2\colon \text{ONE}\ x.f) \wedge \neg\,(\forall x \in \mathsf{a}_1 \cup \mathsf{a}_2\colon \exists y \in \mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5\colon x.f = y)\end{aligned}$$

The first conjunct ensures that $f$ is a function, and the second conjunct is the negation of the HOL conjecture translated to FORL. If the return type is $o$, the function is more efficiently coded as an unconstrained $n$-ary relation than as a constrained $(n+1)$-ary relation. This allows formulas such as $A^+ \cup B^+ = (A \cup B)^+$ to be translated without taking a detour through ternary relations.

Higher-order quantification and functions bring complications of their own. For example, we would like to translate the HOL assumption $\forall g^{\beta \to \alpha}.\ g\,y \neq x$ into something like

$$\forall g \subseteq (\mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5) \times (\mathsf{a}_1 \cup \mathsf{a}_2)\colon (\forall y \in \mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5\colon \text{ONE}\ y.g) \longrightarrow y.g \neq x$$

but since FORL is a first-order formalism, $\subseteq$ is not allowed at the binding site—only $\in$ is. Skolemization solves half of the problem (Section 3.4.3), but for the remaining quantifiers we are forced to adopt an unwieldy $n$-tuple singleton representation of functions, where $n$ is the cardinality of the domain. For the formula above, this gives

$$\forall G \in (\mathsf{a}_1 \cup \mathsf{a}_2)^3\colon\ y.\big(\overbrace{\mathsf{a}_3 \times \pi_1(G)\ \cup\ \mathsf{a}_4 \times \pi_2(G)\ \cup\ \mathsf{a}_5 \times \pi_3(G)}^{g}\big) \neq x$$

where $G$ is the triple corresponding to $g$. In the body, we convert the ternary singleton $G$ to its binary relational representation, then we apply $y$ on it using dot-join. The singleton (or tuple) encoding is also used for passing functions to functions; fortunately, two optimizations, function specialization and boxing (Section 3.4), make this rarely necessary.

We are now ready to look at the translation in more detail. The translation distinguishes between formulas (F), singletons (S), and general relations (R). We start by mapping HOL types to sets of FORL atom tuples. For each type $\sigma$, we provide two encodings, a singleton representation $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ and a relational representation $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$:

$$\begin{aligned}\mathsf{S}\langle\!\langle\sigma \to \tau\rangle\!\rangle &= \mathsf{S}\langle\!\langle\tau\rangle\!\rangle^{|\sigma|} & \mathsf{R}\langle\!\langle\sigma \to o\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\\ \mathsf{S}\langle\!\langle\sigma \times \tau\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{S}\langle\!\langle\tau\rangle\!\rangle & \mathsf{R}\langle\!\langle\sigma \to \tau\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{R}\langle\!\langle\tau\rangle\!\rangle\\ \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle &= \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\sigma|}\} & \mathsf{R}\langle\!\langle\sigma\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\end{aligned}$$

Both metafunctions depend on the scope. In the S-representation, an element of type $\sigma$ is mapped to a single tuple taken from the set $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$. In the R-representation, an element of type $\sigma \to o$ is mapped to a subset of $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ consisting of the points at which the predicate is true; an element of $\sigma \to \tau$ (where $\tau \neq o$) is mapped to a relation $\subseteq \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{R}\langle\!\langle\tau\rangle\!\rangle$; an element of any other type is coded as a singleton (S).

To simplify the presentation, we reuse the same atoms for distinct types. Doing so is sound for well-typed terms. However, our implementation in Nitpick assigns disjoint atom sets to distinct types so as to produce problems that are more amenable to symmetry breaking [62, 187].[1] Symmetry breaking can dramatically speed up model finding, especially for high cardinalities.

It is convenient to assume that free and bound variables are syntactically distinguishable (as in the locally nameless representation of $\lambda$-terms [157]), and we reserve the letter $y$ for the former and $b$ for the latter. Free variables can be coded using the natural $\mathsf{R}$ representation, whereas bound variables require the unwieldy $\mathsf{S}$ representation.

For each free variable $y^\sigma$, we produce the declaration $\textbf{var}\ \emptyset \subseteq y \subseteq \mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$ as well as a constraint $\Phi^\sigma(y)$ to ensure that functions are functions and scalars are singletons:

$$\begin{aligned}
\Phi^{\sigma_1\to\cdots\to\sigma_n\to o}(r) &= \textsc{true}\\
\Phi^{\sigma\to\tau}(r) &= \forall b_{\mathrm{f}} \in \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\colon \Phi^{\tau}\big(\langle b_{\mathrm{f}}\rangle\mathbin{.}r\big)\\
\Phi^{\sigma}(r) &= \textsc{one}\ r
\end{aligned}$$

The symbol $b_{\mathrm{f}}$ denotes a fresh bound variable.

We postulate a total order on atom tuples (e.g., the lexicographic order induced by $\mathsf{a}_i < \mathsf{a}_j \iff i < j$) and let $\mathsf{S}_i\langle\!\langle\sigma\rangle\!\rangle$ denote the $i$th tuple from $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ according to that order. Moreover, we define $s(\sigma)$ and $r(\sigma)$ as the arity of the tuples in $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ and $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$, respectively. The translation of terms requires the following rather technical conversions between formulas, singletons, and relations:

$$\begin{aligned}
\mathsf{s2f}(r) &= (r = \mathsf{a_t})\\
\mathsf{f2s}(\varphi) &= \textsc{if}\ \varphi\ \textsc{then}\ \mathsf{a_t}\ \textsc{else}\ \mathsf{a_f}\\
\mathsf{s2r}_\sigma(r) &= \begin{cases} \bigcup_{i=1}^{|\tau|} \pi_i(r)\mathbin{.}(\mathsf{a_t} \times \mathsf{S}_i\langle\!\langle\tau\rangle\!\rangle) & \text{if } \sigma = \tau \to o\\ \bigcup_{i=1}^{|\tau|} \mathsf{S}_i\langle\!\langle\tau\rangle\!\rangle \times \mathsf{s2r}_\upsilon\Big(\pi^{s(\upsilon)}_{(i-1)\cdot s(\upsilon)+1}(r)\Big) & \text{if } \sigma = \tau \to \upsilon\\ r & \text{otherwise}\end{cases}\\
\mathsf{r2s}_\sigma(r) &= \begin{cases} \{\langle b_{\mathrm{f}} \in \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\rangle \mid \mathsf{s2r}_\sigma(b_{\mathrm{f}}) = r\} & \text{if } \sigma = \tau \to \upsilon\\ r & \text{otherwise}\end{cases}
\end{aligned}$$

The functions $\mathsf{f2s}$ and $\mathsf{s2f}$ convert between singletons (atoms) and formulas, with the convention that the two distinct atoms $\mathsf{a_f}$, $\mathsf{a_t} \in \mathcal{A}$ encode FALSE and TRUE; $\mathsf{s2r}$ expands a relation to a tuple, and $\mathsf{r2s}$ reconstructs the relation.

The translation from HOL terms to formulas, singletons, and relations is performed by the functions $\mathsf{F}\langle\!\langle t^o\rangle\!\rangle$, $\mathsf{S}\langle\!\langle t^\sigma\rangle\!\rangle$, and $\mathsf{R}\langle\!\langle t^\sigma\rangle\!\rangle$, respectively. They implicitly depend on the scope. Their defining equations are matched sequentially modulo $\eta$-expansion in case not all arguments are supplied to the constants we treat specially:

$$\begin{aligned}
\mathsf{F}\langle\!\langle \mathsf{False}\rangle\!\rangle &= \textsc{false} & \mathsf{F}\langle\!\langle t \wedge u\rangle\!\rangle &= \mathsf{F}\langle\!\langle t\rangle\!\rangle \wedge \mathsf{F}\langle\!\langle u\rangle\!\rangle\\
\mathsf{F}\langle\!\langle \mathsf{True}\rangle\!\rangle &= \textsc{true} & \mathsf{F}\langle\!\langle \forall b^\sigma\!.\ t\rangle\!\rangle &= \forall b \in \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\colon \mathsf{F}\langle\!\langle t\rangle\!\rangle\\
\mathsf{F}\langle\!\langle t = u\rangle\!\rangle &= (\mathsf{R}\langle\!\langle t\rangle\!\rangle = \mathsf{R}\langle\!\langle u\rangle\!\rangle) & \mathsf{F}\langle\!\langle t\ u\rangle\!\rangle &= \mathsf{S}\langle\!\langle u\rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle t\rangle\!\rangle\\
\mathsf{F}\langle\!\langle \neg\, t\rangle\!\rangle &= \neg\,\mathsf{F}\langle\!\langle t\rangle\!\rangle & \mathsf{F}\langle\!\langle t\rangle\!\rangle &= \mathsf{s2f}(\mathsf{S}\langle\!\langle t\rangle\!\rangle)
\end{aligned}$$

[1] Because of bound declarations, which refer to atoms by name, FORL atoms are generally not interchangeable. Kodkod's symmetry breaker infers symmetries (classes of atoms that can be permuted with each other) from the bound declarations and generates additional constraints to rule out needless permutations [187].

$$\begin{aligned}
\mathsf{S}\langle\!\langle b\rangle\!\rangle &= b & \mathsf{S}\langle\!\langle \mathsf{snd}\ t^{\sigma\times\tau}\rangle\!\rangle &= \pi^{s(\tau)}_{s(\sigma)+1}(\mathsf{S}\langle\!\langle t\rangle\!\rangle)\\
\mathsf{S}\langle\!\langle (t,u)\rangle\!\rangle &= \mathsf{S}\langle\!\langle t\rangle\!\rangle \times \mathsf{S}\langle\!\langle u\rangle\!\rangle & \mathsf{S}\langle\!\langle t^{\sigma}\rangle\!\rangle &= \mathsf{r2s}_{\sigma}(\mathsf{R}\langle\!\langle t\rangle\!\rangle)\\
\mathsf{S}\langle\!\langle \mathsf{fst}\ t^{\sigma\times\tau}\rangle\!\rangle &= \pi^{s(\sigma)}_{1}(\mathsf{S}\langle\!\langle t\rangle\!\rangle)
\end{aligned}$$

$$\begin{aligned}
\mathsf{R}\langle\!\langle \mathsf{False}\rangle\!\rangle &= \mathsf{a_f} & \mathsf{R}\langle\!\langle \mathsf{insert}\ t\ u\rangle\!\rangle &= \mathsf{S}\langle\!\langle t\rangle\!\rangle \cup \mathsf{R}\langle\!\langle u\rangle\!\rangle\\
\mathsf{R}\langle\!\langle \mathsf{True}\rangle\!\rangle &= \mathsf{a_t} & \mathsf{R}\langle\!\langle t \cup u\rangle\!\rangle &= \mathsf{R}\langle\!\langle t\rangle\!\rangle \cup \mathsf{R}\langle\!\langle u\rangle\!\rangle\\
\mathsf{R}\langle\!\langle y\rangle\!\rangle &= y & \mathsf{R}\langle\!\langle t - u\rangle\!\rangle &= \mathsf{R}\langle\!\langle t\rangle\!\rangle - \mathsf{R}\langle\!\langle u\rangle\!\rangle\\
\mathsf{R}\langle\!\langle b^{\sigma}\rangle\!\rangle &= \mathsf{s2r}_{\sigma}(b) & \mathsf{R}\langle\!\langle (t^{\sigma\times\sigma\to o})^{+}\rangle\!\rangle &= \mathsf{R}\langle\!\langle t\rangle\!\rangle^{+} \quad \text{if } s(\sigma) = 1\\
\mathsf{R}\langle\!\langle (t,u)\rangle\!\rangle &= \mathsf{S}\langle\!\langle (t,u)\rangle\!\rangle & \mathsf{R}\langle\!\langle t^{\sigma\to o}\ u\rangle\!\rangle &= \mathsf{f2s}(\mathsf{F}\langle\!\langle t\ u\rangle\!\rangle)\\
\mathsf{R}\langle\!\langle \mathsf{fst}\ t^{\sigma\times\tau}\rangle\!\rangle &= \mathsf{s2r}_{\sigma}(\mathsf{S}\langle\!\langle \mathsf{fst}\ t\rangle\!\rangle) & \mathsf{R}\langle\!\langle t\ u\rangle\!\rangle &= \langle \mathsf{S}\langle\!\langle u\rangle\!\rangle\rangle \mathbin{.} \mathsf{R}\langle\!\langle t\rangle\!\rangle\\
\mathsf{R}\langle\!\langle \mathsf{snd}\ t^{\sigma\times\tau}\rangle\!\rangle &= \mathsf{s2r}_{\tau}(\mathsf{S}\langle\!\langle \mathsf{snd}\ t\rangle\!\rangle) & \mathsf{R}\langle\!\langle \lambda b^{\sigma}.\ t^{o}\rangle\!\rangle &= \{\langle b \in \mathsf{S}\langle\!\langle \sigma\rangle\!\rangle\rangle \mid \mathsf{F}\langle\!\langle t\rangle\!\rangle\}\\
\mathsf{R}\langle\!\langle \emptyset^{\sigma}\rangle\!\rangle &= \textsc{none}^{r(\sigma)} & \mathsf{R}\langle\!\langle \lambda b^{\sigma}.\ t^{\tau}\rangle\!\rangle &= \{\langle b \in \mathsf{S}\langle\!\langle \sigma\rangle\!\rangle,\ b_{\mathsf{f}} \in \mathsf{R}\langle\!\langle \tau\rangle\!\rangle\rangle \mid\\
\mathsf{R}\langle\!\langle \mathsf{UNIV}^{\sigma}\rangle\!\rangle &= \mathsf{R}\langle\!\langle \sigma\rangle\!\rangle & &\qquad b_{\mathsf{f}} \subseteq \mathsf{R}\langle\!\langle t\rangle\!\rangle\}
\end{aligned}$$

Annoyingly, the translation of transitive closure is defined only if $s(\sigma) = 1$. There are at least three ways to lift this restriction: Fall back on the (expensive) HOL definition in terms of lfp; treat the transitive closure like any other inductive predicate (Section 3.3.3); box the argument (Section 3.4.2).

Countermodels found by Kodkod must be mapped back to HOL before they can be presented to the user. The functions $\lceil t\rceil^{\sigma}_{\mathsf{S}}$ and $\lceil r\rceil^{\sigma}_{\mathsf{R}}$ translate an S-encoded tuple $t \in \mathsf{S}\langle\!\langle \sigma\rangle\!\rangle$ or an R-encoded relation $r \subseteq \mathsf{R}\langle\!\langle \sigma\rangle\!\rangle$ to a HOL term of type $\sigma$:

$$\begin{aligned}
\lceil\langle r_1,\ldots,r_m\rangle\rceil^{\sigma\to\tau}_{\mathsf{S}} &= \mathsf{undefined}^{\sigma\to\tau}(\mathsf{S}_1\langle\!\langle \sigma\rangle\!\rangle := \lceil r_1\rceil^{\tau}_{\mathsf{S}},\ \ldots,\ \mathsf{S}_m\langle\!\langle \sigma\rangle\!\rangle := \lceil r_m\rceil^{\tau}_{\mathsf{S}})\\
\lceil\langle r_1,\ldots,r_m\rangle\rceil^{\sigma\times\tau}_{\mathsf{S}} &= \big(\lceil\langle r_1,\ldots,r_{s(\sigma)}\rangle\rceil^{\sigma}_{\mathsf{S}},\ \lceil\langle r_{s(\sigma)+1},\ldots,r_m\rangle\rceil^{\tau}_{\mathsf{S}}\big)\\
\lceil\langle \mathsf{a}_i\rangle\rceil^{\sigma}_{\mathsf{S}} &= \mathsf{S}_i\langle\!\langle \sigma\rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
\lceil\{t_1,\ldots,t_n\}\rceil^{\sigma\to o}_{\mathsf{R}} &= \{\lceil t_1\rceil^{\sigma}_{\mathsf{S}},\ldots,\lceil t_n\rceil^{\sigma}_{\mathsf{S}}\}\\
\lceil r\rceil^{\sigma\to\tau}_{\mathsf{R}} &= \mathsf{undefined}^{\sigma\to\tau}(\mathsf{S}_1\langle\!\langle \sigma\rangle\!\rangle := \lceil \mathsf{S}_1\langle\!\langle \sigma\rangle\!\rangle.r\rceil^{\tau}_{\mathsf{R}},\ \ldots,\\
&\qquad\qquad \mathsf{S}_m\langle\!\langle \sigma\rangle\!\rangle := \lceil \mathsf{S}_m\langle\!\langle \sigma\rangle\!\rangle.r\rceil^{\tau}_{\mathsf{R}})\\
\lceil\{t\}\rceil^{\sigma}_{\mathsf{R}} &= \lceil t\rceil^{\sigma}_{\mathsf{S}}
\end{aligned}$$

Functions are expressed using Isabelle's function update notation: $f(a := b)$ is the function that maps $a$ to $b$ and that otherwise coincides with $f$. Successive updates $f(a_1 := b_1)\cdots(a_n := b_n)$ are displayed as $f(a_1 := b_1,\ \ldots,\ a_n := b_n)$. The unspecified function $\mathsf{undefined}^{\sigma\to\tau}$ works as a placeholder; since we override its definition at every point, any function would do.

The equations above abuse notation in two ways. First, they employ the FORL dot-join operator semantically, instead of syntactically. Second, they identify the value $\mathsf{S}_i\langle\!\langle \sigma\rangle\!\rangle$ with some term that denotes it; for uninterpreted types (such as nonschematic type variables and types introduced by **typedecl**), Nitpick chooses subscripted one-letter names based on the type (e.g., $\mathsf{b}_1$, $\mathsf{b}_2$, $\mathsf{b}_3$ for the type $\beta$).

For our proof sketches, we find it convenient to define the function $\lceil B\rceil^{o}_{\mathsf{F}}$ for FORL truth values $B$, with $\lceil\bot\rceil^{o}_{\mathsf{F}} = \mathsf{False}$ and $\lceil\top\rceil^{o}_{\mathsf{F}} = \mathsf{True}$. We also identify the terms returned by the $\lceil\ \rceil^{\sigma}_{\mathsf{X}}$ functions with the values they denote (which are independent of any variable environment).

**Example 3.1.** Given a cardinality $k$ for $\beta$, the HOL conjecture $\neg\,(P\ x^{\beta} \wedge \neg\,P\ y)$ is translated into the FORL problem

$$\begin{array}{l} \textbf{var}\ \emptyset \subseteq P, x, y \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_k\} \\ \textbf{solve}\ \textsc{one}\ x \wedge \textsc{one}\ y \wedge \neg\neg\,(x \subseteq P \wedge y \not\subseteq P) \end{array}$$

The first two conjuncts ensure that $x$ and $y$ are singletons (representing scalars), and the third conjunct is the translation of the negated conjecture. Two solutions already exist for $k = 2$, namely the valuation $P = \{\mathsf{a}_1\}$, $x = \{\mathsf{a}_1\}$, and $y = \{\mathsf{a}_2\}$ and its isomorph with $\mathsf{a}_1$ and $\mathsf{a}_2$ permuted. The first valuation corresponds to the HOL variable assignment $P = \{\mathsf{b}_1\}$, $x = \mathsf{b}_1$, and $y = \mathsf{b}_2$.

**Theorem 3.2 (Soundness and Completeness).** *Given a conjecture $P$ with free variables $y_1^{\sigma_1}, \ldots, y_m^{\sigma_m}$ within our HOL fragment and a scope, $P$ is falsifiable for the scope iff there exists a valuation $V$ with $V(y_i) \subseteq \mathsf{R}\langle\!\langle\sigma_i\rangle\!\rangle$ that satisfies the FORL formula $\mathsf{F}\langle\!\langle\neg\,P\rangle\!\rangle \wedge \bigwedge_{i=1}^{m} \Phi^{\sigma_i}(y_i)$. Moreover, the HOL variable assignment $y_i \mapsto \lceil y_i \rceil_{\mathsf{R}}^{\sigma_i}$ falsifies $P$ for the scope.*

*Proof sketch.* Let $[\![t]\!]_A$ denote the semantics of the HOL term $t$ with respect to a variable assignment $A$ and the given scope. Let $[\![\rho]\!]_V$ denote the semantics of the FORL term or formula $\rho$ with respect to a variable valuation $V$ and the scope.

SOUNDNESS (IF): Using well-founded induction, it is straightforward to prove that $\lceil [\![\mathsf{X}\langle\!\langle t^{\sigma}\rangle\!\rangle]\!]_V \rceil_{\mathsf{X}}^{\sigma} = [\![t]\!]_A$ if $\lceil V(y_i) \rceil_{\mathsf{R}}^{\sigma_i} = A(y_i)$ for all free variables $y_i$ and $\lceil V(b_i) \rceil_{\mathsf{S}}^{\sigma_i} = A(b_i)$ for all locally free bound variables $b_i$ occurring in $t$, where $\mathsf{X} \in \{\mathsf{F}, \mathsf{S}, \mathsf{R}\}$. Let $A(y_i) = \lceil V(y_i) \rceil_{\mathsf{R}}^{\sigma_i}$. Clearly, $A$ falsifies $P$: $\lceil [\![\mathsf{F}\langle\!\langle\neg\,P\rangle\!\rangle]\!]_V \rceil_{\mathsf{F}}^{o} = \top = [\![\neg\,P]\!]_A$. The $\Phi^{\sigma_i}(y_i)$ constraints and the variable bounds $V(y_i) \subseteq \mathsf{R}\langle\!\langle\sigma_i\rangle\!\rangle$ ensure that $\lceil V(y_i) \rceil_{\mathsf{R}}^{\sigma_i}$ is defined and yields a type-correct value.

COMPLETENESS (ONLY IF): For any HOL variable assignment $A$, we exhibit a FORL valuation $V$ such that $\lceil V(y_i) \rceil_{\mathsf{R}}^{\sigma_i} = A(y_i)$ for all $y_i$'s. $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ and $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$ are defined so that the tuples in $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ and the $\Phi^{\sigma}$-constrained subsets of $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$ are isomorphic to the elements of $\sigma$. The function $\lceil\ \rceil_{\mathsf{S}}^{\sigma}$ is clearly an isomorphism, and we can take its inverse to define $V$ in terms of $A$. □

### 3.2.2 Approximation of Infinite Types and Partiality

Besides its lack of explicit support for Isabelle's definitional principles, the above translation suffers from a serious limitation: It cannot handle infinite types such as natural numbers, lists, and trees, which are ubiquitous in real-world specifications. Fortunately, it is not hard to adapt the translation to take these into account in a sound way, if we give up completeness.

We first introduce a distinction between the *reference scope*, which allows infinite types, and an *analysis scope* used in the FORL translation. In Section 3.2.1, these two notions coincided. Given an infinite atomic type, our FORL translation considers a finite subset of it and represents every element not in this subset by a special undefined or unknown value $\star$. For the type *nat* of natural numbers, an obvious choice is to consider prefixes $\{0, \ldots, K\}$ of $\mathbb{N}$ and map numbers greater than $K$ to $\star$. Using this technique, the successor function $\mathsf{Suc}$ becomes partial, with $\mathsf{Suc}\ K = \star$. The same approach can also be used to speed up the analysis of finite types with a high cardinality: We can approximate a 256-value *byte* type by a subset of, say, 5 values, by letting $|byte| = 256$ in the reference scope and $|byte| = 5$ in the analysis scope. Since we are mostly concerned with the analysis scope, we usually call it "scope" and leave it implicit that the cardinality operator $|\sigma|$ refers to it.

We call types for which we can represent all the values in FORL without needing $\star$ *complete*; the other types are *incomplete*. Function types with an infinite domain are generally infinite and must be approximated, but the approximation has a different flavor: Functions such as $nat \to o$ are represented by partial functions from the subset $\{0, \ldots, K\}$ to $o$, with the convention that numbers beyond $K$ are mapped to $\star$. Since every function of the space $nat \to o$ can be represented as a function from $\{0, \ldots, K\}$ to $o$, the type is complete, but the ambiguity caused by the truncation of the domain prompts us to call such a type *abstract*, as opposed to *concrete*. Some types, such as $nat \to nat$, are both incomplete and abstract.

We can in principle choose how to treat finite atomic types, but it makes sense to consider types such as $o$ and $\alpha$ concrete and complete, whereas the infinite atomic type $nat$ approximated by $\{0, \ldots, K\}$ is concrete but incomplete. The following rules lift the two notions to functions and products:

$$\frac{\sigma \text{ is concrete} \qquad \tau \text{ is complete}}{\sigma \to \tau \text{ is complete}} \qquad \frac{\sigma \text{ is complete} \qquad \tau \text{ is concrete}}{\sigma \to \tau \text{ is concrete}}$$

$$\frac{\sigma \text{ is complete} \qquad \tau \text{ is complete}}{\sigma \times \tau \text{ is complete}} \qquad \frac{\sigma \text{ is concrete} \qquad \tau \text{ is concrete}}{\sigma \times \tau \text{ is concrete}}$$

Approximation means that the translation must cope with partiality. Constructors and other functions sometimes return the unknown value $\star$, which trickles down from terms all the way to the logical connectives and quantifiers. Moreover, equality for abstract types is problematic, because a single abstract value encodes several values and there is no way to distinguish them. At the formula level, some precision can be regained by adopting a three-valued Kleene logic [100], with such rules as $\star \lor \text{True} \iff \text{True}$ and $\star \land \text{False} \iff \text{False}$.

To ensure soundness, universal quantifiers whose bound variable ranges over an incomplete type, such as $\forall n^{nat}.\ P\ n$, will generally evaluate to either False (if $P\ n$ gives False for some $n \le K$) or $\star$, but never to True, since we cannot ascertain whether $P\ (K+1), P\ (K+2), \ldots$ are true except in special cases. By the same token, $A^{nat \to o} = B$ will generally evaluate to either False or $\star$. In view of this, Nitpick generally cannot falsify conjectures that contain an essentially existential (i.e., non-skolemizable) quantifier ranging over an infinite type. As a fallback, the tool enters an unsound mode in which the quantifiers are artificially bounded, similar to what Refute always does; counterexamples are then labeled "potentially spurious."

Partiality can be encoded in FORL as follows. Inside terms, we let the empty set (NONE) stand for $\star$. This choice is convenient because it is an absorbing element for the dot-join operator, which models function application; thus, $f \star = \star$ irrespective of $f$. Inside a formula, we keep track of polarities: In positive contexts (i.e., under an even number of negations), TRUE encodes True and FALSE encodes False or $\star$; in negative contexts, FALSE encodes False and TRUE encodes True or $\star$.

In the translation of Section 3.2.1, we conveniently identified HOL predicates with FORL relations. This is not desirable here, because predicates must now distinguish between three return values (True, False, and $\star$). Instead, we treat predicates as any other functions. This is reflected in the new definition of $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$:

$$\mathsf{R}\langle\!\langle\sigma \to \tau\rangle\!\rangle = \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{R}\langle\!\langle\tau\rangle\!\rangle \qquad\qquad \mathsf{R}\langle\!\langle\sigma\rangle\!\rangle = \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$$

($\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ is exactly as before.) With the R-representation, the approximated predicate $P^{\alpha\to o}$ that maps $\mathsf{a}_1$ to True, $\mathsf{a}_2$ to $\star$, and $\mathsf{a}_3$ to False would be encoded as the term $(\mathsf{a}_1 \times \mathsf{a}_\mathsf{t}) \cup (\mathsf{a}_2 \times \textsc{none}) \cup (\mathsf{a}_3 \times \mathsf{a}_\mathsf{f})$, which denotes the relation $\{\langle \mathsf{a}_1, \mathsf{a}_\mathsf{t}\rangle,\ \langle \mathsf{a}_3, \mathsf{a}_\mathsf{f}\rangle\}$. With the S-representation, $P$ would be encoded as $\mathsf{a}_\mathsf{t} \times \textsc{none} \times \mathsf{a}_\mathsf{f}$, which collapses to the (ternary) empty set, representing the fully unknown predicate; this is sound but loses all information about the predicate at points $\mathsf{a}_1$ and $\mathsf{a}_3$.

The auxiliary functions $\mathsf{s2r}_\sigma(r)$ and $\mathsf{r2s}_\sigma(r)$ are defined as in Section 3.2.1 except that predicates are now treated the same way as other functions (i.e., the first case in the definition of $\mathsf{s2r}_\sigma(r)$ is omitted). Both gracefully handle unknown values, returning the empty set if passed the empty set.

The translation function $\mathsf{F}^s\langle\!\langle t\rangle\!\rangle$ for formulas depends on the polarity $s$ (+ or −) of the formula. Taken together, the Boolean values of $\mathsf{F}^+\langle\!\langle t\rangle\!\rangle$ and $\mathsf{F}^-\langle\!\langle t\rangle\!\rangle$ encode a three-valued logic, with

$$\begin{array}{lll}
\langle \textsc{true},\ \textsc{true}\rangle & \text{denoting} & \mathsf{True} \\
\langle \textsc{false},\ \textsc{true}\rangle & \text{denoting} & \star \\
\langle \textsc{false},\ \textsc{false}\rangle & \text{denoting} & \mathsf{False}
\end{array}$$

The remaining case, $\langle \textsc{true},\ \textsc{false}\rangle$, is impossible by construction. The conversion functions between FORL formulas and atoms must now take the polarity into account. Observe in particular that $\mathsf{f2s}^+(\textsc{none})$ is false, whereas $\mathsf{f2s}^-(\textsc{none})$ is true:

$$\begin{array}{ll}
\mathsf{s2f}^+(r) = (r = \mathsf{a}_\mathsf{t}) & \qquad \mathsf{f2s}^+(\varphi) = \textsc{if}\ \varphi\ \textsc{then}\ \mathsf{a}_2\ \textsc{else none} \\
\mathsf{s2f}^-(r) = (r \neq \mathsf{a}_\mathsf{f}) & \qquad \mathsf{f2s}^-(\varphi) = \textsc{if}\ \neg\,\varphi\ \textsc{then}\ \mathsf{a}_1\ \textsc{else none}
\end{array}$$

The translation function $\mathsf{F}^s\langle\!\langle t\rangle\!\rangle$ is defined as follows:

$$\begin{array}{rl}
\mathsf{F}^s\langle\!\langle \mathsf{False}\rangle\!\rangle = \textsc{false} & \qquad \mathsf{F}^+\langle\!\langle \neg\, t\rangle\!\rangle = \neg\,\mathsf{F}^-\langle\!\langle t\rangle\!\rangle \\
\mathsf{F}^s\langle\!\langle \mathsf{True}\rangle\!\rangle = \textsc{true} & \qquad \mathsf{F}^-\langle\!\langle \neg\, t\rangle\!\rangle = \neg\,\mathsf{F}^+\langle\!\langle t\rangle\!\rangle \\
\mathsf{F}^s\langle\!\langle t^{\sigma\to\tau} = u\rangle\!\rangle = \mathsf{F}^s\langle\!\langle \forall b_\mathrm{f}^\sigma.\ t\ b_\mathrm{f} = u\ b_\mathrm{f}\rangle\!\rangle & \qquad \mathsf{F}^+\langle\!\langle \forall b^\sigma.\ t\rangle\!\rangle = \textsc{false}\ \text{ if } \sigma \text{ is incomplete} \\
\mathsf{F}^+\langle\!\langle t^\sigma = u\rangle\!\rangle = \textsc{false}\ \text{ if } \sigma \text{ is abstract} & \qquad \mathsf{F}^s\langle\!\langle \forall b^\sigma.\ t\rangle\!\rangle = \forall b \in \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\colon \mathsf{F}^s\langle\!\langle t\rangle\!\rangle \\
\mathsf{F}^+\langle\!\langle t = u\rangle\!\rangle = \textsc{some}\ (\mathsf{R}\langle\!\langle t\rangle\!\rangle \cap \mathsf{R}\langle\!\langle u\rangle\!\rangle) & \qquad \mathsf{F}^+\langle\!\langle t\ u^\sigma\rangle\!\rangle = \mathsf{S}\langle\!\langle u\rangle\!\rangle \not\subseteq \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle - \mathsf{R}\langle\!\langle t\rangle\!\rangle.\mathsf{a}_\mathsf{t} \\
\mathsf{F}^-\langle\!\langle t = u\rangle\!\rangle = \textsc{lone}\ (\mathsf{R}\langle\!\langle t\rangle\!\rangle \cup \mathsf{R}\langle\!\langle u\rangle\!\rangle) & \qquad \mathsf{F}^-\langle\!\langle t\ u^\sigma\rangle\!\rangle = \mathsf{S}\langle\!\langle u\rangle\!\rangle \subseteq \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle - \mathsf{R}\langle\!\langle t\rangle\!\rangle.\mathsf{a}_\mathsf{f} \\
\mathsf{F}^s\langle\!\langle t \wedge u\rangle\!\rangle = \mathsf{F}^s\langle\!\langle t\rangle\!\rangle \wedge \mathsf{F}^s\langle\!\langle u\rangle\!\rangle & \qquad \mathsf{F}^s\langle\!\langle t\rangle\!\rangle = \mathsf{f2s}^s(\mathsf{S}\langle\!\langle t\rangle\!\rangle)
\end{array}$$

Many of the equations deserve some justification:

- $\mathsf{F}^s\langle\!\langle t^{\sigma\to\tau} = u\rangle\!\rangle$: The equation exploits extensionality to avoid comparing approximated functions directly.
- $\mathsf{F}^+\langle\!\langle t^\sigma = u\rangle\!\rangle$ for abstract $\sigma$: Even if $t$ and $u$ evaluate to the same value, if they are of an abstract type it is generally impossible to tell whether they both correspond to the same concrete value.
- $\mathsf{F}^+\langle\!\langle t = u\rangle\!\rangle$: $\mathsf{R}\langle\!\langle t\rangle\!\rangle$ and $\mathsf{R}\langle\!\langle u\rangle\!\rangle$ are either singletons or empty sets. If the intersection of $\mathsf{R}\langle\!\langle t\rangle\!\rangle$ and $\mathsf{R}\langle\!\langle u\rangle\!\rangle$ is nonempty, they must be equal singletons, meaning that $t$ and $u$ must be equal.

- $\mathsf{F}^-\langle\!\langle t = u \rangle\!\rangle$: This case is the dual of the preceding case. If the union of $\mathsf{R}\langle\!\langle t \rangle\!\rangle$ and $\mathsf{R}\langle\!\langle u \rangle\!\rangle$ has more than one element, then $t$ and $u$ must be unequal.
- $\mathsf{F}^+\langle\!\langle \forall b^\sigma.\ t \rangle\!\rangle$ for incomplete $\sigma$: Positive occurrences of universal quantification can never yield True if the bound variable ranges over an incomplete type. (In negative contexts, approximation compromises the translation's completeness but not its soundness.)
- $\mathsf{F}^+\langle\!\langle t\ u^\sigma \rangle\!\rangle$: It is tempting to put $\mathsf{S}\langle\!\langle u \rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_t}$ on the right-hand side but this would be unsound when $\mathsf{S}\langle\!\langle u \rangle\!\rangle$ yields the empty set. Our version correctly returns FALSE (meaning $\star$) in that case and is equivalent to $\mathsf{S}\langle\!\langle u \rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_t}$ when $\mathsf{S}\langle\!\langle u \rangle\!\rangle$ is a singleton.

The $\mathsf{S}\langle\!\langle t \rangle\!\rangle$ equations are exactly as in Section 3.2.1 and are omitted here. On the other hand, the $\mathsf{R}\langle\!\langle t \rangle\!\rangle$ equations concerned with predicates must be adapted:

$$\begin{aligned}
\mathsf{R}\langle\!\langle \emptyset^{\sigma\to o} \rangle\!\rangle &= \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle \times \mathsf{a_f} \\
\mathsf{R}\langle\!\langle \mathsf{UNIV}^{\sigma\to o} \rangle\!\rangle &= \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle \times \mathsf{a_t} \\
\mathsf{R}\langle\!\langle \mathsf{insert}\ t\ u \rangle\!\rangle &= \mathsf{R}\langle\!\langle u \rangle\!\rangle \mathbin{++} (\mathsf{S}\langle\!\langle t \rangle\!\rangle \times \mathsf{a_t}) \\
\mathsf{R}\langle\!\langle t \cup u \rangle\!\rangle &= ((\mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_t} \cup \mathsf{R}\langle\!\langle u \rangle\!\rangle.\mathsf{a_t}) \times \mathsf{a_t}) \cup ((\mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_f} \cap \mathsf{R}\langle\!\langle u \rangle\!\rangle.\mathsf{a_f}) \times \mathsf{a_f}) \\
\mathsf{R}\langle\!\langle t - u \rangle\!\rangle &= ((\mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_t} \cap \mathsf{R}\langle\!\langle u \rangle\!\rangle.\mathsf{a_f}) \times \mathsf{a_t}) \cup ((\mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_f} \cup \mathsf{R}\langle\!\langle u \rangle\!\rangle.\mathsf{a_t}) \times \mathsf{a_f}) \\
\mathsf{R}\langle\!\langle (t^{\sigma\times\sigma\to o})^+ \rangle\!\rangle &= ((\mathsf{S}\langle\!\langle \sigma \rangle\!\rangle - (\mathsf{S}\langle\!\langle \sigma \rangle\!\rangle - \mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_f})^+) \times \mathsf{a_f}) \mathbin{++} ((\mathsf{R}\langle\!\langle t \rangle\!\rangle.\mathsf{a_t})^+ \times \mathsf{a_t}) \\
&\quad\ \text{if } s(\sigma) = 1 \\
\mathsf{R}\langle\!\langle t^{\sigma\to o}\ u \rangle\!\rangle &= \text{IF } \mathsf{F}^+\langle\!\langle t\ u \rangle\!\rangle \text{ THEN } \mathsf{a_t} \text{ ELSE IF } \neg\, \mathsf{F}^-\langle\!\langle t\ u \rangle\!\rangle \text{ THEN } \mathsf{a_f} \text{ ELSE NONE}
\end{aligned}$$

We must relax the definition of the $\Phi^\sigma(y)$ constraint to account for unknown values, by substituting LONE for ONE. We also no longer treat predicates specially:

$$\Phi^{\sigma\to\tau}(r) = \forall b_\mathsf{f} \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle\colon \Phi^\tau(\langle b_\mathsf{f} \rangle.r) \qquad\qquad \Phi^\sigma(r) = \text{LONE } r$$

As before, countermodels found by Kodkod must be mapped back to HOL. To cope with partiality, we extend the HOL term syntax with $\star$ terms. The semantic of an extended HOL term is a nonempty set of values. Countermodels involving $\star$ describe not a single countermodel, but rather a family of countermodels where each occurrence of $\star$ can take an arbitrary type-correct value.

The functions $\lceil s \rceil^\sigma_\mathsf{S}$ and $\lceil r \rceil^\sigma_\mathsf{R}$ translate an S-encoded singleton or empty set $s \subseteq \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle$ or an R-encoded relation $r \subseteq \mathsf{R}\langle\!\langle \sigma \rangle\!\rangle$ to an extended HOL term of type $\sigma$:

$$\begin{aligned}
\lceil \emptyset \rceil^\sigma_\mathsf{S} &= \star^\sigma \\
\lceil \{\langle r_1, \ldots, r_m \rangle\} \rceil^{\sigma\to\tau}_\mathsf{S} &= \star^{\sigma\to\tau}(\mathsf{S}_1\langle\!\langle \sigma \rangle\!\rangle := \lceil r_1 \rceil^\tau_\mathsf{S}, \ldots, \mathsf{S}_m\langle\!\langle \sigma \rangle\!\rangle := \lceil r_m \rceil^\tau_\mathsf{S}) \\
\lceil \{\langle r_1, \ldots, r_m \rangle\} \rceil^{\sigma\times\tau}_\mathsf{S} &= (\lceil \{\langle r_1, \ldots, r_{s(\sigma)} \rangle\} \rceil^\sigma_\mathsf{S}, \lceil \{\langle r_{s(\sigma)+1}, \ldots, r_m \rangle\} \rceil^\tau_\mathsf{S}) \\
\lceil \{\langle \mathsf{a}_i \rangle\} \rceil^\sigma_\mathsf{S} &= \mathsf{S}_i\langle\!\langle \sigma \rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
\lceil r \rceil^{\sigma\to\tau}_\mathsf{R} &= \star^{\sigma\to\tau}(\mathsf{S}_1\langle\!\langle \sigma \rangle\!\rangle := \lceil \mathsf{S}_1\langle\!\langle \sigma \rangle\!\rangle.r \rceil^\tau_\mathsf{R}, \ldots, \mathsf{S}_m\langle\!\langle \sigma \rangle\!\rangle := \lceil \mathsf{S}_m\langle\!\langle \sigma \rangle\!\rangle.r \rceil^\tau_\mathsf{R}) \\
\lceil s \rceil^\sigma_\mathsf{R} &= \lceil s \rceil^\sigma_\mathsf{S}
\end{aligned}$$

We identify extended terms with the nonempty sets of values they denote.

**Example 3.3.** Given a cardinality $k$ for $\beta$, the HOL conjecture $\neg\,(P\ x^\beta \wedge \neg\, P\ y)$ is translated into the FORL problem

$$
\begin{aligned}
&\textbf{var}\ \emptyset \subseteq\ P\ \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_k\} \times \{\mathsf{a}_\mathsf{f}, \mathsf{a}_\mathsf{t}\}\\
&\textbf{var}\ \emptyset \subseteq x, y \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}\\
&\textbf{solve}\ \textsc{lone}\ x \wedge \textsc{lone}\ y \wedge (\forall z \in \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}.\ \textsc{lone}\ z.P) \wedge\\
&\qquad \neg\neg\,(x \not\subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_k\} - P.\mathsf{a}_\mathsf{t} \wedge y \not\subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_k\} - P.\mathsf{a}_\mathsf{f})
\end{aligned}
$$

Two solutions exist for $k = 2$: the valuation $P = \{\langle \mathsf{a}_1, \mathsf{a}_\mathsf{t}\rangle, \langle \mathsf{a}_2, \mathsf{a}_\mathsf{f}\rangle\}$, $x = \{\mathsf{a}_1\}$, and $y = \{\mathsf{a}_2\}$ and its isomorph with $\mathsf{a}_1$ and $\mathsf{a}_2$ permuted. Using extended HOL terms, the first valuation corresponds to the HOL variable assignment $P = \star(\mathsf{b}_1 := \mathsf{True}, \mathsf{b}_2 := \mathsf{False})$ (i.e., the set $\{\mathsf{b}_1\}$), $x = \mathsf{b}_1$, and $y = \mathsf{b}_2$.

**Example 3.4.** Given a finite cardinality $k$ for the (incomplete) type *nat*, the conjecture $\forall x^{nat}.\ x = y$ is translated into

$$
\begin{aligned}
&\textbf{var}\ \emptyset \subseteq y \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}\\
&\textbf{solve}\ \textsc{lone}\ y \wedge \neg\,(\forall x \in \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}.\ \textsc{lone}\ (x \cup y))
\end{aligned}
$$

Again, a solution exists for $k = 2$, falsifying the conjecture.

In the face of partiality, the new encoding is sound but no longer complete.

**Theorem 3.5 (Soundness).** *Given a conjecture $P$ with free variables $y_1^{\sigma_1}, \ldots, y_m^{\sigma_m}$ within our HOL fragment and a reference scope, $P$ is falsifiable for the scope if there exists a valuation $V$ with $V(y_i) \subseteq \mathsf{R}\langle\!\langle \sigma_i \rangle\!\rangle$ that satisfies the FORL formula $\mathsf{F}^+\langle\!\langle \neg P \rangle\!\rangle \wedge \bigwedge_{i=1}^{m} \Phi^{\sigma_i}(y_i)$ generated using an analysis scope.*

*Proof sketch.* The proof is similar to the soundness part of Theorem 3.2, but partiality and the polarity-based encoding of a three-valued logic introduce some complications. Using well-founded induction, we can prove that if $A(y_i) \in \lceil V(y_i) \rceil^{\sigma_i}_{\mathsf{R}}$ for all free variables $y_i$ and $A(b_i) \in \lceil V(b_i) \rceil^{\sigma_i}_{\mathsf{S}}$ for all locally free bound variables $b_i$ occurring in $t$, then

1. $[\![\mathsf{F}^+\langle\!\langle t^o \rangle\!\rangle]\!]_V = \top \implies [\![t]\!]_A = \top$
2. $[\![\mathsf{F}^-\langle\!\langle t^o \rangle\!\rangle]\!]_V = \bot \implies [\![t]\!]_A = \bot$
3. $[\![t]\!]_M \in \lceil [\![X\langle\!\langle t \rangle\!\rangle]\!]_V \rceil^{\sigma}_{X}$ for $X \in \{\mathsf{S}, \mathsf{R}\}$

In particular, if $\mathsf{F}^+\langle\!\langle \neg P \rangle\!\rangle = \top$, then $[\![\neg P]\!]_A = \top$ for any variable assignment such that $A(y_i) \in \lceil V(y_i) \rceil^{\sigma_i}_{\mathsf{R}}$. □

## 3.3 Translation of Definitional Principles

Although our translation is sound, a lot of precision is lost when translating equality and quantification for approximated types. By handling high-level definitional principles specially (as opposed to directly translating their HOL specification), we can bypass the imprecise translation and increase the precision.

### 3.3.1 Simple Definitions

If the conjecture to falsify refers to defined constants, we must take the relevant definitions into account. For simple definitions $\mathsf{c} \equiv t$, earlier versions of Nitpick (like Refute) unfolded (i.e., inlined) the constant's definition, substituting the right-hand side $t$ for the constant $\mathsf{c}$ wherever it appears in the problem. This notion was clearly misguided: If a constant is used many times and its definition is large, expanding it each time hinders reuse. Kodkod often detects shared subterms and factors them out [187, §4.3], but this does not help if Nitpick's translation phase is overwhelmed by the large terms.

The alternative is to translate HOL constants introduced by **definition** to FORL variables and conjoin the definition with the problem as an additional constraint to satisfy. More precisely, if $\mathsf{c}^{\tau}$ is defined and an instance $\mathsf{c}^{\tau'}$ occurs in a formula, we must conjoin *c*'s definition with the formula, instantiating $\tau$ with $\tau'$. This process must be repeated for any defined constants occurring in c's definition. It will eventually terminate, since cyclic definitions are disallowed. If several type instances of the same constant are needed, they must be given distinct names in the translation.

Given the command

**definition** $\mathsf{c}^{\tau}$ **where** $\mathsf{c}\ \bar{x} = t$

the naive approach would be to conjoin $\mathsf{F}^{+}\langle\!\langle \forall \bar{x}.\ \mathsf{c}\ \bar{x} = t \rangle\!\rangle$ with the FORL formula to satisfy and recursively do the same for any defined constants in $t$. However, there are two issues with this:

- If any of the variables $\bar{x}$ is of an incomplete type, the equation $\mathsf{F}^{+}\langle\!\langle \forall \bar{x}.\ t \rangle\!\rangle = \text{FALSE}$ applies, and the axiom becomes unsatisfiable. This is sound but extremely imprecise, as it prevents the discovery of any model.
- Otherwise, the body of $\forall \bar{x}.\ \mathsf{c}\ \bar{x} = t$ is translated to $\text{SOME}\ (\mathsf{R}\langle\!\langle \mathsf{c}\ \bar{x} \rangle\!\rangle \cap \mathsf{R}\langle\!\langle t \rangle\!\rangle)$ (assuming $\tau$ is an atomic type), which evaluates to FALSE (meaning $\star$) whenever $\mathsf{R}\langle\!\langle t \rangle\!\rangle$ is NONE for some values of $\bar{x}$.

Fortunately, we can take a shortcut and translate the definition directly to the following FORL axiom, bypassing $\mathsf{F}^{+}$ altogether (cf. Weber [198, p. 66]):

$$\forall x_1 \in \mathsf{S}\langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_n \in \mathsf{S}\langle\!\langle \sigma_n \rangle\!\rangle\colon\ \mathsf{R}\langle\!\langle \mathsf{c}\ x_1\ \ldots\ x_n \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle$$

We must also declare the variable using appropriate bounds for the constant's type.

**Example 3.6.** The conjecture $\mathsf{K}\ a\ a = a^{\alpha}$, where K is defined as

**definition** $\mathsf{K}^{\alpha\to\beta\to\alpha}$ **where** $\mathsf{K}\ x\ y = x$

is translated to

**var** $\emptyset \subseteq a \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\}$
**var** $\emptyset \subseteq \mathsf{K} \subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\}^3$
**solve** $\text{LONE}\ a\ \wedge\ \neg\ \text{LONE}\ (a.(a.\mathsf{K}) \cup a)\ \wedge\ \forall x, y \in \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\}\colon y.(x.\mathsf{K}) = x$

In the translation, K's schematic type variables $\alpha$ and $\beta$ are instantiated with the (unrelated) nonschematic type variable $\alpha$ from the conjecture.

**Theorem 3.7.** *The encoding of Section 3.2.2 extended with simple definitions is sound.*

*Proof sketch.* Any FORL valuation $V$ that satisfies the FORL axiom associated with a constant c can be extended into a HOL constant model $M$ that satisfies the corresponding HOL axiom, by setting $M(\mathsf{c})(v) = [\![\lambda x_1 \ldots x_n.\, t]\!]_M(v)$ for any value $v$ at which $V(\mathsf{c})$ is not defined (either because $v$ is $\star$ in FORL or because the partial function $V(\mathsf{c})$ is not defined at that point). The apparent circularity in $M(\mathsf{c})(v) = [\![\lambda x_1 \ldots x_n.\, t]\!]_M(v)$ is harmless, because simple definitions are required to be acyclic and so we can construct $M$ one constant at a time. □

Incidentally, we obtain a simpler (and still sound) SAT encoding by replacing the $=$ operator with $\subseteq$ in the encoding of simple definitions. (Kodkod expands $r = s$ to $r \subseteq s \wedge s \subseteq r$.) Any entry of a defined constant's relation table that is not needed to construct the model can then be $\star$, even if the right-hand side is not $\star$.

### 3.3.2 Inductive Datatypes and Recursive Functions

In contrast to Isabelle's constructor-oriented treatment of inductive datatypes, the FORL axiomatization revolves around selectors and discriminators, inspired by Kuncak and Jackson's modeling of lists and trees in Alloy [107]. Let

$$\kappa = \mathsf{C}_1\ \sigma_{11}\ \ldots\ \sigma_{1n_1}\ \mid\ \cdots\ \mid\ \mathsf{C}_\ell\ \sigma_{\ell 1}\ \ldots\ \sigma_{\ell n_\ell}$$

be a datatype instance. With each constructor $\mathsf{C}_i$ we associate a discriminator $\mathsf{D}_i^{\kappa\to o}$ and $n_i$ selectors $S_{ik}^{\kappa\to\sigma_{ik}}$ obeying the laws

$$\mathsf{D}_j\ (\mathsf{C}_i\ x_1\ \ldots\ x_n) = (i = j) \qquad\qquad \mathsf{S}_{ik}\ (\mathsf{C}_i\ x_1\ \ldots\ x_n) = x_k$$

For example, the type *α list* is assigned the discriminators nilp and consp and the selectors head and tail:[1]

$$\begin{array}{rrr} \mathsf{nilp}\ \mathsf{Nil} = \mathsf{True} & \mathsf{consp}\ \mathsf{Nil} = \mathsf{False} & \mathsf{head}\ (\mathsf{Cons}\ x\ xs) = x \\ \mathsf{nilp}\ (\mathsf{Cons}\ x\ xs) = \mathsf{False} & \mathsf{consp}\ (\mathsf{Cons}\ x\ xs) = \mathsf{True} & \mathsf{tail}\ (\mathsf{Cons}\ x\ xs) = xs \end{array}$$

The discriminator and selector view almost always results in a more efficient SAT encoding than the constructor view because it breaks high-arity constructors into several low-arity discriminators and selectors. These are declared as

$$\textbf{var}\ \emptyset \subseteq \mathsf{D}_i \subseteq \mathsf{S}\langle\!\langle\kappa\rangle\!\rangle \qquad\qquad \textbf{var}\ \emptyset \subseteq \mathsf{S}_{ik} \subseteq \mathsf{R}\langle\!\langle\kappa \to \sigma_{ik}\rangle\!\rangle$$

for all possible $i$, $k$. For efficiency, the predicates $\mathsf{D}_i$ are directly coded as sets of atoms rather than as functions to $\{\mathsf{a_f}, \mathsf{a_t}\}$.

Let $\mathsf{C}_i(r_1, \ldots, r_n)$ stand for $\mathsf{S}_{i1}.\langle r_1\rangle \cap \cdots \cap \mathsf{S}_{in}.\langle r_n\rangle$ if $n \geq 1$, and $\mathsf{C}_i() = \mathsf{D}_i$ for parameterless constructors. Intuitively, $\mathsf{C}_i(r_1, \ldots, r_n)$ represents the constructor $\mathsf{C}_i$ with arguments $r_1, \ldots, r_n$ at the FORL level [70]. A faithful axiomatization of data-

[1] These names were chosen for readability; any fresh names would do. The generated selectors head and tail should not be confused with the similar HOL constants hd and tl defined in Isabelle's theory of lists.

types in terms of $\mathsf{D}_i$ and $\mathsf{S}_{ik}$ involves the following axioms (for all possible $i$, $j$, $k$):

$$\begin{aligned}
\textsc{Disjoint}_{ij}\text{:}\quad & \textsc{no}\ (\mathsf{D}_i \cap \mathsf{D}_j) \quad \text{for } i < j \\
\textsc{Exhaustive}\text{:}\quad & \mathsf{D}_1 \cup \cdots \cup \mathsf{D}_\ell = \mathsf{S}\langle\!\langle \kappa \rangle\!\rangle \\
\textsc{Selector}_{ik}\text{:}\quad & \forall y \in \mathsf{S}\langle\!\langle \kappa \rangle\!\rangle\text{: } \textsc{if}\ y \subseteq \mathsf{D}_i\ \textsc{then}\ \Psi^{\sigma_{ik}}(y.\mathsf{S}_{ik})\ \textsc{else no}\ y.\mathsf{S}_{ik} \\
\textsc{Unique}_i\text{:}\quad & \forall x_1 \in \mathsf{S}\langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_{n_i} \in \mathsf{S}\langle\!\langle \sigma_{n_i} \rangle\!\rangle\text{: } \textsc{lone}\ \mathsf{C}_i(x_1, \ldots, x_{n_i}) \\
\textsc{Generator}_i\text{:}\quad & \forall x_1 \in \mathsf{S}\langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_{n_i} \in \mathsf{S}\langle\!\langle \sigma_{n_i} \rangle\!\rangle\text{: } \textsc{some}\ \mathsf{C}_i(x_1, \ldots, x_{n_i}) \\
\textsc{Acyclic}\text{:}\quad & \textsc{no}\ (\mathsf{sup}_\kappa \cap \textsc{iden}).
\end{aligned}$$

In the SELECTOR axioms, the $\Psi^\sigma(r)$ constraint is identical to $\Phi^\sigma(r)$ except with ONE instead of LONE. In the ACYCLIC axiom, $\mathsf{sup}_\kappa$ denotes the proper superterm relation for $\kappa$. For example, we have $\langle \mathsf{Cons}(x, xs),\ xs \rangle \in \mathsf{sup}_\kappa$ for any $x$ and $xs$ because the second component's value is a proper subterm of the first argument's value. We will see shortly how to derive $\mathsf{sup}_\kappa$.

DISJOINT and EXHAUSTIVE ensure that the discriminators partition $\mathsf{S}\langle\!\langle \kappa \rangle\!\rangle$. The four remaining axioms, called the *SUGA axioms* (after the first letter of each axiom name), ensure that selectors are functions whose domain is given by the corresponding discriminator (SELECTOR), that constructors are total functions (UNIQUE and GENERATOR), and that datatype values cannot be proper subterms or superterms of themselves (ACYCLIC). The injectivity of constructors follows from the functionality of selectors.

With this axiomatization, occurrences of $\mathsf{C}_i\ u_1\ \ldots\ u_n$ in HOL are simply mapped to $\mathsf{C}_i(\mathsf{S}\langle\!\langle u_1 \rangle\!\rangle, \ldots, \mathsf{S}\langle\!\langle u_n \rangle\!\rangle)$, whereas

$$\text{case } t \text{ of } \mathsf{C}_1\ \bar{x}_1 \Rightarrow u_1 \mid \ldots \mid \mathsf{C}_\ell\ \bar{x}_\ell \Rightarrow u_\ell^\tau$$

is translated to

$$\begin{aligned}
&\textsc{if}\ \mathsf{S}\langle\!\langle t \rangle\!\rangle \subseteq \mathsf{D}_1\ \textsc{then}\ \mathsf{R}\langle\!\langle u_1^\bullet \rangle\!\rangle \\
&\textsc{else if}\ \ldots \\
&\textsc{else if}\ \mathsf{S}\langle\!\langle t \rangle\!\rangle \subseteq \mathsf{D}_\ell\ \textsc{then}\ \mathsf{R}\langle\!\langle u_\ell^\bullet \rangle\!\rangle \\
&\textsc{else none}^{r(\tau)}
\end{aligned}$$

where $u_i^\bullet$ denotes the term $u_i$ in which all occurrences of the variables $\bar{x}_i = x_{i1}, \ldots, x_{in_i}$ are replaced with the corresponding selector expressions $\mathsf{S}_{i1}(t), \ldots, \mathsf{S}_{in_i}(t)$.

Unfortunately, the SUGA axioms admit no finite models if the type $\kappa$ is recursive (and hence infinite), because they force the existence of infinitely many values. The solution is to leave GENERATOR out, yielding *SUA*. The SUA axioms characterize precisely the subterm-closed finite substructures of an inductive datatype.

Omitting GENERATOR is generally unsound in a two-valued logic, but Kuncak and Jackson [107] showed that it is sound for *existential–bounded-universal* (*EBU*) formulas (i.e., formulas whose prenex normal forms contain no unbounded universal quantifiers ranging over datatypes). In our three-valued setting, omitting GENERATOR is always sound if we consider the datatype as incomplete. The construct $\mathsf{C}_i(r_1, \ldots, r_{n_i})$ sometimes returns NONE for non-NONE arguments, but this is not a problem since our translation of Section 3.2.2 is designed to cope with partiality. Non-EBU formulas such as $\mathsf{True} \vee (\forall n^{nat}.\ P\ n)$ become analyzable when moving to a three-valued logic. This is especially important for complex specifications, because they are likely to contain irrelevant non-EBU parts.

**Example 3.8.** The *nat list* instance of *α list* would be axiomatized as follows:

$$\begin{aligned}
\textsc{Disjoint}&: \ \textsc{no}\ (\mathsf{nilp} \cap \mathsf{consp})\\
\textsc{Exhaustive}&: \ \mathsf{nilp} \cup \mathsf{consp} = \mathsf{S}\langle\!\langle \mathit{nat\ list}\rangle\!\rangle\\
\textsc{Selector}_{\mathsf{head}}&: \ \forall \mathit{ys}{\in}\mathsf{S}\langle\!\langle \mathit{nat\ list}\rangle\!\rangle\!: \textsc{if}\ \mathit{ys} \subseteq \mathsf{consp}\ \textsc{then one}\ \mathit{ys}.\mathsf{head}\ \textsc{else no}\ \mathit{ys}.\mathsf{head}\\
\textsc{Selector}_{\mathsf{tail}}&: \ \forall \mathit{ys}{\in}\mathsf{S}\langle\!\langle \mathit{nat\ list}\rangle\!\rangle\!: \textsc{if}\ \mathit{ys} \subseteq \mathsf{consp}\ \textsc{then one}\ \mathit{ys}.\mathsf{tail}\ \textsc{else no}\ \mathit{ys}.\mathsf{tail}\\
\textsc{Unique}_{\mathsf{Nil}}&: \ \textsc{lone}\ \mathsf{Nil}()\\
\textsc{Unique}_{\mathsf{Cons}}&: \ \forall x \in \mathsf{S}\langle\!\langle \mathit{nat}\rangle\!\rangle,\ \mathit{xs} \in \mathsf{S}\langle\!\langle \mathit{nat\ list}\rangle\!\rangle\!:\ \textsc{lone}\ \mathsf{Cons}(x,\ \mathit{xs})\\
\textsc{Acyclic}&: \ \textsc{no}\ (\mathsf{sup}_{\mathit{nat\ list}} \cap \textsc{iden}) \qquad \text{with } \mathsf{sup}_{\mathit{nat\ list}} = \mathsf{tail}^{+}
\end{aligned}$$

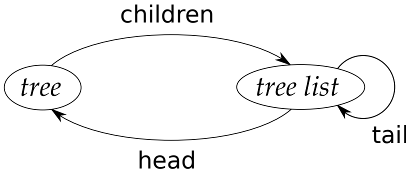
Examples of subterm-closed list substructures using traditional notation are $\{[],\ [0],\ [1]\}$ and $\{[],\ [1],\ [2,1],\ [0,2,1]\}$. In contrast, $L = \{[],\ [1,1]\}$ is not subterm-closed, because tail $[1,1] = [1] \notin L$. Given a cardinality, Kodkod systematically enumerates all corresponding subterm-closed list substructures.

To generate the proper superterm relation needed for ACYCLIC, we must consider the general case of mutually recursive datatypes. We start by computing the datatype dependency graph, in which vertices are labeled with datatypes and edges with selectors. For each selector $\mathsf{S}^{\kappa \to \kappa'}$, we add an edge from $\kappa$ to $\kappa'$ labeled S. Next, we compute for each datatype a regular expression capturing the nontrivial paths in the graph from the datatype to itself. This can be done using Kleene's construction [101; 105, pp. 51–53]. The proper superterm relation is obtained from the regular expression by replacing concatenation with relational composition, alternative with set union, and repetition with transitive closure.

**Example 3.9.** Let *sym* be an uninterpreted type, and consider the declarations

**datatype** *α list* = Nil | Cons *α* (*α list*)
**datatype** *tree* = Leaf *sym* | Node (*tree list*)

Their dependency graph is



where children is the selector associated with Node. The superterm relations are

$$\mathsf{sup}_{\mathit{tree}} = (\mathsf{children}.\mathsf{tail}^{*}.\mathsf{head})^{+} \qquad \mathsf{sup}_{\mathit{tree\ list}} = (\mathsf{tail} \cup \mathsf{head}.\mathsf{children})^{+}$$

Notice that in the presence of polymorphism, instances of sequentially declared datatypes can be mutually recursive.

Our account of datatypes would be incomplete without suitable completeness and concreteness rules:

$$\frac{\sigma_{11} \text{ is complete} \quad \cdots \quad \sigma_{\ell n_\ell} \text{ is complete} \quad |\kappa| = \sum_{i=1}^{\ell} \prod_{j=1}^{n_j} |\sigma_{ij}|}{\kappa \text{ is complete}}$$

$$\frac{\sigma_{11} \text{ is concrete} \quad \cdots \quad \sigma_{\ell n_\ell} \text{ is concrete}}{\kappa \text{ is concrete}}$$

Notice that the condition $|\kappa| = \sum_{i=1}^{\ell} \prod_{j=1}^{n_j} |\sigma_{ij}|$ always holds in the reference scope but not necessarily in the analysis scope, which interests us here. In particular, for recursive datatypes $\kappa$ we necessarily have an inequality $<$, and for nonrecursive datatypes any of $\{<, =, >\}$ is possible. In the $>$ case, the SUA axioms are unsatisfiable; Nitpick detects this condition and simply skips the scope.

With a suitable axiomatization of datatypes as subterm-closed substructures, it is easy to encode **primrec** definitions. A recursive equation

$$f\ (\mathsf{C}_i\ x_1^{\sigma_1}\ \ldots\ x_m^{\sigma_m})\ z^{\upsilon} = t^{\tau}$$

(with, for simplicity, exactly one nonrecursive argument) is translated to

$$\forall y \in \mathsf{D}_i,\ z \in \mathsf{S}\langle\!\langle \upsilon \rangle\!\rangle\colon\ \mathsf{R}\langle\!\langle f\ y\ z \rangle\!\rangle = \mathsf{R}\langle\!\langle t^{\bullet} \rangle\!\rangle$$

where $t^{\bullet}$ is obtained from $t$ by replacing the variables $x_i$ with the selector expressions $\mathsf{S}_i(y)$. By quantifying over the constructed values $y$ rather than over the arguments to the constructors, we reduce the number of copies of the quantified body by a factor of $|\sigma_1| \cdot \ldots \cdot |\sigma_m|\ /\ |\kappa|$ in the SAT problem. Although we focus here on primitive recursion, general well-founded recursion with non-overlapping pattern matching (as defined using Isabelle's function package [106]) can be handled in essentially the same way.

**Example 3.10.** The recursive function @ from Section 2.3 is translated to

$$\forall ys \in \mathsf{nilp},\ zs \in \mathsf{S}\langle\!\langle \alpha\ list \rangle\!\rangle\colon\ zs.(ys.@) = zs$$
$$\forall ys \in \mathsf{consp},\ zs \in \mathsf{S}\langle\!\langle \alpha\ list \rangle\!\rangle\colon\ zs.(ys.@) = \mathsf{Cons}(ys.\mathsf{head},\ zs.((ys.\mathsf{tail}).@))$$

**Theorem 3.11.** *The encoding of Section 3.3.1 extended with inductive datatypes and primitive recursion is sound.*

*Proof sketch.* Kuncak and Jackson [107] proved that the SUA axioms precisely describe subterm-closed finite substructures of an inductive datatype, and sketched how to generalize this result to mutually recursive datatypes. This means that we can always extend the valuation of the SUA-specified descriptors and selectors to obtain a model of the entire datatype. For recursion, we can prove

$$[\![f\ (\mathsf{C}_i\ x_1\ \ldots\ x_m)\ z]\!]_M \in \lceil [\![\mathsf{R}\langle\!\langle f\ (\mathsf{C}_i\ x_1\ \ldots\ x_m)\ z \rangle\!\rangle]\!]_V \rceil_{\mathsf{R}}^{\tau}$$

by structural induction on the value of the first argument to $f$ and extend $f$'s model as in the proof sketch of Theorem 3.7, exploiting the injectivity of constructors. □

### 3.3.3 Inductive and Coinductive Predicates

With datatypes and recursion in place, we are ready to consider inductive and coinductive predicates. Isabelle lets users specify (co)inductive predicates $\mathsf{p}$ by their introduction rules (Section 2.3) and synthesizes a fixed-point definition $\mathsf{p} = \mathsf{lfp}\ F$ or $\mathsf{p} = \mathsf{gfp}\ F$ for some term $F$. For performance reasons, Nitpick avoids expanding $\mathsf{lfp}$ and $\mathsf{gfp}$ to their definitions and translates (co)inductive predicates directly, using appropriate FORL concepts.

A first intuition is that an inductive predicate $\mathsf{p}$ is a fixed point, so we could use the fixed-point equation $\mathsf{p} = F\ \mathsf{p}$ as the axiomatic specification of $p$. In general, this is unsound since it underspecifies $\mathsf{p}$, but there are two important cases for which this method is sound.

First, if the recursion in $\mathsf{p} = F\ \mathsf{p}$ is well-founded, the equation admits exactly one solution [86, §3]; we can safely use it as $\mathsf{p}$'s specification and encode it the same way as a recursive function (Section 3.3.2). To ascertain wellfoundedness, we could perform a simple syntactic check to ensure that each recursive call peels off at least one constructor, but we prefer to invoke an off-the-shelf termination prover (Isabelle's *lexicographic_order* tactic [50]). Given introduction rules of the form

$$\frac{M_{i1}\ (\mathsf{p}\ \bar{t}_{i1}) \quad \cdots \quad M_{in_i}\ (\mathsf{p}\ \bar{t}_{in_i}) \quad Q_i}{\mathsf{p}\ \bar{u}_i}$$

for $i \in \{1, \ldots, m\}$, where the $M_{ij}$'s are optional monotonic operators and the $Q_i$'s are optional side conditions, the termination prover attempts to exhibit a well-founded relation $R$ such that

$$\textstyle\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n_i}\ \big(Q_i \longrightarrow \langle \bar{t}_{ij},\, \bar{u}_i \rangle \in R\big)$$

In our experience, about half of the inductive predicates occurring in practice are well-founded. This includes most type systems and other compositional formalisms, but generally excludes state transition systems.

Second, if $p$ is inductive and occurs negatively in the formula, we can replace these occurrences by a fresh constant $\mathsf{q}$ satisfying $\mathsf{q} = F\ \mathsf{q}$. The resulting formula is equisatisfiable to the original formula: Since $\mathsf{p}$ is a least fixed point, $\mathsf{q}$ overapproximates $\mathsf{p}$ and thus $\neg\ \mathsf{q}\ \bar{x} \Longrightarrow \neg\ \mathsf{p}\ \bar{x}$. Dually, this method can also handle positive occurrences of coinductive predicates.

To deal with positive occurrences of generally non-well-founded inductive predicates, we adapt a technique from bounded model checking [24]: We replace these occurrences of $\mathsf{p}$ by a fresh predicate $\mathsf{r}_k$ defined by the HOL equations

$$\mathsf{r}_0 = (\lambda \bar{x}.\ \mathsf{False}) \qquad\qquad \mathsf{r}_{\mathsf{Suc}\ m} = F\ \mathsf{r}_m$$

which corresponds to $\mathsf{p}$ unrolled $k$ times. In essence, we have made the predicate well-founded by introducing a counter that decreases with each recursive call. The above equations are primitive recursive over *nat* and can be translated using the approach shown in Section 3.3.2. The situation is mirrored for coinductive predicates: Negative occurrences are replaced by the overapproximation $\mathsf{r}_k$ defined by

$$\mathsf{r}_0 = (\lambda \bar{x}.\ \mathsf{True}) \qquad\qquad \mathsf{r}_{\mathsf{Suc}\ m} = F\ \mathsf{r}_m$$

The translation is sound for any $k$, but it is generally more precise when $k$ is larger. On the other hand, the relation table for $\mathsf{r}_k$ is $k$ times larger than that of $\mathsf{p}$ directly encoded as $\mathsf{p} = F\ \mathsf{p}$. If $k$ is the cardinality of $\mathsf{p}$'s tuple of arguments in the analysis scope, the predicate is unrolled to saturation—the point beyond which incrementing $k$ has no effect. (If each unrolling contributes at least one new element, after $k$ iterations $\mathsf{p}$ must be uniformly true; and if an iteration $i < k$ contributes no element, a fixed point has been reached.) By default, Nitpick gradually increments $k$ together with the cardinalities of the types that occur in the problem, using the cardinality of the tuple of arguments as an upper bound.

**Example 3.12.** The even predicate defined by

$$\begin{array}{l}\textbf{inductive}\ \mathsf{even}^{nat \to o}\ \textbf{where} \\ \mathsf{even}\ 0 \\ \mathsf{even}\ n \implies \mathsf{even}\ n \\ \mathsf{even}\ n \implies \mathsf{even}\ (\mathsf{Suc}\ (\mathsf{Suc}\ n))\end{array}$$

is not well-founded because of the (superfluous) cyclic rule $\mathsf{even}\ n \implies \mathsf{even}\ n$. The fixed-point equation

$$\mathsf{even}\ x\ =\ \big(\exists n.\ x = 0\ \vee\ (x = n \wedge \mathsf{even}\ n)\ \vee\ (x = \mathsf{Suc}\ (\mathsf{Suc}\ n) \wedge \mathsf{even}\ n)\big)$$

overapproximates even in negative contexts. Positive contexts require unrolling:

$$\begin{aligned}\mathsf{even}_0\ x\ &=\ \mathsf{False} \\ \mathsf{even}_{\mathsf{Suc}\ m}\ x\ &=\ \big(\exists n.\ x = 0 \vee (x = n \wedge \mathsf{even}_m\ n) \vee (x = \mathsf{Suc}\ (\mathsf{Suc}\ n) \wedge \mathsf{even}_m\ n)\big)\end{aligned}$$

**Theorem 3.13.** *The encoding of Section 3.3.2 extended with* (*co*)*inductive predicates is sound.*

*Proof sketch.* We consider only inductive predicates; coinduction is dual. If $p$ is well-founded, the fixed-point equation fully characterizes $\mathsf{p}$ [86, §3], and the proof is identical to that of primitive recursion in Theorem 3.11 but with well-founded induction instead of structural induction. If $\mathsf{p}$ is not well-founded, $\mathsf{q} = F\ \mathsf{q}$ is satisfied by several $\mathsf{q}$'s, and by Knaster–Tarski $\mathsf{p} \sqsubseteq \mathsf{q}$. Substituting $\mathsf{q}$ for $\mathsf{p}$'s negative occurrences in the FORL formula strengthens it, which is sound. For the positive occurrences, we have $\mathsf{r}_0 \sqsubseteq \cdots \sqsubseteq \mathsf{r}_k \sqsubseteq \mathsf{p}$ by monotonicity of the definition; substituting $\mathsf{r}_k$ for $\mathsf{p}$'s positive occurrences strengthens the formula. □

As an alternative to the explicit unrolling, Nitpick mobilizes FORL's transitive closure for an important class of inductive predicates, which we call *linear inductive predicates* and whose introduction rules are of the form

$$\frac{Q}{\mathsf{p}\ \bar{u}}\ \text{(the \textit{base rules})} \qquad \text{or} \qquad \frac{\mathsf{p}\ \bar{t}\quad Q}{\mathsf{p}\ \bar{u}}\ \text{(the \textit{step rules})}$$

The idea is to replace positive occurrences of $\mathsf{p}\ \bar{x}$ with

$$\exists \bar{x}_0.\ \bar{x}_0 \in \mathsf{p}_{\text{base}} \wedge (\bar{x}_0, \bar{x}) \in \mathsf{p}^*_{\text{step}}$$

where $\bar{x}_0 \in \mathsf{p}_{\text{base}}$ iff $\mathsf{p}\ \bar{x}_0$ can be deduced from a base rule, $(\bar{x}_0, \bar{x}) \in \mathsf{p}_{\text{step}}$ iff $\mathsf{p}\ \bar{x}$ can be deduced by applying one step rule assuming $\mathsf{p}\ \bar{x}_0$, and $\mathsf{p}^*_{\text{step}}$ is the reflexive transitive closure of $\mathsf{p}_{\text{step}}$. The approach is not so different from explicit unrolling, since Kodkod internally unrolls the transitive closure to saturation. Nonetheless, on some problems the transitive closure is considerably faster, presumably because Kodkod unfolds the relation in place.

### 3.3.4 Coinductive Datatypes and Corecursive Functions

Coinductive datatypes are similar to inductive datatypes, but they allow infinite values. For example, the infinite lists $[0, 0, \ldots]$ and $[0, 1, 2, 3, \ldots]$ are possible values of the type *nat llist* of coinductive, or lazy, lists over natural numbers.

Nitpick supports coinductive datatypes, even though Isabelle does not yet provide a high-level mechanism for defining them. Users can define custom coinductive datatypes from first principles and tell Nitpick to substitute its efficient FORL axiomatization for their definitions. Nitpick also recognizes Isabelle's lazy list datatype $\alpha\ \mathit{llist}$, with the constructors $\mathsf{LNil}^{\alpha\ \mathit{llist}}$ and $\mathsf{LCons}^{\alpha \to \alpha\ \mathit{llist} \to \alpha\ \mathit{llist}}$.

In principle, we could use the same SUA axiomatization for codatatypes as for datatypes (Section 3.3.2). This would exclude all infinite values but nevertheless be sound. However, in practice, infinite values often behave in surprising ways; excluding them would also exclude many interesting models.

We can alter the SUA axiomatization to support an important class of infinite values, namely those that are $\omega$-regular. For lazy lists, this means lasso-shaped objects such as $[0, 0, \ldots]$ and $[8, 1, 2, 1, 2, \ldots]$ (where the cycle $1, 2$ is repeated infinitely).

The first step is to leave out the ACYCLIC axiom. However, doing only this is unsound, because we might get several atoms encoding the same value; for example,

$$\mathsf{a}_1 = \mathsf{LCons}\ 0\ \mathsf{a}_1 \qquad\qquad \mathsf{a}_2 = \mathsf{LCons}\ 0\ \mathsf{a}_3 \qquad\qquad \mathsf{a}_3 = \mathsf{LCons}\ 0\ \mathsf{a}_2$$

all encode the infinite list $[0, 0, \ldots]$. This violates the bisimilarity principle, according to which two values are equal unless they lead to different observations (the observations here being $0, 0, \ldots$).

For lazy lists, we translate the following coinductive bisimulation principle along with the HOL formula:

$$\frac{}{\mathsf{LNil} \sim \mathsf{LNil}} \qquad \frac{x = x' \quad xs \sim xs'}{\mathsf{LCons}\ x\ xs \sim \mathsf{LCons}\ x'\ xs'}$$

To ensure that distinct atoms correspond to observably distinct lists, we require that $=$ coincides with $\sim$ on $\alpha\ \mathit{llist}$ values.

More generally, we generate mutual coinductive definitions of $\sim$ for all the codatatypes. For each constructor $\mathsf{C}^{\sigma_1 \to \cdots \to \sigma_n \to \sigma}$, we add an introduction rule

$$\frac{x_1 \approx_1 x_1' \quad \cdots \quad x_n \approx_n x_n'}{\mathsf{C}\ x_1\ \ldots\ x_n \sim \mathsf{C}\ x_1'\ \ldots\ x_n'}$$

where $\approx_i$ is $\sim^{\sigma_i \to \sigma_i \to o}$ if $\sigma_i$ is a codatatype and $=$ otherwise. Finally, for each codatatype $\kappa$, we add the axiom

$$\textsc{Bisimilar}\text{:}\ \ \forall y, y' \in \mathsf{S}\langle\!\langle \kappa \rangle\!\rangle\text{: } y \sim y' \longrightarrow y = y'$$

The completeness and concreteness rules are as for inductive datatypes.

The apparent lack of duality between SUA and SUB is dictated by efficiency concerns. For finite model finding, the ACYCLIC axiom (together with EXHAUSTIVE) approximates the standard higher-order induction principle (which quantifies over a predicate $P$), but it can be expressed more comfortably in FORL. Similarly, BISIMILAR is more efficient than the dual of the induction principle.

With the *SUB axioms* (SU plus BISIMILAR) in place, it is easy to encode **coprimrec** definitions. A corecursive equation $\mathsf{f}\ y_1^{\sigma_1}\ \ldots\ y_n^{\sigma_1} = t$ is translated to

$$\forall y_1 \in \mathsf{S}\langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, y_n \in \mathsf{S}\langle\!\langle \sigma_n \rangle\!\rangle\text{: } \mathsf{R}\langle\!\langle \mathsf{f}\ y_1\ \ldots\ y_n \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle$$

**Theorem 3.14.** *The encoding of Section 3.3.3 extended with coinductive datatypes and primitive corecursion is sound.*

*Proof sketch.* Codatatypes are characterized by selectors, which are axiomatized by the SU axioms, and by finality, which is equivalent to the bisimilarity principle [96, 148]. Our finite axiomatization gives a subterm-closed substructure of the coinductive datatype, which can be extended to yield a HOL model of the complete codatatype, as we did for inductive datatypes in the proof sketch of Theorem 3.11.

The soundness of the encoding of primitive corecursion is proved by coinduction. Given the equation $\mathsf{f}\,\bar{y} = t^{\tau}$, assuming that for each corecursive call $\mathsf{f}\,\bar{x}$ we have $[\![\mathsf{f}\,\bar{x}]\!]_M \in \lceil [\![\mathsf{R}\langle\!\langle \mathsf{f}\,\bar{x} \rangle\!\rangle]\!]_V \rceil^{\tau}_{\mathsf{R}}$, we must show that $[\![\mathsf{f}\,\bar{y}]\!]_M \in \lceil [\![\mathsf{R}\langle\!\langle \mathsf{f}\,\bar{y} \rangle\!\rangle]\!]_V \rceil^{\tau}_{\mathsf{R}}$. This follows from the soundness of the encoding of the constructs occurring in $t$ and from the hypotheses. □

## 3.4 Optimizations

While our translation of definitional principles can be seen as a form of optimization, many more ad hoc optimizations are necessary to increase the tool's scalability in practical applications, especially in the presence of higher-order constructs. In particular, the early experiments leading to the case study presented in Chapter 5 shed some light on areas in earlier versions of Nitpick that could benefit from more optimizations—although we had come up with workarounds, we decided to improve the tool in the hope that future applications would require less manual work.

### 3.4.1 Function Specialization

A function argument is *static* if it is passed unaltered to all recursive calls. A typical example is $f$ in the equational specification of $\mathsf{map}$:

$$\mathsf{map}\ f\ [] = [] \qquad\qquad \mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

An optimization reminiscent of the static argument transformation, or $\lambda$-dropping [65, pp. 148–156], is to specialize the function for each eligible call site, thereby avoiding passing the static argument altogether. At the call site, any term whose free variables are all globally free is eligible for this optimization. Following this scheme, $\mathsf{map}\ (\lambda n.\ n - 1)\ ns$ becomes $\mathsf{map}'\ ns$, where $\mathsf{map}'$ is defined as follows:

$$\mathsf{map}'\ [] = [] \qquad\qquad \mathsf{map}'\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (x - 1)\ (\mathsf{map}'\ xs)$$

For this example, specialization reduces the number of propositional variables that encode the function by a factor of $|nat|^{|nat|}$.

### 3.4.2 Boxing

Nitpick normally translates function and product types directly to the homologous Kodkod concepts. This is not always desirable; for example, a transition relation on states represented as $n$-tuples leads to a $2n$-ary relation, which gives rise to a combinatorial explosion and precludes the use of FORL's binary transitive closure. In such cases, it can be advantageous to approximate functions and products by substructures, similarly to (co)inductive datatypes.

Nitpick approximates a function or a product by wrapping it in a (conceptually isomorphic) datatype $\alpha$ *box* with the single constructor $\mathsf{Box}^{\alpha \to \alpha\ box}$ and associated selector $\mathsf{unbox}^{\alpha\ box \to \alpha}$. The translation must then insert constructor and selector calls as appropriate to convert between the raw type and the boxed type. For example, assuming that function specialization is disabled and that we want to box map's function argument, the second equation for map would become

$$\mathsf{map}\ f^{(nat \to nat)\ box}\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (\mathsf{unbox}\ f\ x)\ (\mathsf{map}\ f\ xs)$$

with $\mathsf{map}\ (\mathsf{Box}\ (\lambda n.\ n - 1))\ ns$ at the call site. Because of approximation, the Box constructor may return $\star$ for non-$\star$ arguments; in exchange, $f^{(nat \to nat)\ box}$ ranges only over a fragment of the function space. For function types, boxing is similar to defunctionalization [17], with selectors playing the role of "apply" functions.

Boxing stands in a delicate trade-off between precision and efficiency. There are situations where the complete function or product space is needed to find a model and others where a small fragment suffices. By default, Nitpick approximates unspecialized higher-order arguments as well as $n$-tuples where $n \geq 3$. Users can override this behavior by specifying which types to box and not to box.

### 3.4.3 Quantifier Massaging

(Co)inductive definitions are marred by existential quantifiers, which blow up the size of the resulting propositional formula. The following steps are applied to eliminate quantifiers or reduce their binding range:

1. Replace quantifications of the forms

$$\forall x.\ x = t \longrightarrow P\ x \qquad \forall x.\ x \neq t \vee P\ x \qquad \exists x.\ x = t \wedge P\ x$$

   by $P\ t$ if $x$ does not occur free in $t$.

2. Skolemize.

3. Distribute quantifiers over congenial connectives ($\forall$ over $\wedge$; $\exists$ over $\vee$, $\longrightarrow$).

4. For any remaining subformula $Qx_1 \ldots x_n.\ p_1 \otimes \cdots \otimes p_m$, where $Q$ is a quantifier and $\otimes$ is a connective, move the $p_i$'s out of as many quantifiers as possible by rebuilding the formula using $\mathit{qfy}(\{x_1, \ldots, x_n\},\ \{p_1, \ldots, p_m\})$, defined by the equations

$$\begin{aligned} \mathit{qfy}(\emptyset,\ P) &= \bigotimes P \\ \mathit{qfy}(x \uplus X,\ P) &= \mathit{qfy}(X,\ P - P_x \cup \{Qx.\ \bigotimes P_x\}) \end{aligned}$$

   where $P_x = \{p \in P \mid x \text{ occurs free in } p\}$.

The order in which individual variables $x$ are removed from the first argument in step 4 is crucial because it affects which $p_i$'s can be moved out. For clusters of up to 7 quantifiers, Nitpick considers all permutations of the bound variables and chooses the one that minimizes the sum

$$\sum_{i=1}^{m} |\tau_{i1}| \cdot \ldots \cdot |\tau_{ik_i}| \cdot \mathit{size}(p_i)$$

where $\tau_{i1}, \ldots, \tau_{ik_i}$ are the types of the variables that have $p_i$ in their binding range, and $\mathit{size}(p_i)$ is a rough syntactic measure of $p_i$'s size; for larger clusters, it falls back on a heuristic inspired by Paradox's clause splitting procedure [62]. Thus,

$$\exists x^{\alpha}\; y^{\alpha}.\; p\; x \wedge q\; x\; y \wedge r\; y\; (f\; y\; y)$$

is rewritten to

$$\exists y^{\alpha}.\; r\; y\; (f\; y\; y) \wedge (\exists x^{\alpha}.\; p\; x \wedge q\; x\; y)$$

Processing $y$ before $x$ in *qfy* would instead give

$$\exists x^{\alpha}.\; p\; x \wedge (\exists y^{\alpha}.\; q\; x\; y \wedge r\; y\; (f\; y\; y))$$

which is more expensive because $r\; y\; (f\; y\; y)$, the most complex conjunct, is doubly quantified and hence $|\alpha|^2$ copies of it are needed in the resulting SAT problem.

### 3.4.4 Alternative Definitions

Isabelle's predefined theories use various constructions to define basic types and constants, many of which are inefficient for SAT solving. A preprocessor replaces selected HOL constants and types with optimized constructs. For example, it replaces occurrences of $\exists! x.\; P\; x$, which is defined as $\exists x.\; P\; x \wedge (\forall y.\; P\; y \longrightarrow y = x)$, with $\exists x.\; P = \{x\}$, and it substitutes a normalized pair representation of rationals and reals for Isabelle's set quotient construction and Dedekind cuts.

### 3.4.5 Tabulation

FORL relations can be assigned fixed values, in which case no propositional variables are generated for them. We can use this facility to store tables that precompute the value of basic operations on natural numbers, such as $\mathsf{Suc}$, $+$, $-$, $*$, $\mathsf{div}$, $\mathsf{mod}$, $<$, $\mathsf{gcd}$, and $\mathsf{lcm}$. This is possible for natural numbers because for any cardinality $k$ there exists exactly one subterm-closed substructure, namely $\{0, 1, \ldots, k-1\}$.

For example, if *nat* is approximated by $\{0, 1, 2, 3, 4\}$, we encode each number $n < 5$ as $\mathsf{a}_{n+1}$, effectively performing our own symmetry breaking. The Suc function is then specified as follows:

$$\textbf{var}\; \mathit{Suc} = \{\langle \mathsf{a}_1, \mathsf{a}_2\rangle,\; \langle \mathsf{a}_2, \mathsf{a}_3\rangle,\; \langle \mathsf{a}_3, \mathsf{a}_4\rangle,\; \langle \mathsf{a}_4, \mathsf{a}_5\rangle\}$$

### 3.4.6 Heuristic Constant Unfolding

When Nitpick meets a constant $\mathsf{c}$ introduced by a simple definition $\mathsf{c} \equiv t$, it has the choice between conjoining the equation to the the problem as an additional constraint to satisfy (Section 3.3.1) and unfolding (i.e., inlining) the definition.

Refute and early versions of Nitpick followed a simplistic approach:

- For simple definitions $\mathsf{c} \equiv t$, they unfolded $\mathsf{c}$'s definition, substituting the right-hand side $t$ for $\mathsf{c}$ wherever it appears in the problem.
- Constants introduced using **primrec**, **fun**, or **function**, which define recursive functions in terms of user-supplied equations in the style of functional programming, were translated to FORL variables, and their equational specifications were conjoined with the problem as additional constraints to satisfy.

This approach works reasonably well for specifications featuring a healthy mixture of simple definitions and recursive functions, but it falls apart when there are too many nested simple definitions. Simple definitions are generally of the form $\mathsf{c} \equiv (\lambda x_1 \ldots x_n.\ u)$, and when c is applied to arguments, these are substituted for $x_1, \ldots, x_n$. The formula quickly explodes if the $x_i$'s occur many times in the body $u$. Kodkod often detects shared subterms and factors them out [187, §4.3], but it does not help if Nitpick's translation phase is overwhelmed by the large terms.

We introduced a simple heuristic based on the size of the right-hand side of a definition: Small definitions are unfolded, whereas larger ones are kept as equations. It is difficult to design a better heuristic because of hard-to-predict interactions with function specialization and other optimizations occurring at later stages. We also made the unfolding more compact by heuristically introducing 'let's to bind the arguments of a constant when it is unfolded.

To give the user complete control, Nitpick provides a *nitpick_simp* attribute that can be attached to a simple definition or to any other theorem of the right form to prevent unfolding as well as a *nitpick_unfold* attribute that can be used to forcefully unfold a larger definition.

### 3.4.7 Necessary Datatype Values

The subterm-closed substructure approach to inductive datatypes described in Section 3.3.2 typically scales up to cardinality 8 or so. However, in larger specifications such as the C++ memory model (Chapter 5), the combinatorial explosion goes off much sooner.

To address this issue, we came up with the following plan: Add an option to let users specify necessary datatype values. We encode that information in the FORL problem. Users provide a set of datatype values as ground constructor terms, and Nitpick assigns one FORL atom to each term and subterm from that set. The atom assignment is coded as an additional constraint in the FORL problem and passed to the SAT solver, which exploits it to prune the search space.

### 3.4.8 Lightweight Translation

There is a second efficiency issue related to Nitpick's handling of inductive datatypes. Attempting to construct a value that Nitpick cannot represent yields the unknown value $\star$. The ensuing partiality is handled by a three-valued Kleene logic.

Unfortunately, the three-valued logic puts a heavy burden on the translation, which must handle $\star$ gracefully and keep track of polarities (positive and negative contexts). An operation as innocuous as equality in HOL, which we could otherwise map to FORL equality, must be translated so that it returns $\star$ if either operand is $\star$. Formulas occurring in unpolarized, higher-order contexts (e.g., in a conditional expression or as an argument to a function) are less common but equally problematic.

The root of the problem is overflow: When a constructor returns $\star$, it overflows in much the same way that IEEE $n$-bit floating-point arithmetic operations can yield NaN ("not a number") for large operands. Nitpick's translation of Section 3.2.2 handles overflows robustly. If the tool were to know that overflows are impossible, it could disable this expensive machinery and use the more direct translation of Section 3.2.1.

With the subterm-closed substructure approximation of inductive datatypes, overflows are a fact of life. For example, the append operator on lists will overflow if the longest representable nonempty list is appended to itself. In contrast, selectors cannot overflow.

For some specifications, such as that of the C++ memory model (Chapter 5), we can convince ourselves that overflows are impossible. To exploit this precious property, we added an option, *total_consts*, to control whether the lightweight translation without $\star$ should be used. The lightweight translation must be used with care: A single overflow in a definition can prevent the discovery of models.

Naturally, it would be desirable to implement an analysis in Nitpick to determine precisely which constants can overflow and use this information in a hybrid translation. This is for future work.

## 3.5 Examples

We show Nitpick in action on five small yet realistic Isabelle formalizations that are beyond Refute's reach: a context-free grammar, AA trees, a security type system, a hotel key card system, and lazy lists. The examples were chosen to illustrate the main definitional principles. With some manual setup, the first three formalizations can also be checked by the latest version of Quickcheck.

### 3.5.1 A Context-Free Grammar

Our first example is adapted from the Isabelle/HOL tutorial [140, §7.4]. The following context-free grammar, originally due to Hopcroft and Ullman, produces all strings with an equal number of $a$'s and $b$'s:

$$S ::= \epsilon \mid bA \mid aB \qquad A ::= aS \mid bAA \qquad B ::= bS \mid aBB$$

The intuition behind the grammar is that $A$ generates all strings with one more $a$ than $b$'s and $B$ generates all strings with one more $b$ than $a$'s.

Context-free grammars can easily be expressed as inductive predicates on lists. The following HOL specification attempts to capture the above grammar, but a few errors were introduced to make the example more interesting.

**datatype** $\Sigma$ = a | b

**inductive_set** $\mathsf{S}^{\Sigma\ list \to o}$ **and** $\mathsf{A}^{\Sigma\ list \to o}$ **and** $\mathsf{B}^{\Sigma\ list \to o}$ **where**
$[] \in \mathsf{S}$
$w \in \mathsf{A} \Longrightarrow \mathsf{Cons\ b}\ w \in \mathsf{S}$
$w \in \mathsf{B} \Longrightarrow \mathsf{Cons\ a}\ w \in \mathsf{S}$
$w \in \mathsf{S} \Longrightarrow \mathsf{Cons\ a}\ w \in \mathsf{A}$
$w \in \mathsf{S} \Longrightarrow \mathsf{Cons\ b}\ w \in \mathsf{S}$
$v \in \mathsf{B} \Longrightarrow v \in \mathsf{B} \Longrightarrow \mathsf{Cons\ a}\ (v\,@\,w) \in \mathsf{B}$

Debugging faulty specifications is at the heart of Nitpick's *raison d'être*. A good approach is to state desirable properties of the specification—here, that S corresponds to the set of strings over Σ with as many a's as b's—and check them with the tool. If the properties are correctly stated, counterexamples will point to bugs in the specification. For our grammar example, we proceed in two steps, separating the soundness and the completeness of the set S:

$$\begin{array}{rl} \textsc{Sound}\text{:} & w \in \mathsf{S} \implies \mathsf{count\ a}\ w = \mathsf{count\ b}\ w \\ \textsc{Complete}\text{:} & \mathsf{count\ a}\ w = \mathsf{count\ b}\ w \implies w \in \mathsf{S} \end{array}$$

The auxiliary function count is defined below:

**primrec** $\mathsf{count}^{\alpha \to \alpha\ list \to nat}$ **where**
$\mathsf{count}\ x\ \mathsf{Nil} = 0$
$\mathsf{count}\ x\ (\mathsf{Cons}\ y\ ys) = (\text{if } x = y \text{ then } 1 \text{ else } 0) + \mathsf{count}\ x\ ys$

We first focus on soundness. The predicate S occurs negatively in SOUND, but positively in the negated conjecture ¬ SOUND. Wellfoundedness is easy to establish because the words in the conclusions are always at least one symbol longer than in the assumptions. As a result, Nitpick can use the fixed-point equations

$$\begin{aligned} x \in \mathsf{S} &= \big(x = \mathsf{Nil} \lor (\exists w.\ x = \mathsf{Cons\ b}\ w \land w \in \mathsf{A}) \\ &\qquad \lor (\exists w.\ x = \mathsf{Cons\ a}\ w \land w \in \mathsf{B}) \lor (\exists w.\ x = \mathsf{Cons\ b}\ w \land w \in \mathsf{S})\big) \\ x \in \mathsf{A} &= \big(\exists w.\ x = \mathsf{Cons\ a}\ w \land w \in \mathsf{S}\big) \\ x \in \mathsf{B} &= \big(\exists v\, w.\ x = \mathsf{Cons\ a}\ v\ @\ w \land v \in \mathsf{B} \land v \in \mathsf{B}\big) \end{aligned}$$

When invoked on SOUND with the default settings, Nitpick generates a sequence of 10 FORL problems corresponding to the scopes $|nat| = |\Sigma\ list| = k$ and $|\Sigma| = \min\{k, 2\}$ for $k \in \{1, \ldots, 10\}$. Datatypes approximated by subterm-closed substructures are always monotonic (Chapter 4), so it would be sufficient to try only the largest scope, but in practice it is usually more efficient to start with smaller scopes. The models obtained this way also tend to be simpler.

Nitpick almost instantly finds the counterexample $w = [b]$ for $k = 2$ built using the substructures

$$\{0,\ 1\} \subseteq nat \qquad \{[],\ [b]\} \subseteq \Sigma\ list \qquad \{a,\ b\} \subseteq \Sigma$$

and the constant interpretations

$$\begin{array}{rrrrr} [\,]\ @\ [\,] = [\,] & \mathsf{count\ a}\ [\,] = 0 & \mathsf{S}\ [\,] = \mathsf{True} & \mathsf{A}\ [\,] = \mathsf{False} & \mathsf{B}\ [\,] = \star \\ [\mathsf{b}]\ @\ [\,] = \star & \mathsf{count\ b}\ [\,] = 0 & \mathsf{S}\ [\mathsf{b}] = \mathsf{True} & \mathsf{A}\ [\mathsf{b}] = \mathsf{False} & \mathsf{B}\ [\mathsf{b}] = \star \\ [\,]\ @\ [\mathsf{b}] = [\mathsf{b}] & \mathsf{count\ a}\ [\mathsf{b}] = 0 & & & \\ [\mathsf{b}]\ @\ [\mathsf{b}] = \star & \mathsf{count\ b}\ [\mathsf{b}] = 1 & & & \end{array}$$

It would seem that $[\mathsf{b}] \in \mathsf{S}$. How could this be? An inspection of the introduction rules reveals that the only rule with a right-hand side of the form $\mathsf{Cons\ b}\ \ldots \in \mathsf{S}$ that could have added [b] to S is

$$w \in \mathsf{S} \implies \mathsf{Cons\ b}\ w \in \mathsf{S}$$

This rule is clearly wrong: To match the production $B ::= bS$, the second S would need to be a B.

If we fix the typo and run Nitpick again, we obtain the counterexample $w = [\mathsf{a}, \mathsf{a}, \mathsf{b}]$, which requires $k = 4$. This takes about 1.5 seconds on modern hardware. Some detective work is required to find out what went wrong. To get $[\mathsf{a},\mathsf{a},\mathsf{b}] \in \mathsf{S}$, we need $[\mathsf{a},\mathsf{b}] \in \mathsf{B}$, which in turn can only originate from

$$v \in \mathsf{B} \Longrightarrow v \in \mathsf{B} \Longrightarrow \mathsf{Cons\ a}\ (v \mathbin{@} w) \in \mathsf{B}$$

This introduction rule is highly suspicious: The same assumption occurs twice, and the variable $w$ is unconstrained. Indeed, one of the two occurrences of $v$ in the assumptions should have been a $w$.

With the correction made, we do not get any counterexample from Nitpick, which exhausts all scopes up to cardinality 10 well within the 30 second timeout. Let us move on and check completeness. Since the predicate S occurs negatively in the negated conjecture ¬ COMPLETE, Nitpick can safely use the fixed-point equations for S, A, and B as their specifications. This time we get the counterexample $w = [\mathsf{b},\mathsf{b},\mathsf{a},\mathsf{a}]$, with $k = 5$.

Apparently, $[\mathsf{b},\mathsf{b},\mathsf{a},\mathsf{a}]$ is not in S even though it has the same numbers of a's and b's. But since our inductive definition passed the soundness check, the introduction rules are likely to be correct. Perhaps we simply lack a rule. Comparing the grammar with the inductive definition, our suspicion is confirmed: There is no introduction rule corresponding to the production $A ::= bAA$, without which the grammar cannot generate two or more $b$'s in a row. So we add the rule

$$v \in \mathsf{A} \Longrightarrow w \in \mathsf{A} \Longrightarrow \mathsf{Cons\ b}\ (v \mathbin{@} w) \in \mathsf{A}$$

With this last change, we do not get any counterexamples for either soundness or completeness. We can even generalize our result to cover A and B as well:

$$\begin{aligned} w \in \mathsf{S} &\longleftrightarrow \mathsf{count\ a}\ w = \mathsf{count\ b}\ w \\ w \in \mathsf{A} &\longleftrightarrow \mathsf{count\ a}\ w = \mathsf{count\ b}\ w + 1 \\ w \in \mathsf{B} &\longleftrightarrow \mathsf{count\ a}\ w + 1 = \mathsf{count\ b}\ w \end{aligned}$$

Nitpick can test these formulas up to cardinality 10 within 30 seconds.

### 3.5.2 AA Trees

AA trees are a variety of balanced trees introduced by Arne Andersson that provide similar performance to red-black trees but are easier to implement [3]. They store sets of elements of type $\alpha$ equipped with a total order $<$. We start by defining the datatype and some basic extractor functions:

**datatype** $\alpha\ aa\_tree = \Lambda \mid \mathsf{N}\ \alpha\ nat\ (\alpha\ aa\_tree)\ (\alpha\ aa\_tree)$

**primrec** $\mathsf{level}^{\alpha\ aa\_tree \to nat}$ **where**
$\mathsf{level}\ \Lambda = 0$
$\mathsf{level}\ (\mathsf{N}\ \_\ k\ \_\ \_) = k$

**primrec** $\mathsf{data}^{\alpha\ aa\_tree \to \alpha}$ **where**
$\mathsf{data}\ (\mathsf{N}\ x\ \_\ \_\ \_) = x$

**primrec** $\mathsf{is\_in}^{\alpha \to \alpha\ aa\_tree \to o}$ **where**
$\mathsf{is\_in}\ \_\ \Lambda \longleftrightarrow \mathsf{False}$
$\mathsf{is\_in}\ a\ (\mathsf{N}\ x\ \_\ t\ u) \longleftrightarrow a = x \lor \mathsf{is\_in}\ a\ t \lor \mathsf{is\_in}\ a\ u$

**primrec** $\mathsf{left}^{\alpha\ aa\_tree \to \alpha\ aa\_tree}$ **where**

```
$\mathsf{left}\ \Lambda = \Lambda$
$\mathsf{left}\ (\mathsf{N}\ \_\ \_\ t\ \_) = t$
```

```
$\mathbf{primrec}\ \mathsf{right}^{\alpha\ aa\_tree \to \alpha\ aa\_tree}\ \mathbf{where}$
$\mathsf{right}\ \Lambda = \Lambda$
$\mathsf{right}\ (\mathsf{N}\ \_\ \_\ \_\ u) = u$
```

The wellformedness criterion for AA trees is fairly complex. Each node carries a *level* field, which must satisfy the following constraints:

- Nil trees $(\Lambda)$ have level 0.
- Leaf nodes (i.e., nodes of the form $N\ \_\ \_\ \Lambda\ \Lambda$) have level 1.
- A node's level must be at least as large as its right child's, and greater than its left child's and its grandchildren's.
- Every node of level greater than 1 must have two children.

The wf predicate formalizes this description:

```
$\mathbf{primrec}\ \mathsf{wf}^{\alpha\ aa\_tree \to o}\ \mathbf{where}$
$\mathsf{wf}\ \Lambda \longleftrightarrow \mathsf{True}$
$\mathsf{wf}\ (\mathsf{N}\ \_\ k\ t\ u) \longleftrightarrow$
    if $t = \Lambda$ then
      $k = 1 \wedge (u = \Lambda \vee (\mathsf{level}\ u = 1 \wedge \mathsf{left}\ u = \Lambda \wedge \mathsf{right}\ u = \Lambda))$
    else
      $\mathsf{wf}\ t \wedge \mathsf{wf}\ u \wedge u \neq \Lambda \wedge \mathsf{level}\ t < k \wedge \mathsf{level}\ u \leq k \wedge \mathsf{level}\ (\mathsf{right}\ u) < k$
```

Rebalancing the tree upon insertion and removal of elements is performed by two auxiliary functions, skew and split:

```
$\mathbf{primrec}\ \mathsf{skew}^{\alpha\ aa\_tree \to \alpha\ aa\_tree}\ \mathbf{where}$
$\mathsf{skew}\ \Lambda = \Lambda$
$\mathsf{skew}\ (\mathsf{N}\ x\ k\ t\ u) =$
    if $t \neq \Lambda \wedge k = \mathsf{level}\ t$ then
      $\mathsf{N}\ (\mathsf{data}\ t)\ k\ (\mathsf{left}\ t)\ (\mathsf{N}\ x\ k\ (\mathsf{right}\ t)\ u)$
    else
      $\mathsf{N}\ x\ k\ t\ u$
```

```
$\mathbf{primrec}\ \mathit{split}^{\alpha\ aa\_tree \to \alpha\ aa\_tree}\ \mathbf{where}$
$\mathsf{split}\ \Lambda = \Lambda$
$\mathsf{split}\ (\mathsf{N}\ x\ k\ t\ u) =$
    if $u \neq \Lambda \wedge k = \mathsf{level}\ (\mathsf{right}\ u)$ then
      $\mathsf{N}\ (\mathsf{data}\ u)\ (\mathsf{Suc}\ k)\ (\mathsf{N}\ x\ k\ t\ (\mathsf{left}\ u))\ (\mathsf{right}\ u)$
    else
      $\mathsf{N}\ x\ k\ t\ u$
```

Performing a skew or split operation should have no impact on the set of elements stored in the tree:

$$\mathsf{is\_in}\ a\ (\mathsf{skew}\ t) = \mathsf{is\_in}\ a\ t \qquad\qquad \mathsf{is\_in}\ a\ (\mathsf{split}\ t) = \mathsf{is\_in}\ a\ t$$

Furthermore, applying skew or split on a well-formed tree should not alter the tree:

$$\mathsf{wf}\ t \implies \mathsf{skew}\ t = t \qquad\qquad \mathsf{wf}\ t \implies \mathsf{split}\ t = t$$

All these properties can be checked up to cardinality 7 or 8 (i.e., for all subterm-closed sets of trees with at most 7 or 8 elements) within 30 seconds.

Insertion is implemented recursively. It preserves the sort order:

$$
\begin{array}{l}
\textbf{primrec}\ \mathsf{insort}^{\alpha\ aa\_tree \to \alpha \to \alpha\ aa\_tree}\ \textbf{where} \\
\mathsf{insort}\ \Lambda\ x = \mathsf{N}\ x\ 1\ \Lambda\ \Lambda \\
\mathsf{insort}\ (\mathsf{N}\ y\ k\ t\ u)\ x = \\
\quad \mathsf{split}\ (\mathsf{skew}\ (\mathsf{N}\ y\ k\ (\text{if } x < y \text{ then } \mathsf{insort}\ t\ x \text{ else } t) \\
\qquad\qquad\qquad\qquad (\text{if } x > y \text{ then } \mathsf{insort}\ u\ x \text{ else } u)))
\end{array}
$$

If we test the property

$$\mathsf{wf}\ t \implies \mathsf{wf}\ (\mathsf{insort}\ t\ x)$$

with the applications of skew and split commented out, we get the counterexample $t = \mathsf{N}\ \mathsf{a}_1\ 1\ \Lambda\ \Lambda$ and $x = \mathsf{a}_2$. It is hard to see why this is a counterexample without looking up the definition of $<$ on type $\alpha$.

However, to improve readability, we can restrict the theorem to *nat* and tell Nitpick to display the value of $\mathsf{insort}\ t\ x$. The counterexample is now $t = \mathsf{N}\ 1\ 1\ \Lambda\ \Lambda$ and $x = 0$, with $\mathsf{insort}\ t\ x = \mathsf{N}\ 1\ 1\ (\mathsf{N}\ 0\ 1\ \Lambda\ \Lambda)\ \Lambda$. The output reveals that the element 0 was added as a left child of 1, where both nodes have a level of 1. This violates the requirement that a left child's level must be less than its parent's. If we reintroduce the tree rebalancing code, Nitpick finds no counterexample up to cardinality 7.

### 3.5.3 The Volpano–Smith–Irvine Security Type System

Assuming a partition of program variables into public and private ones, Volpano, Smith, and Irvine [194] provide typing rules guaranteeing that the contents of private variables stay private. They define two types, High (private) and Low (public). An expression is High if it involves private variables; otherwise it is Low. A command is High if it modifies private variables only; commands that could alter public variables are Low.

As our third example, we consider a fragment of the formal soundness proof by Snelting and Wasserrab [173]. Given a variable partition $\Gamma$, the inductive predicate $\Gamma \vdash e : \sigma$ says whether the expression $e$ has type $\sigma$, whereas $\Gamma, \sigma \vdash c$ says whether the command $c$ has type $\sigma$. Below is a flawed definition of $\Gamma, \sigma \vdash c$:

$$
\frac{}{\Gamma, \sigma \vdash \texttt{skip}}
\qquad
\frac{\Gamma\ v = \lfloor \mathsf{High} \rfloor}{\Gamma, \sigma \vdash v \mathrel{\texttt{:=}} e}
\qquad
\frac{\Gamma \vdash e : \mathsf{Low} \quad \Gamma\ v = \lfloor \mathsf{Low} \rfloor}{\Gamma, \mathsf{Low} \vdash v \mathrel{\texttt{:=}} e}
\qquad
\frac{\Gamma, \sigma \vdash c_1}{\Gamma, \sigma \vdash c_1\ \texttt{;}\ c_2}
$$

$$
\frac{\Gamma \vdash b : \sigma \quad \Gamma, \sigma \vdash c_1 \quad \Gamma, \sigma \vdash c_2}{\Gamma, \sigma \vdash \texttt{if (}b\texttt{)}\ c_1\ \texttt{else}\ c_2}
\qquad
\frac{\Gamma \vdash b : \sigma \quad \Gamma, \sigma \vdash c}{\Gamma, \sigma \vdash \texttt{while (}b\texttt{)}\ c}
\qquad
\frac{\Gamma, \mathsf{High} \vdash c}{\Gamma, \mathsf{Low} \vdash c}
$$

The following theorem constitutes a key step in the soundness proof:

$$\Gamma, \mathsf{High} \vdash c \ \wedge\ \langle c, s \rangle \leadsto^{*} \langle \texttt{skip}, s' \rangle \implies \forall v.\ \Gamma\ v = \lfloor \mathsf{Low} \rfloor \longrightarrow s\ v = s'\ v$$

Informally, it asserts that if executing the High command $c$ in state $s$ terminates in state $s'$, the public variables of $s$ and $s'$ must agree. This is consistent with our intuition that High commands should modify only private variables. However, because we planted a bug in one of the rules for $\Gamma, \sigma \vdash c$, Nitpick finds a counterexample:

$$\begin{aligned} \Gamma &= [\mathsf{v}_1 \mapsto \mathsf{Low}] & s &= [\mathsf{v}_1 \mapsto \mathsf{false}] \\ c &= \mathtt{skip}\ ;\ \mathsf{v}_1 := (\mathsf{Var}\ \mathsf{v}_1 == \mathsf{Var}\ \mathsf{v}_1) & s' &= [\mathsf{v}_1 \mapsto \mathsf{true}] \end{aligned}$$

Even though the command $c$ has type High, it assigns the value true to the Low variable $\mathsf{v}_1$. The bug is a missing assumption $\Gamma, \sigma \vdash c_2$ in the typing rule for sequential composition.

### 3.5.4 A Hotel Key Card System

We consider a state-based model of a vulnerable hotel key card system with recordable locks [136], inspired by an Alloy specification due to Jackson [94, §E.1]. Where Alloy favors relational constraints, the Isabelle version challenges model finders with its reliance on inductive definitions, records, and datatypes.

The formalization revolves around three uninterpreted types, *room*, *guest*, and *key*. A key card, of type $card = key \times key$, combines an old key and a new key. A state is a 7-field record

$$\begin{aligned} &(\!|\mathsf{owns}^{room \to guest\ option},\ \mathsf{curr}^{room \to key},\ \mathsf{issued}^{key \to o},\ \mathsf{cards}^{guest \to card \to o}, \\ &\ \mathsf{roomk}^{room \to key},\ \mathsf{isin}^{room \to guest \to o},\ \mathsf{safe}^{room \to o}|\!) \end{aligned}$$

The set reach of reachable states is defined inductively by the following rules:

$$\frac{\mathsf{inj}\ init}{\begin{aligned} &(\!|\mathsf{owns} = (\lambda r.\ \emptyset),\ \mathsf{curr} = init,\ \mathsf{issued} = \mathsf{range}\ init,\ \mathsf{cards} = (\lambda g.\ \emptyset), \\ &\ \mathsf{roomk} = init,\ \mathsf{isin} = (\lambda r.\ \emptyset),\ \mathsf{safe} = (\lambda r.\ \mathsf{True})|\!) \in \mathsf{reach} \end{aligned}}\ \textsc{Init}$$

$$\frac{s \in \mathsf{reach} \qquad k \notin \mathsf{issued}\ s}{\begin{aligned} &s(\!|\mathsf{curr} := (\mathsf{curr}\ s)(r := k),\ \mathsf{issued} := \mathsf{issued}\ s \cup k, \\ &\ \ \mathsf{cards} := (\mathsf{cards}\ s)(g := \mathsf{cards}\ s\ g \cup \langle \mathsf{curr}\ s\ r,\ k\rangle), \\ &\ \ \mathsf{owns} := (\mathsf{owns}\ s)(r := \lfloor g \rfloor),\ \mathsf{safe} := (\mathsf{safe}\ s)(r := \mathsf{False})|\!) \in \mathsf{reach} \end{aligned}}\ \textsc{Check-In}$$

$$\frac{s \in \mathsf{reach} \qquad \langle k, k'\rangle \in \mathsf{cards}\ s\ g \qquad \mathsf{roomk}\ s\ r \in \{k, k'\}}{\begin{aligned} &s(\!|\mathsf{isin} := (\mathsf{isin}\ s)(r := \mathsf{isin}\ s\ r \cup g),\ \mathsf{roomk} := (\mathsf{roomk}\ s)(r := k'), \\ &\ \ \mathsf{safe} := (\mathsf{safe}\ s)(r := \mathsf{owns}\ s\ r = \lfloor g \rfloor \land \mathsf{isin}\ s\ r = \emptyset \lor \mathsf{safe}\ s\ r)|\!) \in \mathsf{reach} \end{aligned}}\ \textsc{Entry}$$

$$\frac{s \in \mathsf{reach} \qquad g \in \mathsf{isin}\ s\ r}{s(\!|\mathsf{isin} := (\mathsf{isin}\ s)(r := \mathsf{isin}\ s\ r - \{g\})|\!) \in \mathsf{reach}}\ \textsc{Exit}$$

A desirable property of the system is that it should prevent unauthorized access:

$$s \in \mathsf{reach} \implies \mathsf{safe}\ s\ r \implies g \in \mathsf{isin}\ s\ r \implies \mathsf{owns}\ s\ r = \lfloor g \rfloor$$

Nitpick needs some help to contain the state space explosion: We restrict the search to one room and two guests. Within seconds, we get the counterexample

$$\begin{aligned} s = {} &(\!|\mathsf{owns} = \mathsf{undefined}(\mathsf{r}_1 := \lfloor \mathsf{g}_1 \rfloor),\ \mathsf{curr} = \mathsf{undefined}(\mathsf{r}_1 := \mathsf{k}_1), \\ &\ \mathsf{issued} = \{\mathsf{k}_1, \mathsf{k}_2, \mathsf{k}_3, \mathsf{k}_4\}, \\ &\ \mathsf{cards} = \mathsf{undefined}(\mathsf{g}_1 := \{\langle \mathsf{k}_3, \mathsf{k}_1\rangle, \langle \mathsf{k}_4, \mathsf{k}_2\rangle\},\ \mathsf{g}_2 := \{\langle \mathsf{k}_2, \mathsf{k}_3\rangle\}), \\ &\ \mathsf{roomk} = \mathsf{undefined}(\mathsf{r}_1 := \mathsf{k}_3),\ \mathsf{isin} = \mathsf{undefined}(\mathsf{r}_1 := \{\mathsf{g}_1, \mathsf{g}_2\}),\ \mathsf{safe} = \{\mathsf{r}_1\}|\!) \end{aligned}$$

with $g = \mathsf{g}_2$ and $r = \mathsf{r}_1$.

To retrace the steps from the initial state to the state $s$, we can ask Nitpick to show the interpretation of reach at each iteration. This reveals the following "guest in the middle" attack:

1. Guest $\mathsf{g}_1$ checks in and gets a card $\langle \mathsf{k}_4,\ \mathsf{k}_2 \rangle$ for room $\mathsf{r}_1$, whose lock expects $\mathsf{k}_4$. Guest $\mathsf{g}_1$ does not enter the room yet.
2. Guest $\mathsf{g}_2$ checks in, gets a card $\langle \mathsf{k}_2,\ \mathsf{k}_3 \rangle$ for $\mathsf{r}_1$, and waits.
3. Guest $\mathsf{g}_1$ checks in again, gets a card $\langle \mathsf{k}_3,\ \mathsf{k}_1 \rangle$, inadvertently unlocks $\mathsf{r}_1$ with her previous card, $\langle \mathsf{k}_4,\ \mathsf{k}_2 \rangle$, leaves a diamond on the nightstand, and exits.
4. Guest $\mathsf{g}_2$ enters the room and "borrows" the diamond.

This flaw was already detected by Jackson using the Alloy Analyzer on his original specification and can be fixed by adding $k' = \mathsf{curr}\ s\ r$ to the conjunction in ENTRY.

### 3.5.5 Lazy Lists

The codatatype $\alpha\ llist$ of lazy lists [114,148] is generated by the constructors $\mathsf{LNil}^{\alpha\ llist}$ and $\mathsf{LCons}^{\alpha \to \alpha\ llist \to \alpha\ llist}$. It is of particular interest to countermodel finding because many properties of finite lists do not carry over to infinite lists, often in baffling ways. To illustrate this, we conjecture that appending $ys$ to $xs$ yields $xs$ iff $ys$ is LNil:

$$xs \mathbin{@_{\mathsf{L}}} ys = xs \longleftrightarrow ys = \mathsf{LNil}$$

The operator $@_{\mathsf{L}}$ is defined corecursively in Section 2.3. Nitpick immediately finds the countermodel $xs = ys = [0, 0, \ldots]$, in which a cardinality of 1 is sufficient for $\alpha$ and $\alpha\ llist$, and the bisimilarity predicate $\sim$ is unrolled only once. Indeed, appending $[0, 0, \ldots] \neq []$ to $[0, 0, \ldots]$ leaves $[0, 0, \ldots]$ unchanged. Many other counterexamples are possible—for example, $xs = [0, 0, \ldots]$ and $ys = [1]$—but Nitpick tends to reuse the objects that are part of a subterm-closed substructure to keep cardinalities low. Although very simple, the counterexample is beyond Quickcheck's and Refute's reach, since they do not support codatatypes.

The next example requires the following lexicographic order predicate:

$$\begin{array}{l}
\textbf{coinductive} \preceq^{nat\ llist \to nat\ llist \to o}\ \textbf{where} \\
\mathsf{LNil} \preceq xs \\
x \le y \Longrightarrow \mathsf{LCons}\ x\ xs \preceq \mathsf{LCons}\ y\ ys \\
xs \preceq ys \Longrightarrow \mathsf{LCons}\ x\ xs \preceq \mathsf{LCons}\ x\ ys
\end{array}$$

The intention is to define a linear order on lazy lists of natural numbers, and hence the following properties should hold:

$$\begin{array}{rlrl}
\text{REFL:} & xs \preceq xs & \text{ANTISYM:} & xs \preceq ys \land ys \preceq xs \longrightarrow xs = ys \\
\text{LINEAR:} & xs \preceq ys \lor ys \preceq xs & \text{TRANS:} & xs \preceq ys \land ys \preceq zs \longrightarrow xs \preceq zs
\end{array}$$

However, Nitpick finds a counterexample for ANTISYM: $xs = [1, 1]$ and $ys = [1]$. On closer inspection, the assumption $x \le y$ of the second introduction rule for $\preceq$ should have been $x < y$; otherwise, any two lists $xs$, $ys$ with the same head satisfy $xs \preceq ys$. Once we repair the specification, no more counterexamples are found for the four properties up to cardinality 6 for $nat$ and $nat\ llist$ within the time limit of 30 seconds. Andreas Lochbihler used Isabelle to prove all four properties [114].

We could continue like this and sketch a full-blown theory of lazy lists. Once the definitions and main theorems have been thoroughly tested using Nitpick, we could start working on the proofs. Developing theories this way can save a lot of time, because faulty theorems and definitions are discovered early.

## 3.6 Evaluation

An ideal way to assess Nitpick's strength would be to run it against Quickcheck [21, 49] and Refute [198] on a representative database of Isabelle/HOL non-theorems. Lacking such a database, we instead derived formulas from existing theorems by mutation, replacing constants with other constants and swapping arguments, as was done to evaluate an early version of Quickcheck [21, §7]. We filtered out the mutated formulas that could quickly be proved using Sledgehammer or automatic proof methods. The vast majority of the remaining mutants are invalid, and those that are valid do not influence the ranking of the tools. For executable theorems, we made sure that the mutants are executable to prevent a bias against Quickcheck.

In addition to Nitpick and Refute, we ran three instances of Quickcheck with different strategies—random, exhaustive, and narrowing [30, 49]. We set the time limit to 20 seconds and restricted each tool to a single thread. Most counterexamples are found within a few seconds; giving the tools more time would hardly have any impact on the results. For Quickcheck, we followed its maintainer's advice and increased the maximum size parameter from 5 to 20, the number of iterations per size from 100 to 1000, and the maximum depth from 10 to 20, to exploit the available time better. For Refute and Nitpick, we kept the default settings.

We selected ten theories from the official Isabelle distribution and as many from the *Archive of Formal Proofs* (*AFP*) [102]. Both sets of theories are listed below, with abbreviated theory names. The last column gives the features contained by each theory, where A means arithmetic, I means induction/recursion, L means $\lambda$-abstractions, and S means sets.[1]

[1] We owe this categorization scheme to Böhme and Nipkow [44, §2].

| THY. | DESCRIPTION | FEATS. |
|---|---|---|
| *Div* | The division operators ‘div’ and ‘mod’ | A I |
| *Fun* | Functions | L S |
| *GCD* | Greatest common divisor and least common multiple | A I |
| *Lst* | Inductive lists | A I L S |
| *Map* | Maps (partial functions) | I L S |
| *Pr* | Predicates | I L S |
| *Rel* | Relations | I L S |
| *Ser* | Finite summations and infinite series | A L |
| *Set* | Simply typed sets | L S |
| *Wf* | Well-founded recursion | A I L S |
| *BQ* | Functional binomial queues | I S |
| *C++* | Multiple inheritance in C++ | I S |
| *Co* | Coinductive datatypes | I L |
| *FOL* | First-order logic syntax and semantics | I L S |
| *HA* | Hierarchical automata | I S |
| *Huf* | Optimality of Huffman’s algorithm | A I L S |
| *ML* | Type inference for MiniML | A I L S |
| *Mtx* | Executable matrices | I L S |
| *NBE* | Normalization by evaluation | A I L S |
| *TA* | Tree automata | I L S |

These theories cover a wide range of idioms: The theories from the distribution are somewhat simpler, whereas those from the *AFP* are perhaps more representative of real applications. We generated 200 mutants per theory.

We are only interested in counterexamples that are labeled as genuine. In particular, for Refute this rules out all problems involving recursive datatypes: Its three-valued logic is unsound, so all countermodels for conjectures that involve an infinite type are potentially spurious and reported as such to the user.[1] We disqualified Refute for *ML* and *NBE* because it ignores the axioms in these theories and hence finds spurious “genuine” countermodels, whereas Nitpick correctly takes these into consideration.

Figures 3.1 and 3.2 give the success rate for each tool on each theory from the Isabelle distribution and the *AFP*, respectively, together with the unique contributions of each tool (i.e., the percentage of mutants that only that tool can disprove).

It should come as no surprise that Nitpick practically subsumes its precursor Refute. On theories that admit small finite models, such as *Fun*, *Rel*, and *Set*, there is little that distinguishes the two tools: Not only do they falsify about as many conjectures, but they largely falsify the same ones. The TPTP problems used in the CADE ATP System Competition (CASC) exhibit similar features, which explains why the two tools are neck and neck there [181].

That Nitpick would gain the upper hand over Quickcheck was harder to foresee. As a rule of thumb, they are comparable on conjectures that belong to the exe-

[1] Refute is actually sound for monotonic formulas (Chapter 4). It would not be difficult to extend the tool with a syntactic check that infers this, transforming many of the “potentially spurious” counterexamples into “genuine” ones.

| | *Ser* | *Lst* | *GCD* | *Pre* | *Div* | *Wf* | *Fun* | *Map* | *Rel* | *Set* | All | Uniq. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Qc. Random | ***8*** | 42 | 63 | 34 | 53 | 42 | 80 | 85 | ***87*** | 84 | 57.5 | .0 |
| Qc. Exhaust. | ***8*** | 42 | 61 | 33 | 53 | 43 | 79 | ***86*** | ***87*** | 84 | 57.4 | .1 |
| Qc. Narrow. | 4 | 36 | 60 | 27 | 46 | 30 | 44 | 67 | 63 | 53 | 42.7 | .2 |
| Refute | 1 | 2 | 0 | 63 | 0 | 59 | 81 | 50 | 84 | ***87*** | 42.6 | .0 |
| Nitpick | 6 | ***67*** | ***68*** | ***77*** | ***71*** | ***79*** | ***84*** | ***86*** | ***87*** | ***87*** | ***71.0*** | ***8.1*** |
| All Tools | 10 | 70 | 71 | 77 | 77 | 79 | 84 | 87 | 87 | 87 | 72.6 | – |

**Figure 3.1:** Success rates (%) on all mutants per tool and distribution theory

| | *C++* | *HA* | *NBE* | *ML* | *FOL* | *Mtx* | *Co* | *TA* | *Huf* | *BQ* | All | Uniq. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Qc. Random | 0 | 1 | 17 | 11 | 42 | 35 | 1 | 5 | ***68*** | 32 | 20.9 | .6 |
| Qc. Exhaust. | 0 | 1 | 19 | 11 | 37 | ***41*** | 1 | 5 | 58 | 30 | 20.2 | .7 |
| Qc. Narrow. | 0 | 0 | 1 | 10 | 4 | 17 | 3 | 4 | 36 | 25 | 9.8 | .4 |
| Refute | 1 | 5 | 0 | 0 | 7 | 2 | 0 | 1 | 0 | 5 | 1.9 | .1 |
| Nitpick | ***8*** | ***9*** | ***28*** | ***39*** | ***47*** | 28 | ***58*** | ***61*** | ***68*** | ***71*** | ***41.6*** | ***22.1*** |
| All Tools | 8 | 9 | 32 | 39 | 53 | 54 | 60 | 61 | 69 | 72 | 45.6 | – |

**Figure 3.2:** Success rates (%) on all mutants per tool and *AFP* theory

cutable fragment supported by Isabelle's code generator, but Nitpick can additionally handle some non-executable formulas. Executability is a harsh taskmaster: Quickcheck is fundamentally unable to disprove $\mathsf{hd}\ [] = x$, since taking the head of the empty list is unspecified and results in a run-time exception, and it meets similar hurdles with $\mathsf{undefined}$, $\iota$, and $\varepsilon$. In addition, because it lacks a skolemizer, it cowardly refuses to tackle conjectures of the form $\forall x^{\sigma}.\ P\ x$ where $\sigma$ is infinite.

On the other hand, Quickcheck is often one or two orders of magnitudes faster than Nitpick when it succeeds, and it performs well on theories that were designed with code generation in mind (e.g., *Huf* and *Mtx*). It also handles rational and real arithmetic more smoothly than Nitpick (e.g., *Ser*).[1] The unique contribution numbers for the Quickcheck variants are deceptively low because they compete against each other; overall, 4.0% of the *AFP* mutants are only disproved by Quickcheck.

Nitpick and Quickcheck nicely complement each other: Because it is so fast, Quickcheck is enabled by default to run on all conjectures. Users are so accustomed to its feedback that they rarely realize to what extent they benefit from it. Every so often, Nitpick finds a counterexample beyond Quickcheck's reach.

## 3.7 Related Work

We classify the approaches for testing conjectures in four broad categories: reduction to SAT, direct exhaustive search, random testing, and other approaches.

[1] Nitpick considers the type of rationals as an incomplete datatype generated by a non-free constructor, relying on a generalization of the SUA axioms described in Section 3.3.2. Reals are handled in exactly the same way, which is very incomplete; for example, the tool cannot falsify $\pi = 0$.

**Reduction to SAT.** SAT-based model finders translate the problem to propositional logic and pass it to a SAT solver. The translation is parameterized by upper or exact finite bounds on the cardinalities of the atomic types. Each predicate or relation is encoded by propositional variables, with each variable indicating whether a tuple is part of the relation or not. Functions can be treated as relations with an additional constraint. This procedure was pioneered by McCune in the earlier versions of Mace (also capitalized as MACE) [117]. Other SAT-based finders are Paradox [62], Kodkod [187], and Refute [198].

The most successful SAT-based finders implement many optimizations. Paradox and Kodkod generate symmetry breaking constraints to reduce the number of isomorphic models to consider. Both can invoke SAT solvers incrementally, reusing constraints across problems. Paradox is tailored to the untyped first-order logic with equality featured at CASC: It infers types in untyped problems and analyzes the unsatisfiability core returned by the SAT solver to determine which type's cardinality to increment next.

Nitpick's encoding of inductive datatypes in FORL has been introduced by Kuncak and Jackson [107] and further studied by Dunets et al. [70]. Kuncak and Jackson focused on lists and trees. Dunets et al. showed how to handle primitive recursion; their approach to recursion is similar to ours, but the use of a two-valued logic compelled them to generate additional definedness guards. Nitpick's unrolling of inductive predicates was inspired by bounded model checking [24] and by the Alloy idiom for state transition systems [94, pp. 172–175].

Another inspiration has been Weber's higher-order model finder Refute [198]. It uses a three-valued logic, but sacrifices soundness for precision. Datatypes are approximated by subterm-closed substructures that contain *all* datatype values built using up to $k$ nested constructors [198, §3.6.2]. This scheme proved disadvantageous in practice, because it generally requires much higher cardinalities to obtain the same models as with Kuncak and Jackson's approach. Weber handled (co)inductive predicates by expanding their $\mathsf{lfp}/\mathsf{gfp}$ definition, which does not scale beyond a cardinality of 3 or 4 for the predicate's domain.

Satisfiability modulo theories (SMT) solvers, whose core search engine is a SAT solver, can find models of first-order logic with equality and built-in theories (e.g., for arithmetic). However, in the presence of quantifiers, the models may be spurious. Recent research shows how to exploit SMT solvers to find sound models for well-founded recursive functions [185]. The approach can be seen as a dynamic version of Nitpick's translation of recursive functions (Section 3.3.2). The SMT solvers' handling of datatypes incurs less spinning than Kuncak and Jackson's approach while scaling better than Refute's. Theory reasoning and even some symbolic reasoning are supported.

**Direct Exhaustive Search.** An alternative to translating the problem to propositional logic is to perform an exhaustive model search directly on the original (first-order or higher-order) problem. Given fixed cardinalities, the search space is represented in memory as multidimensional arrays. The procedure tries different values in the function and predicate table entries, checking each time if the problem is satisfied and backtracking if necessary. This approach was pioneered by SEM [207] and FINDER [172] and serves as the basis of many more model finders, notably the Alloy Analyzer's precursor (called Nitpick, like our tool) [93] and the later versions of Mace [118].

These finders perform symmetry breaking directly during the proof search, instead of through additional constraints. They can also propagate equality constraints efficiently. As a result, they tend to be more efficient than SAT-based finders on equational problems [62, p. 1].

**Random Testing.** Random testing evaluates the conjecture for randomly generated values of its free variables. It is embodied by the QuickCheck or Quickcheck tools for Haskell [59], Agda [72], and Isabelle/HOL [21,49] as well as similar tools for ACL2 [54] and PVS [144]. As the names of many of these tools suggest, random testing's main strength is its speed.

Random testing tools are generally restricted to formulas that fall within an executable fragment of the source logic and can only exhibit counterexamples in which the conjecture is falsified irrespective of the interpretation of underspecified constants. Given a conjecture $\phi[\bar{y}]$, random testing exhibits ground witnesses $\bar{w}$ for the free variables $\bar{y}$ such that $\phi[\bar{w}]$ is equivalent to $\mathsf{False}$.

Some random testing tools expect the user to provide random value generators for custom datatypes. In contrast, Isabelle's Quickcheck and ACL2's random testing framework automatically synthesize generators following the datatype definition. They also perform a static data-flow analysis to compute dependencies between free variables, taking premises into account; this helps avoid the vacuous test cases that typically plague random testing.

**Other Approaches.** While all of the above tools are restricted to ground reasoning and finite models, some model finders implement a more symbolic approach based on narrowing or logic programming [30,81,112,169]. There is also a line of research on infinite model generation [51], so far with little practical impact.

Some proof methods deliver sound or unsound counterexamples upon failure; for example, linear arithmetic decision procedures can provide concrete values that falsify the conjecture [53], model checking can in principle provide a counterexample in the form of a lasso-shaped trace if there exists a counterexample [73], resolution provers can sometimes output a saturation—an exhaustive list of all (normalized) clauses that can be derived from the given axioms [7], and semantic tableaux have been adapted to produce finite counterexamples [103,162].

While the focus of our work is on adding Alloy-style model finding capabilities to an interactive theorem prover based on higher-order logic, the other direction—adding proof support to Alloy—is also meaningful: The Dynamite tool [79] lets users prove Alloy formulas in the interactive theorem prover PVS.

> Monotony is the ruling characteristic,—
> monotony of beauty, monotony of desolation,
> monotony even of variety.
>
> — Julian Hawthorne (1892)

# Chapter 4

# Monotonicity Inference

In model finders that work by enumerating scopes, the choice and order of the scopes is critical for performance, especially for formulas involving many atomic types (nonschematic type variables and other uninterpreted types). We present a solution that prunes the search space by inferring monotonicity with respect to atomic types. Monotonicity is undecidable, so we approximate it syntactically.

Our measurements show that monotonic formulas are pervasive in HOL formalizations and that syntactic criteria can usually detect them. Our criteria have been implemented as part of Nitpick, with dramatic speed gains.

This chapter describes joint work with Alexander Krauss [32, 33].

## 4.1 Monotonicity

Formulas occurring in logical specifications often exhibit monotonicity in the sense that if the formula is satisfiable when the types are interpreted with sets of given (positive) cardinalities, it is still satisfiable when these sets become larger.

**Definition 4.1 (Monotonicity).** Let $S \leq_\alpha S'$ abbreviate the conditions $S(\alpha) \subseteq S'(\alpha)$ and $S(\beta) = S'(\beta)$ for all $\beta \neq \alpha$. A formula $t$ is *monotonic with respect to* an atomic type $\alpha$ (or $\alpha$ is *monotonic in* $t$) if for all scopes $S, S'$ such that $S \leq_\alpha S'$, if $t$ is satisfiable for $S$, it is also satisfiable for $S'$. It is *antimonotonic with respect to* $\alpha$ if its negation is monotonic with respect to $\alpha$.

Given a scope $S$, the set $S(\alpha)$ can be finite or infinite, although for model finding we usually have finite domains in mind. In contexts where $S$ is clear, the cardinality of $S(\alpha)$ is written $|\alpha|$, and the elements of $S(\alpha)$ are denoted by 0, 1, 2, etc.

**Example 4.2.** Consider the following formulas:

1. $\exists x^\alpha\, y.\; x \neq y$
2. $f\; x^\alpha = x \wedge f\; y \neq y$
3. $(\forall x^\alpha.\; f\; x = x) \wedge f\; y \neq y$
4. $\{y^\alpha\} = \{z\}$
5. $\exists x^\alpha\, y.\; x \neq y \wedge \forall z.\; z = x \vee z = y$
6. $\forall x^\alpha\, y.\; x = y$

It is easy to see that formulas 1 and 2 are satisfiable iff $|\alpha| > 1$, formula 3 is unsatisfiable, formula 4 is satisfiable for any cardinality of $\alpha$, formula 5 is satisfiable iff

$|\alpha| = 2$, and formula 6 is satisfiable iff $|\alpha| = 1$. Formulas 1 to 4 are monotonic with respect to $\alpha$, whereas 5 and 6 are not.

**Example 4.3.** Imagine a village of monkeys [61] where each monkey owns at least two bananas:

$$
\begin{aligned}
&\forall m.\ \mathsf{owns}\ m\ (\mathsf{banana1}\ m) \wedge \mathsf{owns}\ m\ (\mathsf{banana2}\ m)\\
&\forall m.\ \mathsf{banana1}\ m \neq \mathsf{banana2}\ m\\
&\forall b\ m_1\ m_2.\ \mathsf{owns}\ m_1\ b \wedge \mathsf{owns}\ m_2\ b \longrightarrow m_1 = m_2
\end{aligned}
$$

The predicate $\mathsf{owns}^{monkey \to banana \to o}$ associates *monkey*s with their *banana*s, and the functions $\mathsf{banana1}, \mathsf{banana2}^{monkey \to banana}$ witness the existence of each monkey's minimum supply of bananas. The type *banana* is monotonic, because any model with $k$ bananas can be extended to a model with $k' > k$ bananas (where $k$ and $k'$ can be infinite cardinals). In contrast, *monkey* is nonmonotonic, because there can live at most $n$ monkeys in a village with a finite supply of $2n$ bananas.

*Convention.* In the sequel, we denote by $\tilde{\alpha}$ the atomic type with respect to which we consider monotonicity when not otherwise specified.

In plain first-order logic without equality, every formula is monotonic, since it is impossible to express an upper bound on the cardinality of the models and hence any model can be extended to a model of arbitrarily larger cardinality. This monotonicity property is essentially a weak form of the upward Löwenheim–Skolem theorem. When equality is added, nonmonotonicity follows suit.

Our interest in monotonicity arose in the context of model finding, where it can help prune the search space. Nitpick and its predecessor Refute systematically enumerate the domain cardinalities for the atomic types occurring in the formula. To exhaust all models up to a given cardinality bound $k$ for a formula involving $n$ atomic types, a model finder must iterate through $k^n$ combinations of cardinalities and must consider all models for each of these combinations. In general, this exponential behavior is necessary for completeness, since the formula may dictate a model with specific cardinalities. However, if the formula is monotonic with respect to all its atomic types, it is sufficient to consider only the models in which all types have cardinality $k$.

Another related use of monotonicity is to find finite fragments of infinite models. A formal specification of a programming language might represent variables by strings, natural numbers, or values of some other infinite type. Typically, the exact nature of these types is irrelevant; they are merely seen as inexhaustible name stores and used monotonically. If the specification is weakened to allow finite models, we can apply model finders with the guarantee that any finite model found is a substructure of an infinite model.

Monotonicity occurs surprisingly often in practice. Consider the specification of a hotel key card system with recordable locks (Section 3.5.4). Such a specification involves rooms, guests, and keys, modeled as distinct uninterpreted types. A desirable property of the system is that only the occupant of a room may unlock it. A counterexample requiring one room, two guests, and four keys will still be a counterexample if more rooms, guests, or keys are available. Indeed, it should remain a counterexample if infinitely many keys are available, as would be the case if keys are modeled by integers or strings.

**Theorem 4.4 (Undecidability).** *Monotonicity is undecidable.*

*Proof* (reduction). For any closed HOL formula $t$, let $t^\star \equiv t \lor \forall x^{\tilde{\alpha}}\, y.\; x = y$, where $\tilde{\alpha}$ does not occur in $t$. Clearly, $t^\star$ must be monotonic if $t$ is valid, since the second disjunct becomes irrelevant in this case. If $t$ is not valid, then $t^\star$ cannot be monotonic, since it is true for $|\tilde{\alpha}| = 1$ due to the second disjunct but false for some larger scopes. Thus, validity in HOL (which is undecidable) can be reduced to monotonicity. □

Since monotonicity is a semantic property, it is not surprising that it is undecidable; the best we can do is approximate it. In the rest of this chapter, we present three calculi for detecting monotonicity of HOL formulas. The first calculus (Section 4.2) simply tracks the use of equality and quantifiers. Although useful in its own right, it mainly serves as a stepping stone for a second, refined calculus (Section 4.3), which employs a type system to detect the ubiquitous "sets as predicates" idiom and treats it specially. The third calculus (Section 4.4) develops this idea further.

Since HOL is typed, we are interested in monotonicity with respect to a given (nonschematic) type variable or some other uninterpreted type $\tilde{\alpha}$. Moreover, our calculi must cope with occurrences of $\tilde{\alpha}$ in nested function types such as $(\tilde{\alpha} \to \beta) \to \beta$ and in datatypes such as $\tilde{\alpha}$ *list*. We are not aware of any previous work on inferring or proving monotonicity for HOL. While some of the difficulties we face are specific to HOL, the calculi can be adapted to any logic that provides unbounded quantification, such as simply typed first-order logic with equality.

The calculi are constructive: Whenever they infer monotonicity, they also yield a recipe for extending models into larger, possibly infinite models. They are also readily extended to handle constant definitions (Section 4.5.1) and inductive datatypes (Section 4.5.2), which pervade HOL formalizations. Our evaluation (Section 4.5.3) is done in the context of Nitpick. On a corpus of 1183 monotonic formulas from six theories, the strongest calculus infers monotonicity for 85% of them.

## 4.2 First Calculus: Tracking Equality and Quantifiers

This section presents the simple calculus $\mathfrak{M}_1$ for inferring monotonicity, which serves as a stepping stone toward the more general calculi $\mathfrak{M}_2$ and $\mathfrak{M}_3$ of Sections 4.3 and 4.4. Since the results are fairly intuitive and subsumed by those of the next sections, we omit the proofs.

All three calculi assume that $o$ (Boolean) and $\to$ (function) are the only interpreted type constructors. Other types are considered atomic until Section 4.5.2, which extends the calculi to handle inductive datatypes specially.

### 4.2.1 Extension Relation and Constancy

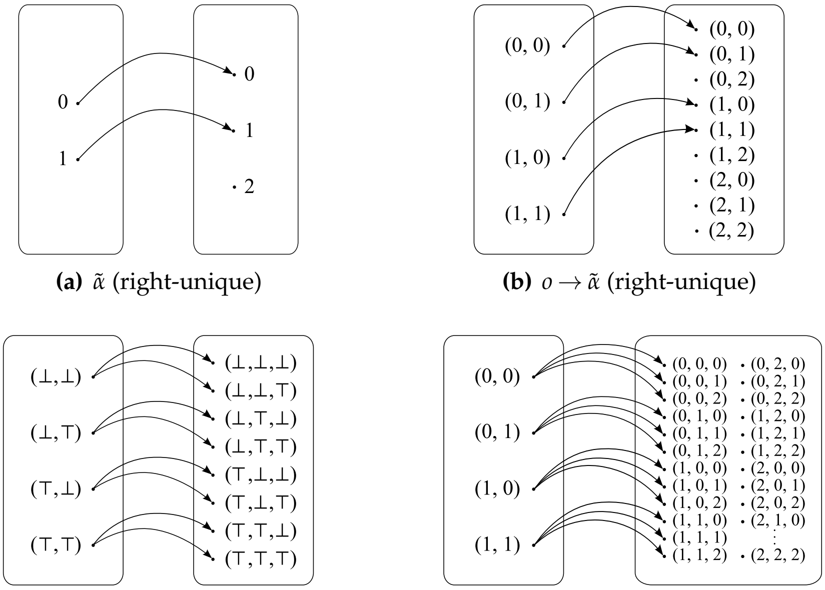
We first introduce a concept that is similar to monotonicity but that applies not only to formulas but also to terms of any type—the notion of *constancy*. Informally, a term is constant if it denotes essentially the same value before and after we enlarge the scope. What it means to denote "essentially the same value" can be formalized using an extension relation $\sqsubseteq$, which relates elements of the smaller scope to elements of the larger scope.

For types such as $o$, $\alpha$, and $\tilde{\alpha}$, this is easy: Any element of the smaller scope is also present in the larger scope and can serve as an extension. For functions, the extended function must coincide with the original one where applicable; elements not present in the smaller scope may be mapped to any value. For example, when going from $|\tilde{\alpha}| = 1$ to $|\tilde{\alpha}| = 2$, the function $f^{\tilde{\alpha}\to o} = [0 \mapsto \top]$ can be extended to $g = [0 \mapsto \top,\, 1 \mapsto \bot]$ or $g' = [0 \mapsto \top,\, 1 \mapsto \top]$. In other words, we take the liberal view that both $g$ and $g'$ are "essentially the same value" as $f$, and we write $f \sqsubseteq^{\tilde{\alpha}\to o} g$ and $f \sqsubseteq^{\tilde{\alpha}\to o} g'$. We reconsider this decision in Section 4.3.

**Definition 4.5 (Extension).** Let $\sigma$ be a type, and let $S$, $S'$ be scopes such that $S \leq_{\tilde{\alpha}} S'$. The *extension* relation $\sqsubseteq^{\sigma} \subseteq [\![\sigma]\!]_S \times [\![\sigma]\!]_{S'}$ for $S$ and $S'$ is defined by the following pair of equivalences:

$$\begin{array}{rll} a \sqsubseteq^{\sigma} b & \text{iff} \quad a = b & \text{if } \sigma \text{ is } o \text{ or an atomic type} \\ f \sqsubseteq^{\sigma\to\tau} g & \text{iff} \quad \forall a\, b.\ a \sqsubseteq^{\sigma} b \longrightarrow f(a) \sqsubseteq^{\tau} g(b) & \end{array}$$

The expression $a \sqsubseteq^{\sigma} b$ is read "$a$ is extended by $b$" or "$b$ extends $a$." The element $a$ is $b$'s *restriction* to $S$, and $b$ is $a$'s *extension* to $S'$. In addition, we will refer to elements $b \in [\![\sigma]\!]_{S'}$ as being *old* if they admit a restriction to $S$ and *new* if they do not admit any restriction.



**(a)** $\tilde{\alpha}$ (right-unique)

**(b)** $o \to \tilde{\alpha}$ (right-unique)

**(c)** $\tilde{\alpha} \to o$ (right-total)

**(d)** $\tilde{\alpha} \to \tilde{\alpha}$ (neither)

**Figure 4.1:** $\sqsubseteq^{\sigma}$ for various types $\sigma$, with $|S(\tilde{\alpha})| = 2$ and $|S'(\tilde{\alpha})| = 3$

Figure 4.1 illustrates $\sqsubseteq^{\sigma}$ for various types. We represent a function from $\sigma$ to $\tau$ by a $|\sigma|$-tuple such that the $n$th element for $\sigma$ (according to the lexicographic order, with $\bot < \top$ and $n < n + 1$) is mapped to the $n$th tuple component. Observe that

$\sqsubseteq^{\sigma}$ is always left-total[1] and left-unique[2]. It is also right-total[3] if $\tilde{\alpha}$ does not occur positively in $\sigma$ (e.g., $\sigma = \tilde{\alpha} \to o$), and right-unique[4] if $\tilde{\alpha}$ does not occur negatively (e.g., $\sigma = o \to \tilde{\alpha}$). These properties are crucial to the correctness of our calculus, which restricts where $\tilde{\alpha}$ may occur. They are proved in Section 4.3.

*Convention.* To simplify the calculi and proofs, constants express the logical primitives, whose interpretation is fixed a priori. Our definition of HOL is fairly minimalistic, with only equality ($=^{\sigma\to\sigma\to o}$ for any $\sigma$) and implication ($\longrightarrow^{o\to o\to o}$) as primitive constants. Other constants are treated as variables whose definition is conjoined with the formula. We always use the standard constant model $\widehat{M}_S$, which interprets $\longrightarrow$ and $=$ in the usual way, allowing us to omit the third component of $\mathcal{M} = (S, V, \widehat{M})$. Since we are interested in finding (counter)models of conjectures, type variables are always nonschematic.

**Definition 4.6 (Model Extension).** Let $\mathcal{M} = (S, V)$ and $\mathcal{M}' = (S', V')$ be models. The model $\mathcal{M}'$ *extends* $\mathcal{M}$, written $\mathcal{M} \sqsubseteq \mathcal{M}'$, if $S \leq_{\tilde{\alpha}} S'$ and $V(x) \sqsubseteq^{\sigma} V'(x)$ for all $x^{\sigma}$.

The relation $\sqsubseteq$ on models provides a recipe to transform a smaller model $\mathcal{M}$ into a larger model $\mathcal{M}'$. Because $\sqsubseteq^{\sigma}$ is left-total, the recipe never fails.

**Definition 4.7 (Constancy).** A term $t^{\sigma}$ is *constant* if $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![t]\!]_{\mathcal{M}'}$ for all models $\mathcal{M}, \mathcal{M}'$ such that $\mathcal{M} \sqsubseteq \mathcal{M}'$.

**Example 4.8.** $f^{\tilde{\alpha}\to\tilde{\alpha}}\ x$ is constant. *Proof:* Let $V(x) = a_1$ and $V(f)(a_1) = a_2$. For any $\mathcal{M}' = (S', V')$ that extends $\mathcal{M} = (S, V)$, we have $V(x) \sqsubseteq^{\tilde{\alpha}} V'(x)$ and $V(f) \sqsubseteq^{\tilde{\alpha}\to\tilde{\alpha}} V'(f)$. By definition of $\sqsubseteq^{\sigma}$, $V'(x) = a_1$ and $V'(f)(a_1) = a_2$. Thus, $[\![f\ x]\!]_{\mathcal{M}} = [\![f\ x]\!]_{\mathcal{M}'} = a_2$. □

**Example 4.9.** $f^{o\to\tilde{\alpha}} = g$ is constant. *Proof:* For any $\mathcal{M}' = (S', V')$ that extends $\mathcal{M} = (S, V)$, we have $V(f) \sqsubseteq^{o\to\tilde{\alpha}} V'(f)$ and $V(g) \sqsubseteq^{o\to\tilde{\alpha}} V'(g)$. By definition of $\sqsubseteq^{\sigma}$, $V'(f) = V(f)$ and $V'(g) = V(g)$. Hence, $[\![f = g]\!]_{\mathcal{M}} = [\![f = g]\!]_{\mathcal{M}'}$. □

**Example 4.10.** $p^{\tilde{\alpha}\to o} = q$ is not constant. *Proof:* A counterexample is given by $|S(\tilde{\alpha})| = 1$, $V(p) = V(q) = (\top)$, $|S'(\tilde{\alpha})| = 2$, $V'(p) = (\top, \bot)$, $V'(q) = (\top, \top)$. Then $[\![p = q]\!]_{(S,V)} = \top$ but $[\![p = q]\!]_{(S',V')} = \bot$. □

Variables are always constant, and constancy is preserved by $\lambda$-abstraction and application. On the other hand, the equality symbol $=^{\sigma\to\sigma\to o}$ is constant only if $\tilde{\alpha}$ does not occur negatively in $\sigma$. Moreover, since $\sqsubseteq^{o}$ is the identity relation, constant formulas are both monotonic and antimonotonic.

As an aside, relations between models of the $\lambda$-calculus that are preserved under abstraction and application are called *logical relations* [130] and are widely used in semantics and model theory. If we had no equality, $\sqsubseteq$ would be a logical relation, and constancy of all terms would follow from the "Basic Lemma," which states

[1] i.e., total: $\forall a.\, \exists b.\ a \sqsubseteq^{\sigma} b$.

[2] i.e., injective: $\forall a\ a'\ b.\ a \sqsubseteq^{\sigma} b \wedge a' \sqsubseteq^{\sigma} b \longrightarrow a = a'$.

[3] i.e., surjective: $\forall b.\, \exists a.\ a \sqsubseteq^{\sigma} b$.

[4] i.e., functional: $\forall a\ b\ b'.\ a \sqsubseteq^{\sigma} b \wedge a \sqsubseteq^{\sigma} b' \longrightarrow b = b'$.

that the interpretations of any term are related by $\sim$ if $\sim$ is a logical relation. This property is spoiled by equality since in general $[\![=]\!]_{\mathcal{M}} \not\sqsubseteq [\![=]\!]_{\mathcal{M}'}$. Our calculus effectively carves out a sublanguage for which $\sqsubseteq$ is a logical relation.

### 4.2.2 Syntactic Criteria

We syntactically approximate constancy, monotonicity, and antimonotonicity with the predicates $\mathsf{K}(t)$, $\mathsf{M}^{+}(t)$, and $\mathsf{M}^{-}(t)$. The goal is to derive $\mathsf{M}^{+}(t)$ for the formula $t$ we wish to prove monotonic. If $\mathsf{M}^{+}(t)$ and the model $\mathcal{M}$ satisfies $t$, we can apply the recipe $\sqsubseteq$ to obtain arbitrarily larger models $\mathcal{M}'$ that also satisfy $t$. The predicates depend on the auxiliary functions $\mathsf{AT}^{+}(\sigma)$ and $\mathsf{AT}^{-}(\sigma)$, which collect the positive and negative atomic types of $\sigma$.

**Definition 4.11 (Positive and Negative Atomic Types).** The set of *positive atomic types* $\mathsf{AT}^{+}(\sigma)$ and the set of *negative atomic types* $\mathsf{AT}^{-}(\sigma)$ of a type $\sigma$ are defined as follows:

$$\begin{array}{ll} \mathsf{AT}^{+}(\alpha) = \{\alpha\} & \mathsf{AT}^{s}(o) = \emptyset \\ \mathsf{AT}^{-}(\alpha) = \emptyset & \mathsf{AT}^{s}(\sigma \to \tau) = \mathsf{AT}^{\sim s}(\sigma) \cup \mathsf{AT}^{s}(\tau) \end{array} \quad \text{where } \sim s = \begin{cases} + & \text{if } s = - \\ - & \text{if } s = + \end{cases}$$

**Definition 4.12 (Constancy and Monotonicity Rules).** The predicates $\mathsf{K}(t)$, $\mathsf{M}^{+}(t)$, and $\mathsf{M}^{-}(t)$ are inductively defined by the rules

$$\frac{}{\mathsf{K}(x)} \qquad \frac{}{\mathsf{K}(\longrightarrow)} \qquad \frac{\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)}{\mathsf{K}(=^{\sigma \to \sigma \to o})} \qquad \frac{\mathsf{K}(t)}{\mathsf{K}(\lambda x.\, t)} \qquad \frac{\mathsf{K}(t^{\sigma \to \tau}) \quad \mathsf{K}(u^{\sigma})}{\mathsf{K}(t\ u)}$$

$$\frac{\mathsf{K}(t)}{\mathsf{M}^{s}(t^{o})} \qquad \frac{\mathsf{M}^{\sim s}(t) \quad \mathsf{M}^{s}(u)}{\mathsf{M}^{s}(t \longrightarrow u)} \qquad \frac{\mathsf{K}(t^{\sigma}) \quad \mathsf{K}(u^{\sigma})}{\mathsf{M}^{-}(t = u)} \qquad \frac{\mathsf{M}^{-}(t)}{\mathsf{M}^{-}(\forall x.\, t)} \qquad \frac{\mathsf{M}^{+}(t) \quad \tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)}{\mathsf{M}^{+}(\forall x^{\sigma}.\, t)}$$

The rules for $\mathsf{K}$ simply traverse the term structure and ensure that equality is not used on types in which $\tilde{\alpha}$ occurs negatively. The rules for $\mathsf{M}^{s}$ are more subtle:

- The first $\mathsf{M}^{s}$ rule lets us derive (anti)monotonicity from constancy.
- The implication rule flips the sign $s$ when analyzing the assumption.
- The $\mathsf{M}^{-}(t = u)$ rule is sound because the extensions of distinct elements are always distinct (since $\sqsubseteq^{\sigma}$ is left-unique).
- The $\mathsf{M}^{-}(\forall x.\ t)$ rule is sound because if enlarging the scope makes $x$ range over new elements, these cannot make $\forall x.\ t$ become true if it was false in the smaller scope.
- The $\mathsf{M}^{+}(\forall x.\ t)$ rule is the most difficult one. If $\tilde{\alpha}$ does not occur at all in $\sigma$, then monotonicity is preserved. Otherwise, there is the danger that the formula $t$ is true for all values $a \in [\![\sigma]\!]_{S}$ but not for some $b \in [\![\sigma]\!]_{S'}$. However, in Section 4.3 we show that this can only happen for new $b$'s (i.e., $b$'s that do not extend any $a$), which can only exist if $\tilde{\alpha} \in \mathsf{AT}^{+}(\sigma)$.

**Definition 4.13 (Derived Monotonicity Rules).** For the other logical constants, we can derive the following rules using Definitions 2.6 and 4.12:

$$\frac{}{\mathsf{M}^{s}(\mathsf{False})} \qquad \frac{}{\mathsf{M}^{s}(\mathsf{True})} \qquad \frac{\mathsf{M}^{\sim s}(t)}{\mathsf{M}^{s}(\neg\, t)} \qquad \frac{\mathsf{M}^{s}(t) \quad \mathsf{M}^{s}(u)}{\mathsf{M}^{s}(t \wedge u)}$$

$$\dfrac{\mathsf{M}^s(t)\quad \mathsf{M}^s(u)}{\mathsf{M}^s(t \lor u)} \qquad \dfrac{\mathsf{M}^+(t)}{\mathsf{M}^+(\exists x.\ t)} \qquad \dfrac{\mathsf{M}^-(t)\quad \tilde{\alpha} \notin \mathsf{AT}^+(\sigma)}{\mathsf{M}^-(\exists x^\sigma.\ t)}$$

**Example 4.14.** The following derivations show that formulas 1 and 2 from Example 4.2 are monotonic with respect to $\alpha$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{K}(x)}\quad \dfrac{}{\mathsf{K}(y)}}{\mathsf{M}^-(x = y)}}{\mathsf{M}^+(x \neq y)}}{\mathsf{M}^+(\exists y.\ x \neq y)}}{\mathsf{M}^+(\exists x^\alpha\ y.\ x \neq y)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha \notin \mathsf{AT}^-(\alpha)}{\mathsf{K}(=^{\alpha\to\alpha\to o})}\quad \dfrac{\dfrac{}{\mathsf{K}(f)}\quad \dfrac{}{\mathsf{K}(x)}}{\mathsf{K}(f\,x)}}{\mathsf{K}((=)\ (f\,x))}\quad \dfrac{}{\mathsf{K}(x)}}{\mathsf{K}(f\,x = x)}}{\mathsf{M}^+(f\,x = x)}\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{K}(f)}\quad \dfrac{}{\mathsf{K}(y)}}{\mathsf{K}(f\,y)}\quad \dfrac{}{\mathsf{K}(y)}}{\mathsf{M}^-(f\,y = y)}}{\mathsf{M}^+(f\,y \neq y)}}{\mathsf{M}^+(f\,x^\alpha = x \land f\,y \neq y)}$$

Similarly, $\mathsf{M}^+$ judgments can be derived for the monkey village axioms of Example 4.3 to show monotonicity with respect to *banana*.

**Example 4.15.** Formula 4 from Example 4.2 is monotonic, but $\mathsf{M}^+$ fails on it:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha \notin \mathsf{AT}^-(\alpha \to o)}{\mathsf{K}(=^{(\alpha\to o)\to(\alpha\to o)\to o})}\quad \dfrac{\vdots}{\mathsf{K}(\{y\})}}{\mathsf{K}((=)\ \{y\})}\quad \dfrac{\vdots}{\mathsf{K}(\{z\})}}{\mathsf{K}(\{y\} = \{z\})}}{\mathsf{M}^+(\{y\} = \{z\})}$$

The assumption $\alpha \notin \mathsf{AT}^-(\alpha \to o)$ cannot be discharged, since $\mathsf{AT}^-(\alpha \to o) = \{\alpha\}$. The formula fares no better if we put it in extensional form ($\forall x.\ (x = y) = (x = z)$).

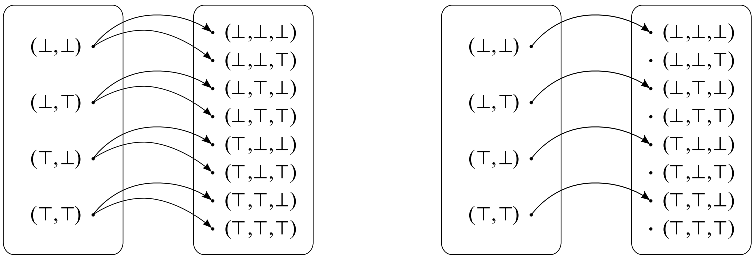
## 4.3 Second Calculus: Tracking Sets

Example 4.15 exhibits a significant weakness of the calculus $\mathfrak{M}_1$. Isabelle/HOL identifies sets with predicates, yet the predicate $\mathsf{M}^+$ prevents us from comparing terms of type $\tilde{\alpha} \to o$ for equality. This restriction is necessary because the extension of a function of this type is not unique (cf. Figure 4.1(c)), and thus equality is generally not preserved as we enlarge the scope.

This behavior of $\sqsubseteq^{\tilde{\alpha}\to o}$ is imprecise for sets, as it puts distinct sets in relation; for example, $\{0\} \sqsubseteq^{\tilde{\alpha}\to o} \{0,1\}$ if $S(\tilde{\alpha}) = 1$ and $S'(\tilde{\alpha}) = 2$. We would normally prefer each set to admit a unique extension, namely the set itself. This would make set equality constant.

To solve the problem sketched above, we could adjust the definition of $\sqsubseteq^\sigma$ so that the extension of a set is always the set itself. Rephrased in terms of functions, this amounts to requiring that the extended function returns $\bot$ for all new elements. Figure 4.2 compares this more conservative "set" approach to the liberal approach of Section 4.2; in subfigure (b), it may help to think of $(\bot,\bot)$ and $(\bot,\bot,\bot)$ as $\emptyset$, $(\top,\bot)$ and $(\top,\bot,\bot)$ as $\{0\}$, and so on.

With this approach, we could easily infer that $\{y\} = \{z\}$ is constant. However, the wholesale application of this principle would have pernicious consequences



**(a)** general view (right-total)

**(b)** set view (right-unique)

**Figure 4.2:** $\sqsubseteq^{\tilde{\alpha}\to o}$ with $S(\tilde{\alpha}) = 2$ and $S'(\tilde{\alpha}) = 3$

on constancy: Semantically, the universal set $\mathsf{UNIV}^{\tilde{\alpha}\to o}$, among others, would no longer be constant; syntactically, the introduction rule for $\mathsf{K}(\lambda x.\ t)$ would no longer be sound. What we propose instead in our calculus $\mathfrak{M}_2$ is a hybrid approach that supports both forms of extensions in various combinations. The required bookkeeping is conveniently expressed as a type system, in which each function arrow is annotated with $\mathsf{G}$ ("general") or $\mathsf{F}$ ("false-extended set").

**Definition 4.16 (Annotated Type).** An *annotated type* is a HOL type in which each function arrow carries an *annotation* $A \in \{\mathsf{G}, \mathsf{F}\}$.

The annotations specify how $\sqsubseteq$ should extend function values to larger scopes: While $\mathsf{G}$-functions are extended as in the previous section, the extension of an $\mathsf{F}$-function must map all new values to $\bot$. The annotations have no influence on the interpretation of types and terms, which is unchanged. For notational convenience, we sometimes use annotated types in contexts where plain types are expected; in such cases, the annotations are simply ignored.

### 4.3.1 Extension Relation

**Definition 4.17 (Extension).** Let $\sigma$ be an annotated type, and let $S$, $S'$ be scopes such that $S \leq_{\tilde{\alpha}} S'$. The *extension* relation $\sqsubseteq^{\sigma} \subseteq [\![\sigma]\!]_S \times [\![\sigma]\!]_{S'}$ for $S$ and $S'$ is defined by the following equivalences:

$$\begin{array}{rll}
a \sqsubseteq^{\sigma} b & \text{iff} & a = b \qquad \text{if } \sigma \text{ is } o \text{ or an atomic type} \\
f \sqsubseteq^{\sigma \to_{\mathsf{G}} \tau} g & \text{iff} & \forall a\, b.\ a \sqsubseteq^{\sigma} b \longrightarrow f(a) \sqsubseteq^{\tau} g(b) \\
f \sqsubseteq^{\sigma \to_{\mathsf{F}} \tau} g & \text{iff} & \forall a\, b.\ a \sqsubseteq^{\sigma} b \longrightarrow f(a) \sqsubseteq^{\tau} g(b) \ \text{ and} \\
 & & \forall b.\ (\nexists a.\ a \sqsubseteq^{\sigma} b) \longrightarrow g(b) = \llparenthesis \tau \rrparenthesis
\end{array}$$

where $\llparenthesis o \rrparenthesis = \bot$, $\llparenthesis \sigma \to \tau \rrparenthesis = a \in [\![\sigma]\!]_{S'} \mapsto \llparenthesis \tau \rrparenthesis$, and $\llparenthesis \alpha \rrparenthesis$ is any element of $S(\alpha)$.

The extension relation $\sqsubseteq^{\sigma}$ distinguishes between the two kinds of arrow. The $\mathsf{G}$ case coincides with Definition 4.5. Although $\mathsf{F}$ is tailored to predicates, the annotated type $\sigma \to_{\mathsf{F}} \tau$ is legal for any type $\tau$. The value $\llparenthesis \tau \rrparenthesis$ then takes the place of $\bot$ as the default extension.

We now prove the crucial properties of $\sqsubseteq^{\sigma}$, which we introduced in Section 4.2 for the $\mathsf{G}$ case.

**Lemma 4.18.** *The relation $\sqsubseteq^{\sigma}$ is left-total* (*i.e., total*) *and left-unique* (*i.e., injective*)*.*

*Proof* (structural induction on $\sigma$). For $o$ and $\alpha$, both properties are obvious. For $\sigma \rightarrow_{A} \tau$, $\sqsubseteq^{\sigma}$ and $\sqsubseteq^{\tau}$ are left-unique and left-total by induction hypothesis. Since $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} \subseteq \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$ by definition, it suffices to show that $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau}$ is left-total and $\sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$ is left-unique.

LEFT-TOTALITY: For $f \in [\![\sigma \rightarrow \tau]\!]_{S}$, we find an extension $g$ as follows: Let $b \in [\![\sigma]\!]_{S'}$. If $b$ extends an $a$, that $a$ is unique by left-uniqueness of $\sqsubseteq^{\sigma}$. Since $\sqsubseteq^{\tau}$ is left-total, there exists a $y$ such that $f(a) \sqsubseteq^{\tau} y$, and we let $g(b) = y$. If $b$ does not extend any $a$, then we set $g(b) = (\!|\tau|\!)$. By construction, $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} g$.

LEFT-UNIQUENESS: We assume $f, f' \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} g$ and show that $f = f'$. For every $a \in [\![\sigma]\!]_{S}$, left-totality of $\sqsubseteq^{\sigma}$ yields an extension $b \sqsupseteq^{\sigma} a$. Then $f(a) \sqsubseteq^{\tau} g(b)$ and $f'(a) \sqsubseteq^{\tau} g(b)$, and since $\sqsubseteq^{\tau}$ is left-unique, $f(a) = f'(a)$. □

**Definition 4.19 (Positive and Negative Atomic Types).** The set of *positive atomic types* $\mathsf{AT}^{+}(\sigma)$ and the set of *negative atomic types* $\mathsf{AT}^{-}(\sigma)$ of an annotated type $\sigma$ are defined as follows:

$$\begin{aligned}
\mathsf{AT}^{+}(o) &= \emptyset & \mathsf{AT}^{-}(o) &= \emptyset \\
\mathsf{AT}^{+}(\alpha) &= \{\alpha\} & \mathsf{AT}^{-}(\alpha) &= \emptyset \\
\mathsf{AT}^{+}(\sigma \rightarrow_{\mathsf{G}} \tau) &= \mathsf{AT}^{+}(\tau) \cup \mathsf{AT}^{-}(\sigma) & \mathsf{AT}^{-}(\sigma \rightarrow_{\mathsf{G}} \tau) &= \mathsf{AT}^{-}(\tau) \cup \mathsf{AT}^{+}(\sigma) \\
\mathsf{AT}^{+}(\sigma \rightarrow_{\mathsf{F}} \tau) &= \mathsf{AT}^{+}(\tau) \cup \mathsf{AT}^{-}(\sigma) \cup \mathsf{AT}^{+}(\sigma) & \mathsf{AT}^{-}(\sigma \rightarrow_{\mathsf{F}} \tau) &= \mathsf{AT}^{-}(\tau)
\end{aligned}$$

This peculiar generalization of Definition 4.11 reflects our wish to treat occurrences of $\tilde{\alpha}$ differently for sets and ensures that the following key lemma holds uniformly.

**Lemma 4.20.** *If $\tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)$, then $\sqsubseteq^{\sigma}$ is right-total* (*i.e., surjective*)*. If $\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)$, then $\sqsubseteq^{\sigma}$ is right-unique* (*i.e., functional*)*.*

*Proof* (structural induction on $\sigma$). For $o$ and $\alpha$, both properties are obvious.

RIGHT-TOTALITY OF $\sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$: If $\tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma \rightarrow_{\mathsf{G}} \tau) = \mathsf{AT}^{+}(\tau) \cup \mathsf{AT}^{-}(\sigma)$, then by induction hypothesis $\sqsubseteq^{\tau}$ is right-total and $\sqsubseteq^{\sigma}$ is right-unique. For $g \in [\![\sigma \rightarrow \tau]\!]_{S'}$, we find a restriction $f$ as follows: Let $a \in [\![\sigma]\!]_{S}$. Since $\sqsubseteq^{\sigma}$ is both left-total (Lemma 4.18) and right-unique, there is a unique $b$ such that $a \sqsubseteq^{\sigma} b$. By right-totality of $\sqsubseteq^{\tau}$, we obtain an $x \sqsubseteq^{\tau} g(b)$, and we set $f(a) = x$. By construction, $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} g$.

RIGHT-TOTALITY OF $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau}$: If $\tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma \rightarrow_{\mathsf{F}} \tau) = \mathsf{AT}^{+}(\tau) \cup \mathsf{AT}^{-}(\sigma) \cup \mathsf{AT}^{+}(\sigma)$, then by induction hypothesis $\sqsubseteq^{\tau}$ is right-total, and $\sqsubseteq^{\sigma}$ is both right-total and right-unique. By right-totality of $\sqsubseteq^{\sigma}$, the second condition in the definition of $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau}$ becomes vacuous, and $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} = \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$, whose right-totality was shown above.

RIGHT-UNIQUENESS OF $\sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$: If $\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma \rightarrow_{\mathsf{G}} \tau) = \mathsf{AT}^{-}(\tau) \cup \mathsf{AT}^{+}(\sigma)$, then by induction hypothesis $\sqsubseteq^{\tau}$ is right-unique and $\sqsubseteq^{\sigma}$ is right-total. We consider $g, g'$ such that $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} g$ and $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} g'$, and show that $g = g'$. For every $b \in [\![\sigma]\!]_{S'}$, right-totality of $\sqsubseteq^{\sigma}$ yields a restriction $a \sqsubseteq^{\sigma} b$. Then $f(a) \sqsubseteq^{\tau} g(b)$ and $f(a) \sqsubseteq^{\tau} g'(b)$, and since $\sqsubseteq^{\tau}$ is right-unique, $g(b) = g'(b)$.

RIGHT-UNIQUENESS OF $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau}$: If $\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma \rightarrow_{\mathsf{F}} \tau) = \mathsf{AT}^{-}(\tau)$, then by induction hypothesis $\sqsubseteq^{\tau}$ is right-unique. We consider $g, g'$ such that $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} g$ and $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau}$

$g'$, and show that $g = g'$. For any $b \in [\![\sigma]\!]_{S'}$, if there exists no restriction $a \sqsubseteq^{\sigma} b$, then by definition $g(b) = g'(b) = (\!|\tau|\!)$. Otherwise, we assume $a \sqsubseteq^{\sigma} b$. Then $f(a) \sqsubseteq^{\tau} g(b)$ and $f(a) \sqsubseteq^{\tau} g'(b)$, and since $\sqsubseteq^{\tau}$ is right-unique, $g(b) = g'(b)$. □

The new definition of $\mathsf{AT}^{+}$ and $\mathsf{AT}^{-}$ solves the problem raised by $\{y\} = \{z\}$ in Example 4.15, by counting the $\tilde{\alpha}$ in $\tilde{\alpha} \rightarrow_{\mathsf{F}} o$ as a positive occurrence. However, we must ensure that types are consistently annotated; otherwise, we could easily overconstrain the free variables and end up in a situation where there exists no model $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq \mathcal{M}'$ for two scopes $S \leq_{\tilde{\alpha}} S'$.

### 4.3.2 Type Checking

Checking constancy can be seen as a type checking problem involving annotated types. The main idea is to derive typing judgments $\Gamma \vdash t : \sigma$, whose intuitive meaning is that the denotations of $t$ in a smaller and a larger scope are related by $\sqsubseteq^{\sigma}$ (i.e., that $t$ is constant in a sense given by $\sigma$). Despite this new interpretation, the typing rules are similar to those of the simply typed $\lambda$-calculus, extended with a particular form of subtyping.

**Definition 4.21 (Context).** A *context* is a pair of mappings $\Gamma = (\Gamma_{\mathrm{c}}, \Gamma_{\mathrm{v}})$, where $\Gamma_{\mathrm{c}}$ maps constant symbols to sets of annotated types, and $\Gamma_{\mathrm{v}}$ maps variables to annotated types.

Allowing constants to have multiple annotated types gives us a form of polymorphism on the annotations, which is sometimes useful.

**Definition 4.22 (Compatibility).** A constant context $\Gamma_{\mathrm{c}}$ is *compatible* with a constant model $M$ if $\sigma \in \Gamma_{\mathrm{c}}(\mathsf{c})$ implies $M_S(\mathsf{c}) \sqsubseteq^{\sigma} M_{S'}(\mathsf{c})$ for all scopes $S$, $S'$ with $S \leq_{\tilde{\alpha}} S'$ and for all constants $\mathsf{c}$ and annotated types $\sigma$.

*Convention.* In the sequel, we consistently use a fixed constant context $\Gamma_{\mathrm{c}}$ compatible with the standard constant model $\widehat{M}$, allowing us to omit the first component of $\Gamma = (\Gamma_{\mathrm{c}}, \Gamma_{\mathrm{v}})$.

Definitions 4.6 and 4.7 and the K part of Definition 4.12 are generalized as follows.

**Definition 4.23 (Model Extension).** Let $\mathcal{M} = (S, V)$ and $\mathcal{M}' = (S', V')$ be models. The model $\mathcal{M}'$ *extends* $\mathcal{M}$ in a context $\Gamma$, written $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$, if $S \leq_{\tilde{\alpha}} S'$ and $\Gamma(x) = \sigma$ implies $V(x) \sqsubseteq^{\sigma} V'(x)$ for all $x$.

**Definition 4.24 (Constancy).** Let $\sigma$ be an annotated type. A term $t$ is *$\sigma$-constant* in a context $\Gamma$ if $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![t]\!]_{\mathcal{M}'}$ for all models $\mathcal{M}, \mathcal{M}'$ such that $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$.

**Definition 4.25 (Typing Rules).** The *typing* relation $\Gamma \vdash t : \sigma$ is given by the rules

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}\,\textsc{Var} \qquad \frac{\sigma \in \Gamma_{\mathrm{c}}(\mathsf{c})}{\Gamma \vdash \mathsf{c} : \sigma}\,\textsc{Const} \qquad \frac{\Gamma \vdash t : \sigma \quad \sigma \leq \sigma'}{\Gamma \vdash t : \sigma'}\,\textsc{Sub}$$

$$\frac{\Gamma[x \mapsto \sigma] \vdash t : \tau}{\Gamma \vdash \lambda x.\, t : \sigma \rightarrow_{\mathsf{G}} \tau}\,\textsc{Lam} \qquad \frac{\Gamma \vdash t : \sigma \rightarrow_{A} \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\, u : \tau}\,\textsc{App}$$

where the *subtype* relation $\sigma \leq \tau$ is defined by the rules

$$\frac{}{\sigma \leq \sigma}\,\textsc{Refl} \qquad \frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \rightarrow_{\mathsf{A}} \tau \leq \sigma' \rightarrow_{\mathsf{G}} \tau'}\,\textsc{Gen} \qquad \frac{\sigma' \leq \sigma \quad \sigma \leq \sigma' \quad \tau \leq \tau'}{\sigma \rightarrow_{\mathsf{F}} \tau \leq \sigma' \rightarrow_{\mathsf{F}} \tau'}\,\textsc{False}$$

**Lemma 4.26.** *If $\sigma \leq \sigma'$, then $\sqsubseteq^{\sigma} \subseteq \sqsubseteq^{\sigma'}$.*

*Proof* (induction on the derivation of $\sigma \leq \sigma'$). The REFL case is trivial.

GEN: We may assume $\sqsubseteq^{\sigma'} \subseteq \sqsubseteq^{\sigma}$ and $\sqsubseteq^{\tau} \subseteq \sqsubseteq^{\tau'}$ and must show $\sqsubseteq^{\sigma \rightarrow \tau} \subseteq \sqsubseteq^{\sigma' \rightarrow \tau'}$. Since $\sqsubseteq^{\sigma \rightarrow_{A} \tau} \subseteq \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$ (by Definition 4.17), it is sufficient to consider the case $A =$ F. From $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} g$, we have $a \sqsubseteq^{\sigma} b \Longrightarrow f(a) \sqsubseteq^{\tau} g(b)$; and using the assumptions we conclude $a \sqsubseteq^{\sigma'} b \Longrightarrow f(a) \sqsubseteq^{\tau'} g(b)$, i.e., $f \sqsubseteq^{\sigma' \rightarrow_{\mathsf{G}} \tau'} g$.

FALSE: We may assume $\sqsubseteq^{\sigma} = \sqsubseteq^{\sigma'}$ and $\sqsubseteq^{\tau} \subseteq \sqsubseteq^{\tau'}$. If $f \sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} g$, the first condition for $f \sqsubseteq^{\sigma' \rightarrow_{\mathsf{F}} \tau'} g$ follows for the same reason as above. The second condition holds since $\sqsubseteq^{\sigma} = \sqsubseteq^{\sigma'}$. □

As a consequence of Lemma 4.26, $\leq$ is a partial quasiorder. Observe that $\sigma \rightarrow_{\mathsf{F}} o \leq \sigma \rightarrow_{\mathsf{G}} o$ for any $\sigma$, and if $\tilde{\alpha}$ does not occur in $\sigma$ we also have $\sigma \rightarrow_{\mathsf{G}} o \leq \sigma \rightarrow_{\mathsf{F}} o$. On the other hand, $(\tilde{\alpha} \rightarrow_{\mathsf{G}} o) \rightarrow_{\mathsf{G}} o$ and $(\tilde{\alpha} \rightarrow_{\mathsf{F}} o) \rightarrow_{\mathsf{F}} o$ are not related, so the quasiorder $\leq$ is not total (linear).

**Theorem 4.27 (Soundness of Typing).** *If $\Gamma \vdash t : \sigma$, then $t$ is $\sigma$-constant in $\Gamma$.*

*Proof* (induction on the derivation of $\Gamma \vdash t : \sigma$). VAR: Because $V(x) \sqsubseteq^{\sigma} V'(x)$ by assumption for $\sigma = \Gamma(x)$.

CONST: By compatibility of $\Gamma_{\mathsf{c}}$, $\widehat{M}_S(\mathsf{c}) \sqsubseteq^{\sigma} \widehat{M}_{S'}(\mathsf{c})$ for all $\sigma \in \Gamma_{\mathsf{c}}(\mathsf{c})$.

SUB: By Lemma 4.26 and Definition 4.24.

LAM: Let $a \in [\![\sigma]\!]_S$ and $b \in [\![\sigma]\!]_{S'}$ such that $a \sqsubseteq^{\sigma} b$. Then we have the altered models $\mathcal{M}_a = (S, V[x \mapsto a])$ and $\mathcal{M}'_b = (S', V'[x \mapsto b])$. Thus, $\mathcal{M}_a \sqsubseteq_{\Gamma[x \mapsto \sigma]} \mathcal{M}'_b$, and by induction hypothesis $[\![\lambda x.\ t]\!]_{\mathcal{M}}(a) = [\![t]\!]_{\mathcal{M}_a} \sqsubseteq^{\tau} [\![t]\!]_{\mathcal{M}'_b} = [\![\lambda x.\ t]\!]_{\mathcal{M}'}(b)$. Hence $[\![\lambda x.\ t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} [\![\lambda x.\ t]\!]_{\mathcal{M}'}$.

APP: By induction hypothesis, and since $\sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} \subseteq \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau}$, we have $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma \rightarrow_{\mathsf{G}} \tau} [\![t]\!]_{\mathcal{M}'}$ and $[\![u]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![u]\!]_{\mathcal{M}'}$. By Definition 4.17, we have $[\![t\ u]\!]_{\mathcal{M}} = [\![t]\!]_{\mathcal{M}}\,([\![u]\!]_{\mathcal{M}}) \sqsubseteq^{\tau} [\![t]\!]_{\mathcal{M}'}\,([\![u]\!]_{\mathcal{M}'}) = [\![t\ u]\!]_{\mathcal{M}'}$, which shows that $t\ u$ is $\tau$-constant in $\Gamma$. □

Notice a significant weakness here: While our typing rules propagate F-annotations nicely, they cannot derive them, since the LAM rule annotates all arrows with G. In particular, the basic set operations $\emptyset$, $\cup$, $\cap$, and $-$ cannot be typed appropriately from their definitions (Definition 2.7). We solve this issue pragmatically by treating common set operations as primitive constants along with implication and equality. The typing rules then propagate type information through expressions such as $A \cup (B \cap C)$. We address this limitation more generally in Section 4.4.

**Definition 4.28 (Standard Constant Context).** The *standard constant context* $\widehat{\Gamma}_{\mathrm{c}}$ is the following mapping:

$$\begin{array}{rcl}
\longrightarrow & \mapsto & \{o \rightarrow_{\mathsf{G}} o \rightarrow_{\mathsf{G}} o\} \\
= & \mapsto & \{\sigma \rightarrow_{\mathsf{G}} \sigma \rightarrow_{\mathsf{F}} o \mid \tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)\} \\
\emptyset & \mapsto & \{\sigma \rightarrow_{\mathsf{F}} o\} \\
\mathsf{UNIV} & \mapsto & \{\sigma \rightarrow_{\mathsf{G}} o\} \\
\cup,\, \cap & \mapsto & \{(\sigma \rightarrow_{A} o) \rightarrow_{\mathsf{G}} (\sigma \rightarrow_{A} o) \rightarrow_{\mathsf{G}} \sigma \rightarrow_{A} o \mid A \in \{\mathsf{G}, \mathsf{F}\}\} \\
- & \mapsto & \{(\sigma \rightarrow_{A} o) \rightarrow_{\mathsf{G}} (\sigma \rightarrow_{\mathsf{G}} o) \rightarrow_{\mathsf{G}} \sigma \rightarrow_{A} o \mid A \in \{\mathsf{G}, \mathsf{F}\}\} \\
\in & \mapsto & \{\sigma \rightarrow_{\mathsf{G}} (\sigma \rightarrow_{A} o) \rightarrow_{\mathsf{G}} o \mid A \in \{\mathsf{G}, \mathsf{F}\}\} \\
\mathsf{insert} & \mapsto & \{\sigma \rightarrow_{\mathsf{G}} (\sigma \rightarrow_{A} o) \rightarrow_{\mathsf{G}} \sigma \rightarrow_{A} o \mid A \in \{\mathsf{G}, \mathsf{F}\},\ \tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)\}
\end{array}$$

Notice how the lack of a specific annotation for "true-extended sets" prevents us from giving precise typings to the complement of an F-annotated function; for example, $\emptyset$ is captured precisely by $\sigma \rightarrow_{\mathsf{F}} o$, but UNIV can be typed only as $\sigma \rightarrow_{\mathsf{G}} o$. In Section 4.4, we introduce a T-annotation similar to F but with $\top$ instead of $\bot$ as the default extension.

**Lemma 4.29.** *The standard constant context $\widehat{\Gamma}_{\mathrm{c}}$ is compatible with the standard constant model $\widehat{M}$.*

*Proof.* CASE $\longrightarrow$: Obvious.

CASE $=$: Since $\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)$, $\sqsubseteq^{\sigma}$ is right-unique (Lemma 4.20). Unfolding the definition of $\sqsubseteq$, we assume $a \sqsubseteq^{\sigma} b$ and show that if $a' \sqsubseteq^{\sigma} b'$, then $(a = a') = (b = b')$, and that if there exists no restriction $a'$ such that $a' \sqsubseteq^{\sigma} b'$, then $(b = b') = \bot$. The first part follows from the left-uniqueness and right-uniqueness of $\sqsubseteq^{\sigma}$. For the second part, $b \neq b'$ because $b$ extends $a$ while $b'$ admits no restriction.

CASE $\emptyset$: Obvious, since $\llbracket \lambda x.\ \mathsf{False} \rrbracket_{\mathcal{M}} = \emptyset \sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} o} \emptyset = \llbracket \lambda x.\ \mathsf{False} \rrbracket_{\mathcal{M}'}$. (For notational convenience, we identify sets with predicates in the semantics.)

CASES UNIV, $\in$, AND insert: The typings are derivable from the constants' definitions using the rules of Definition 4.25, and $\sigma$-constancy follows from Theorem 4.27.

CASE $\cup$: The subcase $A = \mathsf{G}$ follows from the derivability of $\Gamma \vdash \lambda s\, t\, x.\ s\ x \vee t\ x\ :\ \sigma$ and Theorem 4.27. Otherwise, $A = \mathsf{F}$ and we let $\tau = \sigma \rightarrow_{\mathsf{F}} o$. We have $\llbracket \lambda s\, t\, x.\ s\ x \vee t\ x \rrbracket_{\mathcal{M}} = (A, A') \in \llbracket \sigma \rightarrow o \rrbracket_{S}^{2} \mapsto A \cup A' \sqsubseteq^{\tau \rightarrow_{\mathsf{G}} \tau \rightarrow_{\mathsf{G}} o} (B, B') \in \llbracket \sigma \rightarrow o \rrbracket_{S'}^{2} \mapsto B \cup B' = \llbracket \lambda s\, t\, x.\ s\ x \vee t\ x \rrbracket_{\mathcal{M}'}$, because if $B$ and $B'$ map all new elements $b \in \llbracket \sigma \rrbracket_{S'}$ to $\bot$ (as required by the typing $\sigma \rightarrow_{\mathsf{F}} o$), then $B \cup B'$ also maps $b$ to $\bot$.

CASE $\cap$: Similar to $\cup$.

CASE $-$: The subcase $A = \mathsf{G}$ follows from the derivability of $\Gamma \vdash \lambda s\, t\, x.\ s\ x \wedge \neg\, t\ x\ :$ $(\sigma \rightarrow_{\mathsf{G}} o) \rightarrow_{\mathsf{G}} (\sigma \rightarrow_{\mathsf{G}} o) \rightarrow_{\mathsf{G}} s \rightarrow_{\mathsf{G}} o$ and Theorem 4.27. Otherwise, $\llbracket \lambda s\ t\ x.\ s\ x \vee t\ x \rrbracket_{\mathcal{M}} = (A, A') \in \llbracket \tau \rightarrow o \rrbracket_{S}^{2} \mapsto A - A' \sqsubseteq^{(\sigma \rightarrow_{\mathsf{F}} o) \rightarrow_{\mathsf{G}} (\sigma \rightarrow_{\mathsf{G}} o) \rightarrow_{\mathsf{G}} \sigma \rightarrow_{\mathsf{F}} o} (B, B') \in \llbracket \tau \rightarrow o \rrbracket_{S'}^{2} \mapsto B - B' = \llbracket \lambda s\, t\, x.\ s\ x - t\ x \rrbracket_{\mathcal{M}'}$, because if $B$ maps all elements $b \in \llbracket \tau \rrbracket_{S'}$ that do not extend any $a \in \llbracket \tau \rrbracket_{S}$ to $\bot$ (as required by the typing $\sigma \rightarrow_{\mathsf{F}} o$), then $B - B'$ also maps $b$ to $\bot$. □

**Example 4.30.** Let $\sigma = \alpha \rightarrow_{\mathsf{F}} o$ and $\Gamma_{\mathrm{v}} = [x \mapsto \sigma,\ y \mapsto \sigma]$. The following derivation shows that $x^{\alpha \rightarrow o} = y$ is constant with respect to $\alpha$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\sigma \to_{\mathsf{G}} \sigma \to_{\mathsf{F}} o \in \Gamma_{\mathsf{c}}(=)}}{\Gamma \vdash (=) : \sigma \to_{\mathsf{G}} \sigma \to_{\mathsf{F}} o}\,\textsc{Const} \qquad \dfrac{\dfrac{}{\Gamma(x) = \sigma}}{\Gamma \vdash x : \sigma}\,\textsc{Var}}{\Gamma \vdash (=)\; x : \sigma \to_{\mathsf{G}} o}\,\textsc{App} \qquad \dfrac{\dfrac{}{\Gamma(y) = \sigma}}{\Gamma \vdash y : \sigma}\,\textsc{Var}}{\Gamma \vdash x = y : o}\,\textsc{App}$$

### 4.3.3 Monotonicity Checking

The rules for checking monotonicity and antimonotonicity are analogous to those presented in Section 4.2, except that they must now extend the context when moving under a quantifier.

**Definition 4.31 (Monotonicity Rules).** The predicates $\Gamma \vdash \mathsf{M}^{+}(t)$ and $\Gamma \vdash \mathsf{M}^{-}(t)$ are given by the rules

$$\dfrac{\Gamma \vdash t : o}{\Gamma \vdash \mathsf{M}^{s}(t)}\,\textsc{Term} \qquad \dfrac{\Gamma \vdash \mathsf{M}^{\sim s}(t) \quad \Gamma \vdash \mathsf{M}^{s}(u)}{\Gamma \vdash \mathsf{M}^{s}(t \longrightarrow u)}\,\textsc{Imp} \qquad \dfrac{\Gamma \vdash t : \sigma \quad \Gamma \vdash u : \sigma}{\Gamma \vdash \mathsf{M}^{-}(t = u)}\,\textsc{Eq}^{-}$$

$$\dfrac{\Gamma[x \mapsto \sigma] \vdash \mathsf{M}^{-}(t)}{\Gamma \vdash \mathsf{M}^{-}(\forall x.\, t)}\,\textsc{All}^{-} \qquad \dfrac{\Gamma[x \mapsto \sigma] \vdash \mathsf{M}^{+}(t) \quad \tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)}{\Gamma \vdash \mathsf{M}^{+}(\forall x.\, t)}\,\textsc{All}^{+}$$

**Definition 4.32 (Derived Monotonicity Rules).** From Definitions 2.6, 4.25, and 4.31, we derive the following rules for logical constants:

$$\dfrac{}{\Gamma \vdash \mathsf{M}^{s}(\mathsf{False})}\,\textsc{False} \qquad \dfrac{}{\Gamma \vdash \mathsf{M}^{s}(\mathsf{True})}\,\textsc{True} \qquad \dfrac{\Gamma \vdash \mathsf{M}^{\sim s}(t)}{\Gamma \vdash \mathsf{M}^{s}(\neg\, t)}\,\textsc{Not}$$

$$\dfrac{\Gamma \vdash \mathsf{M}^{s}(t) \quad \Gamma \vdash \mathsf{M}^{s}(u)}{\Gamma \vdash \mathsf{M}^{s}(t \wedge u)}\,\textsc{And} \qquad \dfrac{\Gamma \vdash \mathsf{M}^{s}(t) \quad \Gamma \vdash \mathsf{M}^{s}(u)}{\Gamma \vdash \mathsf{M}^{s}(t \vee u)}\,\textsc{Or}$$

$$\dfrac{\Gamma[x \mapsto \sigma] \vdash \mathsf{M}^{+}(t)}{\Gamma \vdash \mathsf{M}^{+}(\exists x^{\sigma}.\, t)}\,\textsc{Ex}^{+} \qquad \dfrac{\Gamma[x \mapsto \sigma] \vdash \mathsf{M}^{-}(t) \quad \tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)}{\Gamma \vdash \mathsf{M}^{-}(\exists x^{\sigma}.\, t)}\,\textsc{Ex}^{-}$$

**Theorem 4.33 (Soundness of $\mathsf{M}^{s}$).** *Let $\mathcal{M}$ and $\mathcal{M}'$ be models such that $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$. If $\Gamma \vdash \mathsf{M}^{+}(t)$, then $\mathcal{M} \vDash t$ implies $\mathcal{M}' \vDash t$. If $\Gamma \vdash \mathsf{M}^{-}(t)$, then $\mathcal{M} \nvDash t$ implies $\mathcal{M}' \nvDash t$.*

*Proof* (induction on the derivation of $\Gamma \vdash \mathsf{M}^{s}(t)$). The Term and Imp cases are obvious. Let $\mathcal{M} = (S, V)$ and $\mathcal{M}' = (S', V')$.

Eq$^{-}$: Assume $\Gamma \vdash t : \sigma$, $\Gamma \vdash u : \sigma$, and $\mathcal{M} \nvDash t = u$. Since $\mathcal{M}$ is a standard model, we know that $[\![t]\!]_{\mathcal{M}} \neq [\![u]\!]_{\mathcal{M}}$. By Theorem 4.27, we have $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![t]\!]_{\mathcal{M}'}$ and $[\![u]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![u]\!]_{\mathcal{M}'}$. By the left-uniqueness of $\sqsubseteq^{\sigma}$, the extensions cannot be equal, and thus $\mathcal{M}' \nvDash t = u$.

All$^{-}$: Assume $\Gamma[x \mapsto \sigma] \vdash \mathsf{M}^{-}(t)$ and $\mathcal{M} \nvDash \forall x^{\sigma}.\ t$. Then there exists $a \in [\![\sigma]\!]_{S}$ such that $(S, V[x \mapsto a]) \nvDash t$. Since $\sqsubseteq^{\sigma}$ is left-total, there exists an extension $b \sqsupseteq^{\sigma} a$. Since $(S, V[x \mapsto a]) \sqsubseteq_{\Gamma} (S', V'[x \mapsto b])$, we have $(S', V'[x \mapsto b]) \nvDash t$ by induction hypothesis. Thus $\mathcal{M}' \nvDash \forall x^{\sigma}.\ t$.

All$^{+}$: Assume $\Gamma[x \mapsto \sigma] \vdash \mathsf{M}^{+}(t)$, $\tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)$, and $\mathcal{M} \vDash \forall x^{\sigma}.\ t$. We show that $\mathcal{M}' \vDash \forall x^{\sigma}.\ t$. Let $b \in [\![\sigma]\!]_{S'}$. Since $\sqsubseteq^{\sigma}$ is right-total (Lemma 4.20), there exists a restriction $a \in [\![\sigma]\!]_S$ with $a \sqsubseteq^{\sigma} b$. By assumption, $(S, V[x \mapsto a]) \vDash t$. Since $(S, V[x \mapsto a]) \sqsubseteq_\Gamma (S', V'[x \mapsto b])$, we have $(S', V'[x \mapsto b]) \vDash t$ by induction hypothesis. Thus $\mathcal{M}' \vDash \forall x^{\sigma}.\, t$. □

**Theorem 4.34 (Soundness of the Calculus).** *If* $\Gamma \vdash \mathsf{M}^{+}(t)$ *can be derived in some arbitrary context* $\Gamma$*, then* $t$ *is monotonic. If* $\Gamma \vdash \mathsf{M}^{-}(t)$ *can be derived in some arbitrary context* $\Gamma$*, then* $t$ *is antimonotonic.*

*Proof.* The definition of monotonicity requires showing the existence of a model $\mathcal{M}' = (S', V')$ for any scope $S'$ such that $S \leq_{\tilde{\alpha}} S'$. By Theorem 4.33, we can take any model $\mathcal{M}'$ for which $\mathcal{M} \sqsubseteq_\Gamma \mathcal{M}'$. Such a model exists because $\sqsubseteq_\Gamma$ is left-total (by Lemma 4.18 and Definition 4.23). □

**Example 4.35.** The following table lists some example formulas, including all those from Example 4.2. For each formula, we indicate whether it is monotonic or antimonotonic with respect to $\alpha$ according to the calculi $\mathfrak{M}_1$ and $\mathfrak{M}_2$ and to the semantic definitions.

| | MONOTONIC | | | ANTIMONOTONIC | | |
|---|---|---|---|---|---|---|
| FORMULA | $\mathfrak{M}_1$ | $\mathfrak{M}_2$ | SEM. | $\mathfrak{M}_1$ | $\mathfrak{M}_2$ | SEM. |
| $\exists x^{\alpha}\, y.\; x \neq y$ | ✓ | ✓ | ✓ | · | · | · |
| $f\, x^{\alpha} = x \wedge f\, y \neq y$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x^{o \to \alpha} = y$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $s^{\alpha \to o} = t$ | · | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\{y^{\alpha}\} = \{z\}$ | · | ✓ | ✓ | ✓ | ✓ | ✓ |
| $(\lambda x^{\alpha}.\; x = y) = (\lambda x.\; x = z)$ | · | · | ✓ | ✓ | ✓ | ✓ |
| $(\forall x^{\alpha}.\; f\, x = x) \wedge f\, y \neq y$ | · | · | ✓ | ✓ | ✓ | ✓ |
| $\forall x^{\alpha}\, y.\; x = y$ | · | · | · | ✓ | ✓ | ✓ |
| $\exists x^{\alpha}\, y.\; x \neq y \wedge \forall z.\; z = x \vee z = y$ | · | · | · | · | · | · |

In Definition 4.19, all atomic types in $\sigma$ count as positive occurrences in $\sigma \to_{\mathsf{F}} \tau$. This raises the question of whether a fully covariant behavior, with $\mathsf{AT}^{s}(\sigma \to_{\mathsf{F}} \tau) = \mathsf{AT}^{s}(\sigma) \cup \mathsf{AT}^{s}(\tau)$, could be achieved, presumably with a different definition of $\sqsubseteq^{\sigma \to_{\mathsf{F}} \tau}$. Although such a behavior looks more regular, it would make the calculus unsound, as the following counterexample shows:

$$\forall F^{(\tilde{\alpha} \to o) \to o}\, f^{\tilde{\alpha} \to o}\, g\, h.\; f \in F \wedge g \in F \wedge f\, a \neq g\, a \longrightarrow h \in F$$

The formula is not monotonic: Regardless of the value of the free variable $a$, it is true for $|\tilde{\alpha}| = 1$, since the assumptions imply that $f \neq g$, and as there are only two functions of type $\tilde{\alpha} \to o$, $h$ can only be one of them, so it must be in $F$. This argument breaks down for larger scopes, so the formula is not monotonic. However, with a fully covariant F-arrow, we could type $F$ as $F^{(\tilde{\alpha} \to_{\mathsf{G}} o) \to_{\mathsf{F}} o}$ and the rule ALL$^{+}$ would apply, since there are no positive occurrences of $\tilde{\alpha}$ in the types of $F$, $f$, $g$, and $h$.

### 4.3.4 Type Inference

Expecting all types to be fully annotated with G and F is unrealistic, so we now face the problem of computing annotations such that a given term is typable—a type inference problem. We follow a standard approach to type inference: We start by annotating all types with *annotation variables* ranging over $\{\mathsf{G}, \mathsf{F}\}$. Then we construct a typing derivation by backward chaining, collecting a set of constraints over the annotations. Finally, we look for an instantiation of the annotation variables that satisfies all the constraints.

**Definition 4.36 (Annotation Constraint).** An *annotation constraint* over a set of annotation variables $X$ is an expression of the form $\sigma \leq \tau$, $\tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)$, or $\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)$, where the types $\sigma$ and $\tau$ may contain annotation variables in $V$. Given a valuation $\rho : X \to \{\mathsf{G}, \mathsf{F}\}$, the meaning of a constraint is given by Definitions 4.19 and 4.25.

A straightforward way of solving such constraints is to encode them in propositional logic, following Definitions 4.19 and 4.25, and give them to a SAT solver. Annotation variables, which may take two values, are mapped directly to propositional variables. Only one rule, SUB, is not syntax-directed, but it is sufficient to apply it to the second argument of an application and to variables and constants before invoking VAR or CONST. This approach proved very efficient on the problems that we encountered in our experiments.

It is unclear whether the satisfiability problem for this constraint language is NP-complete or in P. We suspect that it is in P, but we have not found a polynomial-time algorithm. Thus, it is unclear if our use of a SAT solver is fully appropriate from a theoretical point of view, even though it works perfectly well in practice.

Similarly to the simply typed $\lambda$-calculus, our type system admits principal types if we promote annotation variables to first-class citizens. When performing type inference, we would keep the constraints as part of the type, instead of computing a solution to the collected constraints. More precisely, a *type schema* would have the form $\forall\overline{A}.\ \forall\overline{\alpha}.\ \sigma\ \langle C\rangle$, where $\sigma$ is a type containing annotation variables $\overline{A}$ and type variables $\overline{\alpha}$, and $C$ is a list of constraints of the form given in Definition 4.36. Equality would have the principal type schema $\forall\alpha.\ \alpha \to_{\mathsf{G}} \alpha \to_{A} o\ \ \langle\tilde{\alpha} \notin \mathsf{AT}^{-}(\alpha)\rangle$. This approach nicely extends ML-style polymorphism.

## 4.4 Third Calculus: Handling Set Comprehensions

An obvious deficiency of the calculus $\mathfrak{M}_2$ from the previous section is that the rule LAM always types $\lambda$-abstractions with G-arrows. The only way to construct a term with F-annotations is to build it from primitives whose types are justified semantically. In other words, we cannot type set comprehensions precisely.

This solution is far from optimal. Consider the term $\lambda R\, S\, x\, z.\ \exists y.\ R\, x\, y \wedge S\, y\, z$, which composes two binary relations $R$ and $S$. Semantically, composition is constant for type $(\alpha \to_{\mathsf{F}} \beta \to_{\mathsf{F}} o) \to_{\mathsf{G}} (\beta \to_{\mathsf{F}} \gamma \to_{\mathsf{F}} o) \to_{\mathsf{G}} \alpha \to_{\mathsf{F}} \gamma \to_{\mathsf{F}} o$, but $\mathfrak{M}_2$ cannot infer this. As a result, it cannot infer the monotonicity of any of the four type variables occurring in the associativity law for composition, unless composition is considered a primitive and added to the constant context $\Gamma_{\mathsf{c}}$. Another annoyance is that the $\eta$-expansion $\lambda x.\ t\ x$ of a term $t^{\sigma \to_{\mathsf{F}} \tau}$ can only be typed with a G-arrow.

The calculus $\mathfrak{M}_3$ introduced in this section is designed to address this. The underlying intuition is that a term $\lambda x^{\alpha}.\ t$ can be typed as $\alpha \to_{\mathsf{F}} o$ if we can show that the body $t$ evaluates to $\bot$ whenever $x$ is new in the larger scope. The key is to track what happens to a term when one or several of its free variables are assigned a new value. When abstracting over a variable, we can then use this information to annotate the function arrow precisely. This scheme covers bounded set comprehensions such as $\lambda x.\ x \in A^{\tilde{\alpha}\to_{\mathsf{F}} o} \wedge x \in B^{\tilde{\alpha}\to_{\mathsf{F}} o}$ and, by way of consequence, bounded quantifications such as $\forall x.\ x \in A^{\tilde{\alpha}\to_{\mathsf{F}} o} \longrightarrow q\ x$.

**Definition 4.37 (Annotated Type).** An *annotated type* is a HOL type in which each function arrow carries an *annotation* $A \in \{\mathsf{G}, \mathsf{N}, \mathsf{F}, \mathsf{T}\}$.

The G- and F-annotations have the same meaning as in Section 4.3. The T-annotation is similar to F, but with $\top$ instead of $\bot$ as its default extension of type $o$; the universal set $\mathsf{UNIV}^{\alpha\to o}$ can be typed precisely as $\alpha \to_{\mathsf{T}} o$. Finally, the N-annotation indicates that if the argument to the function is a new value, so is its result.

### 4.4.1 Extension Relation

The extension relation $\sqsubseteq^\sigma$ distinguishes the four kinds of arrow. The G and F cases are the same as in Definition 4.17.

**Definition 4.38 (Extension).** Let $\sigma$ be an annotated type, and let $S$, $S'$ be scopes such that $S \leq_{\tilde{\alpha}} S'$. The *extension* relation $\sqsubseteq^\sigma \subseteq [\![\sigma]\!]_S \times [\![\sigma]\!]_{S'}$ for $S$ and $S'$ is defined by the equivalences

$$\begin{array}{rll} a \sqsubseteq^\sigma b & \text{iff} & a = b \qquad \text{if } \sigma \text{ is } o \text{ or an atomic type} \\ f \sqsubseteq^{\sigma\to_A\tau} g & \text{iff} & \forall a\, b.\ a \sqsubseteq^\sigma b \longrightarrow f(a) \sqsubseteq^\tau g(b) \text{ and} \\ & & \forall b.\ \mathsf{N}^\sigma(b) \longrightarrow A^\tau(g(b)) \end{array}$$

and

$$\begin{array}{llll} \mathsf{G}^\sigma(b) \text{ iff } \top & \qquad & \mathsf{F}^\sigma(b) \text{ iff } b = (\!|\sigma|\!)^{\mathsf{F}} \\ \mathsf{N}^\sigma(b) \text{ iff } \nexists a.\ a \sqsubseteq^\sigma b & \qquad & \mathsf{T}^\sigma(b) \text{ iff } b = (\!|\sigma|\!)^{\mathsf{T}} \end{array}$$

where $(\!|o|\!)^{\mathsf{F}} = \bot$, $(\!|o|\!)^{\mathsf{T}} = \top$, $(\!|\alpha|\!)^{A} \in S(\alpha)$, and $(\!|\sigma \to \tau|\!)^{A} = a \in [\![\sigma]\!]_{S'} \mapsto (\!|\tau|\!)^{A}$.

We cannot require $(\!|\alpha|\!)^{\mathsf{F}} \neq (\!|\alpha|\!)^{\mathsf{T}}$ because this would be impossible for $|\alpha| = 1$. Hence, we must be careful to assume $(\!|\sigma|\!)^{\mathsf{F}} \neq (\!|\sigma|\!)^{\mathsf{T}}$ only if $\sigma$ is of the form $\sigma_1 \to \cdots \to \sigma_n \to o$.

**Definition 4.39 (Positive and Negative Atomic Types).** The set of *positive atomic types* $\mathsf{AT}^+(\sigma)$ and the set of *negative atomic types* $\mathsf{AT}^-(\sigma)$ of an annotated type $\sigma$ are defined as follows:

$$\begin{array}{ll} \mathsf{AT}^+(o) = \emptyset & \mathsf{AT}^+(\sigma \to_A \tau) = \begin{cases} \mathsf{AT}^+(\tau) \cup \mathsf{AT}^-(\sigma) & \text{if } A = \mathsf{G} \\ \mathsf{AT}^+(\tau) \cup \mathsf{AT}^-(\sigma) \cup \mathsf{AT}^+(\sigma) & \text{otherwise} \end{cases} \\ \mathsf{AT}^+(\alpha) = \{\alpha\} & \\ \mathsf{AT}^-(o) = \emptyset & \mathsf{AT}^-(\sigma \to_A \tau) = \begin{cases} \mathsf{AT}^-(\tau) \cup \mathsf{AT}^+(\sigma) & \text{if } A \in \{\mathsf{G}, \mathsf{N}\} \\ \mathsf{AT}^-(\tau) & \text{otherwise} \end{cases} \\ \mathsf{AT}^-(\alpha) = \emptyset & \end{array}$$

With the introduction of an N-annotation, not all annotated types are legitimate. For example, $\tilde{\alpha} \to_{\mathsf{N}} o$ would mean that new values of type $\tilde{\alpha}$ are mapped to new Booleans, but there is no such thing as a new Boolean, since $[\![o]\!]$ is always $\{\bot, \top\}$.

**Definition 4.40 (Well-Annotated Type).** An annotated type $\sigma$ is *well-annotated* if $\mathsf{WA}(\sigma)$ holds, where $\mathsf{WA}(\sigma)$ is defined by the following equivalences:

$$\begin{array}{rll} \mathsf{WA}(\sigma) & \text{iff} & \top \qquad\qquad\qquad \text{if } \sigma \text{ is } o \text{ or an atomic type} \\ \mathsf{WA}(\sigma \rightarrow_A \tau) & \text{iff} & \mathsf{WA}(\sigma) \text{ and } \mathsf{WA}(\tau) \text{ and } A = \mathsf{N} \longrightarrow \mathit{body}(\tau) = \tilde{\alpha} \end{array}$$

where $\mathit{body}(o) = o$, $\mathit{body}(\alpha) = \alpha$, and $\mathit{body}(\sigma \rightarrow \tau) = \mathit{body}(\tau)$.

**Lemma 4.41.** *If $\sigma$ is well-annotated, then $\sqsubseteq^{\sigma}$ is left-total (i.e., total) and left-unique (i.e., injective).*

*Proof* (structural induction on $\sigma$). The proof is similar to that of Lemma 4.18, except that we must propagate the WA assumption. There is one genuinely new case.

LEFT-TOTALITY OF $\sqsubseteq^{\sigma \rightarrow_{\mathsf{N}} \tau}$: For $f \in [\![\sigma \rightarrow \tau]\!]_S$, we find an extension $g$ as follows: Let $b \in [\![\sigma]\!]_{S'}$. If $b$ extends an $a$, that $a$ is unique by left-uniqueness of $\sqsubseteq^{\sigma}$. Since $\sqsubseteq^{\tau}$ is left-total, there exists a $y$ such that $f(a) \sqsubseteq^{\tau} y$, and we let $g(b) = y$. If $b$ does not extend any $a$, then we set $g(b)$ to a new element $y$ constructed as follows: Since $\sigma \rightarrow_{\mathsf{N}} \tau$ is well-annotated, $\tau$ must be of the form $\tau_1 \rightarrow_{A_1} \cdots \rightarrow_{A_{n-1}} \tau_n \rightarrow_{A_n} \tilde{\alpha}$. As value for $y$, we simply take $y_1 \in [\![\tau_1]\!]_{S'} \mapsto \cdots \mapsto y_n \in [\![\tau_n]\!]_{S'} \mapsto y'$, where $y'$ is not the extension of any $x'$. Such a $y'$ exists, because otherwise $|S(\tilde{\alpha})| = |S'(\tilde{\alpha})|$, which is inconsistent with the existence of a new element $b$. □

**Lemma 4.42.** *Let $\sigma$ be a well-annotated type. If $\tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)$, then $\sqsubseteq^{\sigma}$ is right-total (i.e., surjective). If $\tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)$, then $\sqsubseteq^{\sigma}$ is right-unique (i.e., functional).*

*Proof.* The proof is similar to that of Lemma 4.20. Lemma 4.41, which replaces Lemma 4.18, requires $\sigma$ to be well-annotated. □

### 4.4.2 Type Checking

As in Section 4.3, constancy checking is treated as a type checking problem involving annotated types. The judgments still have the form $\Gamma \vdash t : \sigma$, but the context now carries more information about $t$'s value when its free variables are assigned new values.

**Definition 4.43 (Context).** A *context* is a pair $\Gamma = (\Gamma_{\mathrm{c}}, \Gamma_{\mathrm{v}})$, where $\Gamma_{\mathrm{c}}$ maps constant symbols to sets of annotated types, and $\Gamma_{\mathrm{v}}$ is a list $[x_1 :^{A_1} \sigma_1, \ldots, x_m :^{A_m} \sigma_m]$ associating $m$ distinct variables $x_i$ with annotations $A_i$ and annotated types $\sigma_i$. A context is *well-annotated* if all the types $\sigma_i$ are well-annotated.

We assume as before that $\Gamma_{\mathrm{c}}$ is fixed and compatible with the standard constant model and write $\Gamma$ for $\Gamma_{\mathrm{v}}$. We abbreviate $[x_1 :^{A_1} \sigma_1, \ldots, x_m :^{A_m} \sigma_m]$ to $\langle x_m :^{A_m} \sigma_m \rangle$. The intuitive meaning of a typing judgment $\Gamma \vdash t : \sigma$ with $\Gamma = \langle x_m :^{A_m} \sigma_m \rangle$ is that if $x_1$ is new, then $A_1^{\sigma_1}(t)$ holds; if $V(x_1) \sqsubseteq^{\sigma_1} V'(x_1)$ but $x_2$ is new, then $A_2^{\sigma_2}(t)$ holds; and so on. Furthermore, if $V(x_i) \sqsubseteq^{\sigma_i} V'(x_i)$ for all $i \in \{1, \ldots, m\}$, then $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![t]\!]_{\mathcal{M}'}$. It may help to think of a judgment $\langle x_m :^{A_m} \sigma_m \rangle \vdash t : \sigma$ as meaning roughly the same as $[] \vdash \lambda x_1 \ldots x_m.\, t : \sigma_1 \rightarrow_{A_1} \cdots \rightarrow_{A_{n-1}} \sigma_n \rightarrow_{A_n} \sigma$.[1]

[1] In fact, by Lemma 4.50 (and Definitions 4.46 and 4.47), they are *exactly* the same if none of the annotations $A_i$ are $\mathsf{N}$.

**Example 4.44.** Given a constant $r$ such that $\tilde\alpha \rightarrow_{\mathsf{T}} \tilde\alpha \rightarrow_{\mathsf{F}} o \in \Gamma_{\mathsf{c}}(\mathsf{r})$, the new typing rules will let us derive the following judgments:

$$\begin{aligned}
[x :^{\mathsf{T}} \tilde\alpha,\, y :^{\mathsf{F}} \tilde\alpha] &\vdash r\; x\; y : o \\
[x :^{\mathsf{T}} \tilde\alpha] &\vdash \lambda y.\; r\; x\; y : \tilde\alpha \rightarrow_{\mathsf{F}} o \\
[] &\vdash \lambda x\, y.\; r\; x\; y : \tilde\alpha \rightarrow_{\mathsf{T}} \tilde\alpha \rightarrow_{\mathsf{F}} o
\end{aligned}$$

Notice that the $\eta$-expanded form $\lambda x\, y.\; r\; x\; y$ can be typed in the same way as $r$.

The following definitions make this more rigorous.

**Definition 4.45 (Model Extension).** Let $\mathcal{M} = (S, V)$ and $\mathcal{M}' = (S', V')$ be models, and let $\Gamma$, $\Delta$ be two contexts with disjoint sets of variables. The model $\mathcal{M}'$ *extends* $\mathcal{M}$ strongly in $\Gamma$ and weakly in $\Delta$, written $\mathcal{M} \sqsubseteq_{\Gamma}^{\Delta} \mathcal{M}'$, if $S \leq_{\tilde\alpha} S'$, $x :^{A} \sigma \in \Gamma$ implies $V(x) \sqsubseteq^{\sigma} V'(x)$, and $x :^{A} \sigma \in \Delta$ implies either $V(x) \sqsubseteq^{\sigma} V'(x)$ or $\mathsf{N}^{\sigma}(V'(x))$ for all $x$. If $\Delta = []$, we write $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$.

**Definition 4.46 (Constancy).** Let $\sigma$ be an annotated type. A term $t$ is *$\sigma$-constant* in a context $\Gamma$ if $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![t]\!]_{\mathcal{M}'}$ for all models $\mathcal{M}$, $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$.

**Definition 4.47 (Conformity).** Let $\sigma$ be an annotated type, and let $\Gamma$ be a context. A term $t$ is *$\sigma$-conformant* to $\Gamma$ if for all decompositions $\Gamma = \Delta, [x :^{A} \tau], \mathrm{E}$ and for all models $\mathcal{M}$, $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq_{\Delta}^{\mathrm{E}} \mathcal{M}'$, we have that $\mathsf{N}^{\tau}([\![x]\!]_{\mathcal{M}'})$ implies $A^{\sigma}([\![t]\!]_{\mathcal{M}'})$.

Equipped with these semantic definitions, we are ready to examine the inference rules of $\mathfrak{M}_3$ relating to constancy and monotonicity.

**Definition 4.48 (Typing Rules).** The *typing* relation $\Gamma \vdash t : \sigma$ is specified by the context, nonlogical, and logical rules below.

*Context rules:*

$$\frac{\Gamma, \Delta \vdash t : \tau}{\Gamma, [x :^{\mathsf{G}} \sigma], \Delta \vdash t : \tau}\;\textsc{Add} \qquad \frac{\Gamma, [x :^{A} \sigma], \Delta \vdash t : \tau}{\Gamma, [x :^{\mathsf{G}} \sigma], \Delta \vdash t : \tau}\;\textsc{Ann}$$

$$\frac{\Gamma, [y :^{B} \tau,\; x :^{A} \sigma], \Delta \vdash t : \upsilon}{\Gamma, [x :^{A} \sigma,\; y :^{B} \tau], \Delta \vdash t : \upsilon}\;\textsc{Swap} \quad \text{where } A \in \{B, \mathsf{G}\}$$

*Nonlogical rules:*

$$\frac{}{[x :^{\mathsf{N}} \sigma] \vdash x : \sigma}\;\textsc{Var} \qquad \frac{\sigma \in \Gamma_{\mathsf{c}}(\mathsf{c})}{[] \vdash \mathsf{c} : \sigma}\;\textsc{Const}$$

$$\frac{\Gamma \vdash t : \sigma \quad \sigma \leq \sigma'}{\Gamma \vdash t : \sigma'}\;\textsc{Sub} \qquad \frac{\Gamma, [x :^{A} \sigma] \vdash t : \tau}{\Gamma \vdash \lambda x.\; t : \sigma \rightarrow_{A} \tau}\;\textsc{Lam}$$

$$\frac{\langle x_m :^{A_m} \sigma_m \rangle \vdash t : \sigma \rightarrow_{B} \tau \quad \langle x_m :^{\mathsf{G}} \sigma_m \rangle, \langle y_n :^{\mathsf{N}} \tau_n \rangle \vdash u : \sigma}{\langle x_m :^{A_m} \sigma_m \rangle, \langle y_n :^{B} \tau_n \rangle \vdash t\; u : \tau}\;\textsc{App} \quad \text{where } A_i \in \{\mathsf{G}, \mathsf{F}, \mathsf{T}\}$$

*Logical rules:*

$$\frac{}{\langle x_m :^{\mathsf{F}} \sigma_m \rangle \vdash \mathsf{False} : o}\ \textsc{False} \qquad \frac{}{\langle x_m :^{\mathsf{T}} \sigma_m \rangle \vdash \mathsf{True} : o}\ \textsc{True}$$

$$\frac{\langle x_m :^{A_m} \sigma_m \rangle \vdash t : o \quad \langle x_m :^{B_m} \sigma_m \rangle \vdash u : o}{\langle x_m :^{A_m \rightsquigarrow B_m} \sigma_m \rangle \vdash t \longrightarrow u : o}\ \textsc{Imp}$$

$$\text{where } A \rightsquigarrow B = \begin{cases} \mathsf{T} & \text{if } A = \mathsf{F} \text{ or } B = \mathsf{T} \\ \mathsf{F} & \text{if } A = \mathsf{T} \text{ and } B = \mathsf{F} \\ \mathsf{G} & \text{otherwise} \end{cases}$$

The *subtype* relation $\sigma \leq \tau$ is defined by the rules

$$\frac{}{\sigma \leq \sigma}\ \textsc{Refl} \qquad \frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \rightarrow_{A} \tau \leq \sigma' \rightarrow_{\mathsf{G}} \tau'}\ \textsc{Gen} \qquad \frac{\sigma' \leq \sigma \quad \sigma \leq \sigma' \quad \tau \leq \tau'}{\sigma \rightarrow_{A} \tau \leq \sigma' \rightarrow_{A} \tau'}\ \textsc{Any}$$

The nonlogical rules are similar to the rules of Definition 4.25, but the LAM rule now allows arbitrary annotations, and the other rules impose various restrictions on the annotations in the contexts. The logical rules support rudimentary propositional reasoning within terms.

In the previous calculus, the context was a set and we could dispense with explicit context rules. The context now being a list, we need weakening rules to add and permute variables and to change the annotations in a controlled way. The new typing rules form a substructural type system [195].

**Lemma 4.49.** *If $\sigma \leq \sigma'$, then $\sqsubseteq^{\sigma} \subseteq \sqsubseteq^{\sigma'}$.*

*Proof.* Similar to the proof of Lemma 4.26. □

The proof of the soundness theorem relies on two closure properties of functional abstraction and application.

**Lemma 4.50.** *Let $g \in [\![\sigma \rightarrow \tau]\!]_{S'}$.*

- **(a)** *If $A^{\tau}(g(b))$ for all $b \in [\![\sigma]\!]_{S'}$, then $A^{\sigma \rightarrow_B \tau}(g)$.*
- **(b)** *If $A \in \{\mathsf{G}, \mathsf{F}, \mathsf{T}\}$ and $A^{\sigma \rightarrow_B \tau}(g)$, then $A^{\tau}(g(b))$ for all $b \in [\![\sigma]\!]_{S'}$.*

*Proof.* Immediate from Definition 4.38. □

It is regrettable that Lemma 4.50(b) does not hold uniformly for all annotation types and, as a result, that the APP rule has a side condition $A_i \in \{\mathsf{G}, \mathsf{F}, \mathsf{T}\}$. The crux of the matter is that while a function that maps old values to new values is necessarily new, the converse does not hold: A function may be new even if it maps old values to old values. Given the type $\tilde{\alpha} \rightarrow_{\mathsf{F}} o$, the function $(\bot, \bot, \top)$ depicted in Figure 4.2(b) is an example of this.

**Theorem 4.51 (Soundness of Typing).** *If $\Gamma \vdash t : \sigma$, then $t$ is both $\sigma$-constant in $\Gamma$ and $\sigma$-conformant to $\Gamma$.*

*Proof* (induction on the derivation of $\Gamma \vdash t : \sigma$). The cases ADD, ANN, VAR, CONST, FALSE, TRUE, and IMP are easy to prove using the definitions of $\sqsubseteq^{\sigma}$, constancy, and conformity. The remaining cases are proved below.

SWAP: The case $A = \mathsf{G}$ is easy. In the remaining case, the only subtlety occurs when both $x$ and $y$ are new, i.e., $\mathsf{N}^{\sigma}(\llbracket x\rrbracket_{\mathcal{M}'})$ and $\mathsf{N}^{\tau}(\llbracket y\rrbracket_{\mathcal{M}'})$; but since $A = B$, the behaviors dictated by $x :^{A} \sigma$ and $y :^{B} \tau$ agree and we can exchange them.

SUB: By Lemma 4.49 and Definition 4.46.

LAM: The $(\sigma \to_{A} \tau)$-conformity of $\lambda x.\ t$ follows from Lemma 4.50(a) and the induction hypothesis; constancy is easy to prove from the induction hypothesis and the definition of $\sqsubseteq^{\sigma \to_A \tau}$. We can omit $x :^{A} \sigma$ in the conclusion because $x$ does not occur free in $\lambda x.\ t$.

APP: Let $\Gamma = \langle x_m :^{A_m} \sigma_m\rangle, \langle y_n :^{B} \tau_n\rangle$. The constancy proof is as for Theorem 4.27. It remains to show that $t\ u$ is $\tau$-conformant to $\Gamma$. Let $\Gamma = \Delta, [z :^{C} \upsilon], \mathrm{E}$ and assume $\mathsf{N}^{\upsilon}(\llbracket z\rrbracket_{\mathcal{M}'})$. If $z$ is one of the $x_i$'s, we have $C^{\sigma \to_B \tau}(\llbracket t\rrbracket_{\mathcal{M}'})$ by the first induction hypothesis and hence $C^{\tau}(\llbracket t\ u\rrbracket_{\mathcal{M}'})$ by Lemma 4.50(b). If $z$ is among the $y_j$'s (in which case $C = B$), the second induction hypothesis tells us that $\llbracket u\rrbracket_{\mathcal{M}'}$ is new, and since $\llbracket t\rrbracket_{\mathcal{M}} \sqsubseteq^{\sigma \to_B \tau} \llbracket t\rrbracket_{\mathcal{M}'}$ by the first induction hypothesis, we have $B^{\tau}(\llbracket t\ u\rrbracket_{\mathcal{M}'})$ by the definition of $\sqsubseteq^{\sigma \to_B \tau}$. □

**Definition 4.52 (Derived Typing Rules).** From Definitions 2.6 and 4.48, we derive the following rules for logical constants:

$$\frac{\langle x_m :^{A_m} \sigma_m\rangle \vdash t : o}{\langle x_m :^{\sim A_m} \sigma_m\rangle \vdash \neg\, t : o}\ \textsc{Not} \quad \text{where } {\sim}A = \begin{cases} \mathsf{T} & \text{if } A = \mathsf{F} \\ \mathsf{F} & \text{if } A = \mathsf{T} \\ \mathsf{G} & \text{otherwise} \end{cases}$$

$$\frac{\langle x_m :^{A_m} \sigma_m\rangle \vdash t : o \quad \langle x_m :^{B_m} \sigma_m\rangle \vdash u : o}{\langle x_m :^{A_m \frown B_m} \sigma_m\rangle \vdash t \wedge u : o}\ \textsc{And} \quad \text{where } A \frown B = \begin{cases} \mathsf{T} & \text{if } A = B = \mathsf{T} \\ \mathsf{F} & \text{if } A = \mathsf{F} \text{ or } B = \mathsf{F} \\ \mathsf{G} & \text{otherwise} \end{cases}$$

$$\frac{\langle x_m :^{A_m} \sigma_m\rangle \vdash t : o \quad \langle x_m :^{B_m} \sigma_m\rangle \vdash u : o}{\langle x_m :^{A_m \smile B_m} \sigma_m\rangle \vdash t \vee u : o}\ \textsc{Or} \quad \text{where } A \smile B = \begin{cases} \mathsf{T} & \text{if } A = \mathsf{T} \text{ or } B = \mathsf{T} \\ \mathsf{F} & \text{if } A = B = \mathsf{F} \\ \mathsf{G} & \text{otherwise} \end{cases}$$

Equipped with powerful typing rules, we no longer need to reason semantically about set constructs. This leaves us with a much reduced standard constant context.

**Definition 4.53 (Standard Constant Context).** The *standard constant context* $\widehat{\Gamma}_{\mathrm{c}}$ is the following mapping:

$$\begin{array}{rcl} \longrightarrow & \mapsto & \{o \to_{\mathsf{G}} o \to_{\mathsf{G}} o\} \\ = & \mapsto & \{\sigma \to_{\mathsf{G}} \sigma \to_{\mathsf{F}} o \mid \tilde{\alpha} \notin \mathsf{AT}^{-}(\sigma)\} \end{array}$$

**Lemma 4.54.** *The standard constant context $\widehat{\Gamma}_{\mathrm{c}}$ is compatible with the standard constant model $\widehat{M}$.*

*Proof.* Analogous to the proof of Lemma 4.29. □

The examples below exploit the new calculus to type set constructs precisely.

**Example 4.55.** The empty set $\emptyset^{\sigma \to o}$ and the universal set $\mathsf{UNIV}^{\sigma \to o}$ get their natural typings:

$$\dfrac{\dfrac{}{[x :^{\mathsf{F}} \sigma] \vdash \mathsf{False} : o}\ \textsc{False}}{[] \vdash \lambda x.\ \mathsf{False} : \sigma \rightarrow_{\mathsf{F}} o}\ \textsc{Lam} \qquad \dfrac{\dfrac{}{[x :^{\mathsf{T}} \sigma] \vdash \mathsf{True} : o}\ \textsc{True}}{[] \vdash \lambda x.\ \mathsf{True} : \sigma \rightarrow_{\mathsf{T}} o}\ \textsc{Lam}$$

**Example 4.56.** The complement $\bar{s}$ of a set $s$ is the set $\mathsf{UNIV} - s$. It can be typed as follows for $A \in \{\mathsf{G}, \mathsf{F}, \mathsf{T}\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[s :^{\mathsf{N}} \sigma \rightarrow_{A} o] \vdash s : \sigma \rightarrow_{A} o}\ \textsc{Var}}{[s :^{\mathsf{G}} \sigma \rightarrow_{A} o] \vdash s : \sigma \rightarrow_{A} o}\ \textsc{Ann} \quad \dfrac{\dfrac{}{[x :^{\mathsf{N}} \sigma] \vdash x : \sigma}\ \textsc{Var}}{[s :^{\mathsf{G}} \sigma \rightarrow_{A} o,\ x :^{\mathsf{N}} \sigma] \vdash x : \sigma}\ \textsc{Add}}{[s :^{\mathsf{G}} \sigma \rightarrow_{A} o,\ x :^{A} \sigma] \vdash s\ x : o}\ \textsc{App}}{[s :^{\mathsf{G}} \sigma \rightarrow_{A} o,\ x :^{\sim A} \sigma] \vdash \neg\, s\ x : o}\ \textsc{Not}}{[s :^{\mathsf{G}} \sigma \rightarrow_{A} o] \vdash \lambda x.\ \neg\, s\ x : \sigma \rightarrow_{\sim A} o}\ \textsc{Lam}}{[] \vdash \lambda s\, x.\ \neg\, s\ x : (\sigma \rightarrow_{A} o) \rightarrow_{\mathsf{G}} \sigma \rightarrow_{\sim A} o}\ \textsc{Lam}$$

### 4.4.3 Monotonicity Checking

The rules for checking monotonicity and antimonotonicity are similar to those given in Section 4.3. The only new rule is $\textsc{All}^{+}_{\mathsf{T}}$; it exploits the context to avoid the restriction on $\tilde{\alpha}$.

**Definition 4.57 (Monotonicity Rules).** The predicates $\Gamma \vdash \mathsf{M}^{+}(t)$ and $\Gamma \vdash \mathsf{M}^{-}(t)$ are given by the rules

$$\dfrac{\Gamma \vdash t : o}{\Gamma \vdash \mathsf{M}^{s}(t)}\ \textsc{Term} \qquad \dfrac{\langle x_m :^{A_m} \sigma_m \rangle \vdash \mathsf{M}^{\sim s}(t) \quad \langle x_m :^{B_m} \sigma_m \rangle \vdash \mathsf{M}^{s}(u)}{\langle x_m :^{A_m \rightsquigarrow B_m} \sigma_m \rangle \vdash \mathsf{M}^{s}(t \longrightarrow u)}\ \textsc{Imp}$$

$$\dfrac{\langle x_m :^{\mathsf{G}} \sigma_m \rangle \vdash t : \sigma \quad \langle x_m :^{\mathsf{G}} \sigma_m \rangle \vdash u : \sigma}{\langle x_m :^{\mathsf{G}} \sigma_m \rangle \vdash \mathsf{M}^{-}(t = u)}\ \textsc{Eq}^{-} \qquad \dfrac{\Gamma, [x :^{\mathsf{G}} \sigma] \vdash \mathsf{M}^{-}(t)}{\Gamma \vdash \mathsf{M}^{-}(\forall x.\ t)}\ \textsc{All}^{-}$$

$$\dfrac{\Gamma, [x :^{\mathsf{G}} \sigma] \vdash \mathsf{M}^{+}(t) \quad \tilde{\alpha} \notin \mathsf{AT}^{+}(\sigma)}{\Gamma \vdash \mathsf{M}^{+}(\forall x.\ t)}\ \textsc{All}^{+} \qquad \dfrac{\Gamma, [x :^{\mathsf{T}} \sigma] \vdash \mathsf{M}^{+}(t)}{\Gamma \vdash \mathsf{M}^{+}(\forall x.\ t)}\ \textsc{All}^{+}_{\mathsf{T}}$$

From Definition 4.57, it would be straightforward to derive monotonicity rules for $\mathsf{False}$, $\mathsf{True}$, $\neg$, $\wedge$, $\vee$, and $\exists$.

**Theorem 4.58 (Soundness of $\mathbf{M}^{s}$).** *Let $\mathcal{M}$ and $\mathcal{M}'$ be models such that $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$. If $\Gamma \vdash \mathsf{M}^{+}(t)$, then $t$ is monotonic and $o$-conformant to $\Gamma$. If $\Gamma \vdash \mathsf{M}^{-}(t)$, then $t$ is antimonotonic and $o$-conformant to $\Gamma$.*

*Proof* (induction on the derivation of $\Gamma \vdash \mathsf{M}^{s}(t)$). The $\textsc{Term}$, $\textsc{Eq}^{-}$, $\textsc{All}^{-}$, and $\textsc{All}^{+}$ cases are similar to the corresponding cases in the proof of Theorem 4.33. The $\textsc{Imp}$ case is analogous to the $\textsc{Imp}$ case of Theorem 4.51. Finally, the rule $\textsc{All}^{+}_{\mathsf{T}}$ can be derived by treating $\forall x.\ t$ as an abbreviation for $(\lambda x.\ t) = (\lambda x.\ \mathsf{True})$. □

**Theorem 4.59 (Soundness of the Calculus).** *If $\Gamma \vdash \mathsf{M}^{+}(t)$ can be derived in some arbitrary well-annotated context $\Gamma$, then $t$ is monotonic. If $\Gamma \vdash \mathsf{M}^{-}(t)$ can be derived in some arbitrary well-annotated context $\Gamma$, then $t$ is antimonotonic.*

*Proof.* By Theorem 4.58, we can take any model $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq_\Gamma \mathcal{M}'$ as witness for monotonicity. Such a model exists because $\sqsubseteq_\Gamma$ is left-total for well-annotated contexts $\Gamma$ (by Lemma 4.18 and Definition 4.23). □

**Example 4.60.** The bounded quantification $\forall x.\ x \in A^{\tilde\alpha \to o} \longrightarrow q\ x$ can be inferred monotonic in the context $\Gamma = [A :^{\mathsf{G}} \tilde\alpha \rightarrow_{\mathsf{F}} o]$ if $q\ x$ can be inferred monotonic:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[A :^{\mathsf{N}} \tilde\alpha \rightarrow_{\mathsf{F}} o] \vdash A : \tilde\alpha \rightarrow_{\mathsf{F}} o}\,\textsc{Var}}{\Gamma \vdash A : \tilde\alpha \rightarrow_{\mathsf{F}} o}\,\textsc{Ann} \qquad \dfrac{}{\Gamma[x :^{\mathsf{N}} \tilde\alpha] \vdash x : \tilde\alpha}\,\textsc{Var}}{\Gamma[x :^{\mathsf{F}} \tilde\alpha] \vdash x \in A : o}\,\textsc{App}}{\Gamma[x :^{\mathsf{F}} \tilde\alpha] \vdash \mathsf{M}^{-}(x \in A)}\,\textsc{Term} \qquad \dfrac{\dfrac{\vdots}{[x :^{\mathsf{G}} \tilde\alpha] \vdash \mathsf{M}^{+}(q\ x)}}{\Gamma[x :^{\mathsf{G}} \tilde\alpha] \vdash \mathsf{M}^{+}(q\ x)}\,\textsc{Add}}{\dfrac{\Gamma[x :^{\mathsf{T}} \tilde\alpha] \vdash \mathsf{M}^{+}(x \in A \longrightarrow q\ x)}{\Gamma \vdash \mathsf{M}^{+}(\forall x.\ x \in A \longrightarrow q\ x)}\,\textsc{All}^{+}_{\mathsf{T}}}\,\textsc{Imp}$$

### 4.4.4 Type Inference

At the cost of some inelegant technicalities, the approach sketched in Section 4.3.4 for inferring types can be adapted to the new setting.

The nonlogical rules VAR, CONST, and LAM as well as all the logical rules are syntax-directed and pose no problem when constructing a typing derivation by backward chaining. The SUB rule is unproblematic for the same reasons as for $\mathfrak{M}_2$. Similarly, the context rules ADD and ANN can be deferred to the leaves of the derivation. The SWAP rule is useful only in conjunction with APP, the only other rule that examines the order of the context variables; other occurrences of SWAP can be either postponed or omitted altogether.

The challenge is to handle APP and SWAP, because it is not obvious where to split the context in two parts and whether variables should be exchanged first. Since the context is finite, we could enumerate all permutations and splittings, but this might lead to an exponential explosion in the number of constraints.

Fortunately, there is a general approach to determine which variables to permute and where to split the context, based on the rule

$$\dfrac{\langle x_m :^{A_m} \sigma_m\rangle \vdash t : \sigma \rightarrow_B \tau \qquad \langle x_m :^{\mathsf{G}} \sigma_m\rangle, \langle y_n :^{\mathsf{G}} \tau_n\rangle, \langle z_p :^{\mathsf{N}} \upsilon_p\rangle \vdash u : \sigma}{\langle x_m :^{A_m} \sigma_m\rangle, \langle y_n :^{\mathsf{G}} \tau_n\rangle, \langle z_p :^{B} \upsilon_p\rangle \vdash t\ u : \tau}\;3\textsc{App}$$

where $A_i \in \{\mathsf{G}, \mathsf{F}, \mathsf{T}\}$, $x_i \in \mathsf{FV}(t)$, and $y_j$, $z_k \notin \mathsf{FV}(t)$. This rule is easy to derive from APP and ADD. Before applying it, we invoke SWAP repeatedly to separate the variables occurring free in $t$ (the $x_i$'s) from the others (the $y_j$'s and $z_k$'s).

The remaining hurdle is to determine where to split between the $y_j$'s and the $z_k$'s. This can be coded as polynomial-size constraints. If $B = \mathsf{G}$, we can ignore the $z_k$'s and list all variables $\notin \mathsf{FV}(t)$ as part of the $y_j$'s; otherwise, the right-hand part of the context must have the form $\langle y_n :^{\mathsf{G}} \tau_n\rangle, \langle z_p :^{B} \upsilon_p\rangle$ already—the SWAP rule cannot bring it into that form if it is not already in that form. So we keep the variables $\notin \mathsf{FV}(t)$ in the order in which they appear without distinguishing statically between the $y_j$ and $z_k$ variables, and generate constraints to ensure that the annotations for the $y_j$'s and $z_k$'s in the conclusion have the desired form $\mathsf{G}, \ldots, \mathsf{G}, B, \ldots, B$ and correspondingly in the second hypothesis but with $\mathsf{N}$ instead of $B$.

The completeness proof rests on two ideas:

- Any APP instance in which some of the variables $x_i$ do not occur free in $t$ can only carry a G-annotation and will eventually be eliminated using ADD.[1]
- The variable exchanges we perform to bring the $x_i$'s to the left are necessary to meet the general invariant $\mathsf{FV}(t) \subseteq \{x_1, \ldots, x_m\}$ on all derivations of a judgment $\langle x_m :^{A_m} \sigma_m \rangle \vdash t : \sigma$, and any additional SWAPs can be postponed.

Type inference is in NP because our procedure generates a polynomial-size SAT problem. It is also NP-hard because we can reduce SAT to it using the following simple scheme: Translate propositional variables to HOL variables of type $\tilde{\alpha} \rightarrow o$ and connectives to the corresponding set operations; for example, $(A \wedge \neg B) \vee B$ becomes $(A \cap \overline{B}) \cup B$. The propositional formula is satisfiable iff the corresponding term is typable as $\tilde{\alpha} \rightarrow_{\mathsf{T}} o$ in some context.

## 4.5 Practical Considerations

To make monotonicity checking useful in practice, we must add support for user-defined constants and types, which we have not yet considered. In this section, we briefly sketch these extensions before we evaluate our approach empirically on existing Isabelle theories.

### 4.5.1 Constant Definitions

In principle, user-defined constant symbols are easy to handle: We can simply build a conjunction from the definitions and the negated conjecture (where the user-defined symbols appear as variables) and hand it to the monotonicity checker. Unfortunately, this has disastrous effects on the precision of our method: Even a simple definition of $\mathsf{f}^{\alpha \rightarrow \beta}$ leads to a formula of the form $(\forall x^{\alpha}.\ \mathsf{f}\ x = t) \wedge \neg\, u$, which does not pass the monotonicity check because the universal quantifier requires $\alpha \notin \mathsf{AT}^{+}(\alpha)$. Rewriting the definition to the form $\mathsf{f} = (\lambda x.\ t)$ does not help. We must thus treat definitions specially.

**Definition 4.61.** Let $\mathsf{c}^{\sigma}$ be a constant. A formula $t$ is a *specification* for $\mathsf{c}$ if $t$ is satisfiable in each scope and $\mathsf{FV}(t) = \emptyset$. Then the (nonempty) set of values in $[\![\sigma]\!]_S$ that satisfy $t$ is denoted by $Spec_S^t$.

In our application, specifications arise from Isabelle/HOL; we can assume that they are explicitly marked as such and do not have to recognize them syntactically. Specifications are trivially monotonic because they are satisfiable in each scope, and we can omit the monotonicity check for them. However, we must assign an annotated type to the defined constant, which we can use for checking the rest of the formula.

[1] In pathological cases such as the application $(\lambda y.\ \mathsf{False})\ x$, the variables not occurring in $t$ can be mapped to F or T and eliminated using FALSE or TRUE. We can avoid these cases by simply requiring that terms are $\beta$-reduced.

**Definition 4.62.** Given an annotated type $\sigma$, a specification $t$ for $\mathsf{c}$ *respects* $\sigma$ if $S \leq_{\bar{\alpha}} S'$ implies that for each $a \in \mathit{Spec}_S^t$ there exists $b \in \mathit{Spec}_{S'}^t$ such that $a \sqsubseteq^\sigma b$.

If a specification respects $\sigma$, we may assume that the defined value is $\sigma$-constant and augment our context with $[\mathsf{c} :^{\mathsf{G}} \sigma]$ while checking the rest of the formula.

We can check this property for specification formats occurring in practice. For brevity, we consider only recursive function definitions of the form $\forall x.\ \mathsf{f}^{\sigma \to \tau}\ x = F\, \mathsf{f}\, x$ where the recursion is known to be terminating (as is required by Isabelle's function package [106]). The termination argument yields a well-founded relation $R^{\sigma \to \sigma \to o}$ such that

$$\models \forall f\, g\, x.\ (\forall y.\ R\, y\, x \longrightarrow f\, y = g\, y) \longrightarrow F\, f\, x = F\, g\, x \qquad (*)$$

It then suffices to type-check the functional $F$, circumventing the quantifiers and equality in the definition of f.

**Lemma 4.63.** *Let $\sigma$, $\tau$ be annotated types and $t = (\forall x.\ \mathsf{f}^{\sigma \to \tau}\ x = F\, \mathsf{f}\, x)$ a specification for* f *such that the property* $(*)$ *holds for some well-founded relation $R$. If $[] \vdash F : (\sigma \to_{\mathsf{G}} \tau) \to_{\mathsf{G}} \tau$, then $t$ respects $\sigma \to_{\mathsf{G}} \tau$.*

*Proof.* Let $S, S'$ be scopes such that $S \leq_{\bar{\alpha}} S$ and $\mathcal{M}$, $\mathcal{M}'$ be models for $S$ and $S'$ that both satisfy $t$. Let $\hat{f} \in \mathit{Spec}_S^t$ and $\hat{g} \in \mathit{Spec}_{S'}^t$. We show $\hat{f} \sqsubseteq^{\sigma \to_{\mathsf{G}} \tau} \hat{g}$.

Let $\prec\ = [\![R]\!]_{\mathcal{M}'}$ and $F_{\mathcal{M}}(\mathsf{f}, x) = [\![F]\!]_{\mathcal{M}}(\mathsf{f})(x)$. By well-founded induction on $b \in [\![\sigma]\!]_{S'}$ using the relation $\prec$, we show that $\forall a.\ a \sqsubseteq^\sigma b \longrightarrow \hat{f}(a) \sqsubseteq^\tau \hat{g}(b)$. As induction hypothesis we have $\forall b' \prec b.\ \forall a'.\ a' \sqsubseteq^\sigma b' \longrightarrow \hat{f}(a') \sqsubseteq \hat{g}(b')$. We pick an arbitrary extension $g \sqsupseteq^{\sigma \to_{\mathsf{G}} \tau} \hat{f}$, and define the modified function $g'$ as follows:

$$g'(x) = \begin{cases} \hat{g}(x) & \text{if } x \prec b \\ g(x) & \text{otherwise} \end{cases}$$

From the induction hypothesis, we know that $\hat{f} \sqsubseteq^{\sigma \to_{\mathsf{G}} \tau} g'$ and can thus use the typing for $F$ (and Theorem 4.51) to conclude $F_S(\hat{f}, a) \sqsubseteq^\tau F_{S'}(g', b)$. Moreover, the condition $(*)$ implies that $F_{S'}(g', b) = F_{S'}(\hat{g}, b)$, since $g'$ and $\hat{g}$ behave the same on $\prec$-smaller elements. Thus, unfolding the fixed-point equation (which holds in $\mathcal{M}$ and $\mathcal{M}'$), we finally get

$$\hat{f}(a) = F_S(\hat{f}, a) \sqsubseteq^\tau F_{S'}(g', b) = F_{S'}(\hat{g}, b) = \hat{g}(b) \quad \square$$

### 4.5.2 Inductive Datatypes

The most important way of introducing new types in Isabelle/HOL is to declare an inductive datatype using the command

$$\textbf{datatype}\ \bar{\alpha}\ \kappa\ =\ \mathsf{C}_1\ \sigma_{11} \ldots \sigma_{1k_1}\ \mid\ \cdots\ \mid\ \mathsf{C}_n\ \sigma_{n1} \ldots \sigma_{nk_n}$$

Datatypes are a derived concept in HOL [22]; however, our analysis benefits from handling them specially as opposed to unfolding the underlying construction.

The datatype declaration introduces the type constructor $\kappa$, together with the term constructors $\mathsf{C}_i$ of type $\sigma_{i1} \to_{\mathsf{G}} \cdots \to_{\mathsf{G}} \sigma_{ik_i} \to_{\mathsf{G}} \bar{\alpha}\ \kappa$. The type $\bar{\alpha}\ \kappa$ may occur recursively in the $\sigma_{ij}$'s, but only in positive positions. For simplicity, we assume that any arrows in the $\sigma_{ij}$'s already carry annotations. (In the implementation, annotation variables are used to infer them.) The interpretation $[\![\bar\alpha\ \kappa]\!]_S$ is given by the corresponding free term algebra.

We must now extend the basic definitions of $\sqsubseteq$, $\leq$, and $\mathsf{AT}^s$ to this new construct. For Definition 4.38, we add the following case:

$$C_i(a_1,\ldots,a_{k_i}) \sqsubseteq^{\bar\tau\ \kappa} C_i(b_1,\ldots,b_{k_i}) \quad\text{iff}\quad \forall j \in \{1,\ldots,k_i\}.\ a_j \sqsubseteq^{\sigma_{ij}[\bar\alpha\mapsto\bar\tau]} b_j$$

Similarly, Definition 4.39 is extended with

$$\mathsf{AT}^s(\bar\tau\ \kappa) = \bigcup_{\substack{1\leq i\leq n\\ 1\leq j\leq k_i}} \mathsf{AT}^s(\sigma_{ij}[\bar\alpha\mapsto\bar\tau])$$

and Definition 4.48 with

$$\frac{\sigma_{ij}[\bar\alpha\mapsto\bar\tau] \leq \sigma_{ij}[\bar\alpha\mapsto\bar\tau'] \quad \text{for all } 1\leq i\leq n,\ 1\leq j\leq k_i}{\bar\tau\ \kappa \leq \bar\tau'\ \kappa}$$

To extend our soundness result, we must show that Lemmas 4.41 to 4.50 still hold. The proofs are straightforward. Constancy of the datatype constructors also follows directly from the above definitions.

**Example 4.64.** A theory of lists could comprise the following definitions:

**primrec** $@^{\alpha\ list\to\alpha\ list\to\alpha\ list}$ **where**
$\mathsf{Nil}\ @\ ys = ys$
$\mathsf{Cons}\ x\ xs\ @\ ys = \mathsf{Cons}\ x\ (xs\ @\ ys)$

**primrec** $\mathsf{set}^{\alpha\ list\to\alpha\to o}$ **where**
$\mathsf{set}\ \mathsf{Nil} = \emptyset$
$\mathsf{set}\ (\mathsf{Cons}\ x\ xs) = \{x\} \cup \mathsf{set}\ xs$

**primrec** $\mathsf{distinct}^{\alpha\ list\to o}$ **where**
$\mathsf{distinct}\ \mathsf{Nil} \longleftrightarrow \mathsf{True}$
$\mathsf{distinct}\ (\mathsf{Cons}\ x\ xs) \longleftrightarrow x \notin \mathsf{set}\ xs \land \mathsf{distinct}\ xs$

The table below presents the results of our analyses on three lemmas about lists.

|  | MONOTONIC |  |  | ANTIMONO. |  |  |
|---|---|---|---|---|---|---|
| FORMULA | $\mathfrak{M}_1$ | $\mathfrak{M}_2$ | $\mathfrak{M}_3$ | $\mathfrak{M}_1$ | $\mathfrak{M}_2$ | $\mathfrak{M}_3$ |
| $\mathsf{set}\ (xs\ @\ ys) = \mathsf{set}\ xs \cup \mathsf{set}\ ys$ | · | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\mathsf{distinct}\ (xs\ @\ ys) \longrightarrow \mathsf{distinct}\ xs \land \mathsf{distinct}\ ys$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\mathsf{distinct}\ (xs\ @\ ys) \longrightarrow \mathsf{set}\ xs \cap \mathsf{set}\ ys = \emptyset$ | · | ✓ | ✓ | ✓ | ✓ | ✓ |

### 4.5.3 Evaluation

What proportion of monotonic formulas are detected as such by our calculi? We applied Nitpick's implementations of $\mathfrak{M}_1$, $\mathfrak{M}_2$, and $\mathfrak{M}_3$ on the user-supplied theorems from six highly polymorphic Isabelle theories. In the spirit of counterexample generation, we conjoined the negated theorems with the relevant axioms. The results are given below.

| | Formulas | | | | Success Rate | | |
|---|---|---|---|---|---|---|---|
| Theory | $\mathfrak{M}_1$ | $\mathfrak{M}_2$ | $\mathfrak{M}_3$ | Total | $\mathfrak{M}_1$ | $\mathfrak{M}_2$ | $\mathfrak{M}_3$ |
| *AVL2* | 29 | 33 | 33 | 33 | 88% | 100% | 100% |
| *Fun* | 71 | 94 | 116 | 118 | 60% | 80% | 98% |
| *Huffman* | 46 | 91 | 90 | 99 | 46% | 92% | 91% |
| *List* | 441 | 510 | 545 | 659 | 67% | 77% | 83% |
| *Map* | 113 | 117 | 117 | 119 | 95% | 98% | 98% |
| *Relation* | 64 | 87 | 100 | 155 | 41% | 56% | 65% |

The table indicates how many formulas were found to involve at least one monotonic atomic type using $\mathfrak{M}_1$, $\mathfrak{M}_2$, and $\mathfrak{M}_3$, over the total number of formulas involving atomic types in the six theories. Since the formulas are all negated theorems, they are all semantically monotonic (no models exist for any scope).

Ideally, we would have evaluated the calculi on a representative database including (negated) non-theorems, but we lack such a database. Nonetheless, our experience suggests that the calculi perform as well on non-theorems as on theorems, because realistic non-theorems tend to use equality and quantifiers in essentially the same way as theorems. Interestingly, non-theorems that are derived from theorems by omitting an assumption or mistyping a variable name are even more likely to pass the monotonicity check than the corresponding theorems.

Although the study of monotonicity is interesting in its own right and leads to an elegant theory, our main motivation—speeding up model finders—is resolutely pragmatic. For Nitpick, which uses a default upper bound of 10 on the cardinality of the atomic types, we observed a speed increase factor of about 6 per inferred monotonic type. Since each monotonic type reduces the number of scopes to consider by a factor of 10, we could perhaps expect a 10-fold speed increase; however, the scopes that can be omitted by exploiting monotonicity are smaller and faster to check than those that are actually checked.

The time spent performing the monotonicity analysis (i.e., generating the annotation constraints and solving the resulting SAT problem) for $\mathfrak{M}_2$ is negligible. For $\mathfrak{M}_3$, the SAT solver (Jerusat [132]) occasionally reached the time limit of one second, which explains why $\mathfrak{M}_2$ beat $\mathfrak{M}_3$ on the *Huffman* theory.

## 4.6 Related Work

Monotonicity has been studied in the context of Alloy before. Although Alloy's logic is unsorted, models must give a semantics to "primitive types," which are sets of uninterpreted atoms. Early versions of the logic ensured monotonicity with respect to the primitive types by providing only bounded quantification and disallowing explicit references to the sets that denote the types [95]. Monotonicity has been lost in more recent versions of Alloy, which allow such references [94, p. 165]. Nonetheless, many Alloy formulas are monotonic, notably those in existential–bounded-universal form [107].

The satisfiability modulo theories (SMT) community has also shown interest in monotonicity. The original Nelson–Oppen method allows the combination of decision procedures for first-order theories meeting certain requirements, notably that the theories are stably infinite (or finitely unsatisfiable) [133]. This criterion has since been weakened, and one of the remaining requirements on the theories to be composed is that they be "smooth"—that is, every quantifier-free formula must be monotonic modulo the theory [186]. Smoothness is usually proved with pen and paper for the theories that need to be combined.

For some logics, small model theorems provide an upper bound on the cardinality of a sort [58], primitive type [131], or variable's domain [156]. If no models exist below that bound, no larger models exist. The Alloy Analyzer and Paradox exploit such theorems to terminate early. Our approach is complementary and could be called a *large* model theorem.

We presented an earlier version of this chapter at IJCAR 2010 [32]. Following on the IJCAR 2010 paper, Claessen et al. [61] devised two calculi for first-order logic similar to ours. Their first calculus is slightly stronger than $\mathfrak{M}_1$ in that it infers $\forall X^{\tilde{\alpha}}.\ \mathsf{f}(X) = \mathsf{g}(X)$ monotonic, by ensuring that the extensions of $\mathsf{f}$ and $\mathsf{g}$ coincide on new values. Their second calculus handles bounded quantifications specially; it is closely related to $\mathfrak{M}_3$ but was developed independently. Each predicate can be false- or true-extended, corresponding to our $\mathsf{F}$- and $\mathsf{T}$-annotations. The inference problem is NP-complete, and they use a SAT solver to determine which predicates should be false-extended and which should be true-extended. They observe that monotonic types can be safely erased when translating from simply typed to untyped first-order logic and exploit this in their new translation tool Monotonox, with applications in both theorem provers and model finders. A refinement of their approach is now implemented in Sledgehammer to encode HOL type information soundly and efficiently (Section 6.5).

When your hammer is C++,
everything begins to look like a thumb.

— Steve Haflich (1994)

# Chapter 5

# Case Study: Nitpicking C++ Concurrency

Previous work formalized the C++ memory model in Isabelle/HOL in an effort to clarify the new standard's semantics. Here we employ the model finder Nitpick to check litmus test programs that exercise the memory model, including a simple locking algorithm. This is joint work with Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar [37].

## 5.1 Background

To be reasonably confident of the correctness of a sequential program, we can often get by with informal arguments based on a reading of the source code. However, when it comes to concurrent programs, their inherent nondeterminism makes it extremely difficult to gain confidence of their correctness purely on the basis of informal or semiformal reasoning. Ten-line programs can have millions of possible executions. Subtle race condition bugs can remain hidden for years despite extensive testing and code reviews before they start causing failures. Even tiny concurrent programs expressed in idealized languages with clean mathematical semantics can be amazingly subtle [68, §1.4].

In the real world, performance considerations prompt hardware designers and compiler writers to further complicate the semantics of concurrent programs. For example, at the hardware level, a write operation taking place at instant $t$ might not yet be reflected in a read at $t + 1$ from a different thread because of cache effects. The final authority in this matter is the processor's memory consistency model.

The Java language abstracts the processor memory models, and compiler reordering, behind a software memory model designed to be efficiently implementable on actual hardware. However, the original Java model was found to be flawed [159], and even the revised version had some unpleasant surprises in store [52, 170, 188].

The new C++ standard, C++11 [1], attempts to provide a clear semantics for concurrent programs, including a memory model and library functions. Batty et al. [14–16] formalized a large fragment of the prose specification in Isabelle/HOL [140] (Section 5.2). From the Isabelle formalization, they extracted the core of a tool, CPPMEM, that can check litmus test programs—small multithreaded programs that exercise various aspects of the memory model. As we have come to expect, the researchers found flaws in the original prose specification and clarified several issues, which have now been addressed by the standard. To validate the semantics, they proved the correctness of proposed Intel x86 and IBM POWER implementations of the concurrency primitives.

In this chapter, we are interested in tool support for verifying litmus test programs. CPPMEM exhaustively enumerates the possible executions of the program, checking each against the (executable) formal semantics. An attractive alternative is to apply a SAT solver to the memory model constraints and litmus tests. The MemSAT tool's success on the Java memory model [188] and experiments described in Batty et al. [16, §7] suggest that SAT solvers scale better than explicit-state model checkers, allowing us to verify more complex litmus tests. Nitpick and its predecessor Refute [198] featured in several case studies [25, 34, 98, 120, 197] but were never successfully applied to a specification as complex as the C++ memory model.

Although the memory model specification was not designed with SAT solving in mind, we expected that with some adjustments it should be within Nitpick's reach. The specification is written in a fairly abstract and axiomatic style, which should favor SAT solvers. Various Kodkod optimizations help cope with large problems. Moreover, although the memory model is subtle and complicated, the specification is mostly restricted to first-order logic with sets, transitive closure, and inductive datatypes, all of which are handled efficiently in Nitpick or Kodkod.

Initially, though, we had to make drastic changes to the specification so that Nitpick would scale to handle the simplest litmus tests in reasonable time. These early results had been obtained at the cost of several days' work by people who understood Nitpick's internals. Based on our experience adapting the specification by hand, we proceeded to address scalability issues directly in the tool. This resulted in three new optimizations, described in a previous chapter: heuristic constant unfolding (Section 3.4.6), necessary datatype values (Section 3.4.7), and a lightweight translation (Section 3.4.8). With the optimizations in place, a few minor adjustments to the original specification sufficed to support efficient model finding. We applied the optimized version of Nitpick to several litmus tests (Section 5.3), including a simple sequential locking algorithm, thereby increasing our confidence in the specification's adequacy. Litmus tests that were previously too large for CPPMEM can now be checked within minutes.

## 5.2 The C++ Memory Model

The C++ International Standard [1] defines the concurrency memory model axiomatically, by imposing constraints on the order of memory accesses in program executions. The semantics of a program is a set of allowed executions. Here we briefly present the memory model and its Isabelle formalization [14–16], focusing on the aspects that are necessary to understand the rest of this chapter.

### 5.2.1 Introductory Example

To facilitate efficient implementations on modern parallel architectures, the C++ memory model (like other relaxed memory models) has no global linear notion of time. Program executions are not guaranteed to be *sequentially consistent* (*SC*)—that is, equivalent to a simple interleaving of threads [110]. The following program fragment is a simple example that can exhibit non-SC behavior:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
    printf("y: %d\n", y.load(ord_relaxed));
|||
    y.store(1, ord_relaxed);
    printf("x: %d\n", x.load(ord_relaxed));
}}}
```

(To keep examples short, we use the notation `{{{ ... ||| ... }}}` for parallel composition and `ord_`*xxx* for `memory_order_`*xxx*.)

Two threads write to separate memory locations `x` and `y`; then each thread reads from the other location. Can both threads read the original value of the location they read from? According to the standard, they can. The program has eight outputs that are permitted by the memory model:

```
x: 0 | x: 0 | x: 1 | x: 1 | y: 0 | y: 0 | y: 1 | y: 1
y: 0 | y: 1 | y: 0 | y: 1 | x: 0 | x: 1 | x: 0 | x: 1
```

Among these, the two outputs

```
x:0       y:0
     and
y:0       x:0
```

exhibit the counterintuitive non-SC behavior.

### 5.2.2 Memory Actions and Orders

From the memory model's point of view, program executions consist of *memory actions*. The main actions are

| | |
|---|---|
| L x | locking of x |
| U x | unlocking of x |
| $R_{ord}\ x = v$ | atomic read of value $v$ from x |
| $W_{ord}\ x = v$ | atomic write of value $v$ to x |
| $RMW_{ord}\ x = v_1/v_2$ | atomic read–modify–write at x |
| $R_{na}\ x = v$ | nonatomic read of value $v$ from x |
| $W_{na}\ x = v$ | nonatomic write of value $v$ to x |
| $F_{ord}$ | fence |

where *ord*, an action's *memory order*, can be any of the following:

| | | |
|---|---|---|
| sc | `ord_seq_cst` | sequentially consistent |
| rel | `ord_release` | release |
| acq | `ord_acquire` | acquire |
| a/r | `ord_acq_rel` | acquire and release |
| con | `ord_consume` | consume |
| rlx | `ord_relaxed` | relaxed |

Memory orders control synchronization and ordering of atomic actions. The `ord_seq_cst` order provides the strongest guarantees (SC), while `ord_relaxed` provides the weakest guarantees. The release/acquire discipline, where writes use `ord_release` and reads use `ord_acquire`, occupies an intermediate position on the continuum. The weaker variant release/consume, with `ord_consume` for the reads, can be implemented more efficiently on hardware with weak memory ordering.

If we wanted to prohibit the non-SC executions in the program above, we could simply pass `ord_seq_cst` as argument to the `load` and `store` functions instead of `ord_relaxed`.

### 5.2.3 Original Formalization

In place of a global timeline, the standard postulates several relations over different subsets of a program's memory actions. These relations establish some weak notion of time. They are not necessarily total or transitive (and can therefore be hard to understand intuitively) but must satisfy various constraints.

In the Isabelle formalization of the memory model, a candidate execution $\mathcal{X}$ is a pair $\langle \mathcal{X}_{\mathsf{opsem}}, \mathcal{X}_{\mathsf{witness}} \rangle$. The component $\mathcal{X}_{\mathsf{opsem}}$ specifies the program's memory actions (*acts*), its threads (*thrs*), a simple typing of memory locations (*lk*), and four relations over actions (*sb*, *asw*, *dd*, and *cd*) that constrain their evaluation order. The other component, $\mathcal{X}_{\mathsf{witness}}$, consists of three relations: Read actions *read from* some write action (*rf*), *sequentially consistent* actions are totally ordered (*sc*), and a *modification order* (*mo*) gives a per-location linear-coherence order for atomic writes.

$\mathcal{X}_{\mathsf{opsem}}$ is given by the program's operational semantics and can be determined statically from the program source. $\mathcal{X}_{\mathsf{witness}}$ is existentially quantified. The memory model imposes constraints on the components of $\mathcal{X}_{\mathsf{opsem}}$ and $\mathcal{X}_{\mathsf{witness}}$ as well as on various relations derived from them. A candidate execution is *consistent* if it satisfies these constraints. The top-level definition of consistency is shown below:

```
definition consistent_execution acts thrs lk sb asw dd cd rf mo sc ≡
    well_formed_threads acts thrs lk sb asw dd cd ∧
    well_formed_reads_from_mapping acts lk rf ∧
    consistent_locks acts thrs lk sb asw dd cd sc ∧
    let
      rs   = release_sequence acts lk mo
      hrs  = hypothetical_release_sequence acts lk mo
      sw   = synchronizes_with acts sb asw rf sc rs hrs
      cad  = carries_a_dependency_to acts sb dd rf
      dob  = dependency_ordered_before acts rf rs cad
```

$$
\begin{array}{l}
\quad\begin{array}{lcl}
\mathit{ithb} & = & \textsf{inter\_thread\_happens\_before}\ \mathit{acts\ thrs\ lk\ sb\ asw\ dd\ cd\ sw\ dob} \\
\mathit{hb} & = & \textsf{happens\_before}\ \mathit{acts\ thrs\ lk\ sb\ asw\ dd\ cd\ ithb} \\
\mathit{vse} & = & \textsf{visible\_side\_effect}\ \mathit{acts\ thrs\ lk\ sb\ asw\ dd\ cd\ hb}
\end{array} \\
\text{in} \\
\quad \textsf{consistent\_inter\_thread\_happens\_before}\ \mathit{acts\ ithb}\ \wedge \\
\quad \textsf{consistent\_sc\_order}\ \mathit{acts\ thrs\ lk\ sb\ asw\ dd\ cd\ mo\ sc\ hb}\ \wedge \\
\quad \textsf{consistent\_modification\_order}\ \mathit{acts\ thrs\ lk\ sb\ asw\ dd\ cd\ sc\ mo\ hb}\ \wedge \\
\quad \textsf{consistent\_reads\_from\_mapping}\ \mathit{acts\ thrs\ lk\ sb\ asw\ dd\ cd\ rf\ sc\ mo\ hb\ vse}
\end{array}
$$

The derived relations (*rs*, *hrs*, ..., *vse*) and the various consistency conditions follow the C++ standard; we omit their definitions. The complete memory model comprises approximately 1200 lines of Isabelle text.

### 5.2.4 CPPMEM

For any given $\mathcal{X}_{\mathsf{opsem}}$, there may be one, several, or perhaps no choices for $\mathcal{X}_{\mathsf{witness}}$ that give rise to a consistent execution. Since the memory model is complex, and the various consistency conditions and their interactions can be difficult to understand intuitively, tool support for exploring the model and computing the possible behaviors of C++ programs is highly desirable.

The CPPMEM tool [16] was designed to assist with these tasks. It consists of three parts: (1) a preprocessor that computes the $\mathcal{X}_{\mathsf{opsem}}$ component of a candidate execution from a C++ program's source code; (2) a search procedure that enumerates the possible values for $\mathcal{X}_{\mathsf{witness}}$; (3) a checking procedure that calculates the derived relations and evaluates the consistency conditions for each pair $\langle \mathcal{X}_{\mathsf{opsem}}, \mathcal{X}_{\mathsf{witness}} \rangle$.

For the second part, the CPPMEM developers refrained from "coding up a sophisticated memory-model-aware search procedure in favour of keeping this part of the code simple" [16]. Their code enumerates all possible combinations for the *rf*, *mo*, and *sc* relations that respect a few basic constraints:

- *sc* only contains SC actions and is a total order over them.
- *mo* only contains pairs $(a, b)$ such that $a$ and $b$ write to the same location; for each location, *mo* is a total order over the set of writes at this location.
- *rf* only contains pairs $(a, b)$ such that $a$ writes a given value to a location and $b$ reads the same value from that location; for each read $b$, it contains exactly one such pair.

Because the search space grows asymptotically with $n!$ in the worst case, where $n$ is the number of actions in the program execution, CPPMEM is mostly limited to small litmus tests, which typically involve up to eight actions. This does cover many interesting tests, but not larger parallel algorithms.

Writing a more sophisticated search procedure would require a detailed understanding of the memory model (which we hope to gain through proper tool support in the first place) and could introduce errors that are difficult to detect—unless, of course, the procedure was automatically generated from the formal specification. This is where Nitpick comes into play.

### 5.2.5 Fine-Tuned Formalization

With the optimizations described in Sections 3.4.6 to 3.4.8 in place, Nitpick handles the memory model specification reasonably efficiently without any modifications. Nonetheless, it proves worthwhile to fine-tune the specification in three respects.

First, the types *act_id*, *thr_id*, *loc*, and *val*—corresponding to action IDs, thread IDs, memory locations, and values—are defined as aliases for *nat*. This is unfortunate because it prevents us from specifying different cardinalities for the different notions. A litmus test involving eight actions, five threads, two locations, and two values gives rise to a much smaller SAT problem if we tell Nitpick to use the cardinalities $|act\_id| = 8$, $|thr\_id| = 5$, and $|loc| = |val| = 2$ than if all four types are set to have cardinality 8. To solve this, we replace the aliases

**type_synonym** *act_id* = *nat*
**type_synonym** *thr_id* = *nat*
**type_synonym** *loc* = *nat*

with copies of the natural numbers:

**datatype** *act_id* = A0 | ASuc *act_id*
**datatype** *thr_id* = T0 | TSuc *thr_id*
**datatype** *loc* = L0 | LSuc *loc*

For notational convenience, we define the following abbreviations:

$$\mathsf{a}_k \equiv \mathsf{ASuc}^k\ \mathsf{A0} \qquad \mathsf{t}_k \equiv \mathsf{TSuc}^k\ \mathsf{T0} \qquad \mathsf{x} \equiv \mathsf{L0} \qquad \mathsf{y} \equiv \mathsf{LSuc}\ \mathsf{L0}$$

Second, while the unfolding heuristic presented in Section 3.4.6 allowed us to remove many *nitpick_simp* attributes, we found that the heuristic was too aggressive with respect to two constants, which are better unfolded (using *nitpick_unfold*). We noticed them because they were the only relations of arity greater than 3 in the generated Kodkod problem. Both were called with a higher-order argument that was not eligible for specialization.

Third, some of the basic definitions in the specification are gratuitously inefficient for SAT solving. The specification had its own definition of relational composition and transitive closure, but it is preferable to replace them with equivalent concepts from Isabelle's libraries, which are mapped to appropriate FORL constructs. This is achieved by providing lemmas that redefine the memory model's constants in terms of the desired Isabelle concepts:

**lemma** [*nitpick_unfold*]: compose $R\ S = R \circ S$
**lemma** [*nitpick_unfold*]: tc $A\ R =$ (restrict_relation $R\ A)^+$

These lemmas are part of the separate theory file mentioned above. We do not even need to prove them; since they are intended for model finding, it is enough to assert them with the **sorry** placeholder and check them with Nitpick.

Similarly, we supply a more compact definition of the predicate strict_total_order_over $A\ R$. The original definition cleanly separates the constraints expressing the relation's domain, irreflexivity, transitivity, and totality:

$$\begin{aligned}
&\big(\forall (a,b) \in R.\ a \in A \wedge b \in A\big) \wedge \\
&\big(\forall x \in A.\ (x,x) \notin R\big) \wedge \\
&\big(\forall x \in A.\ \forall y \in A.\ \forall z \in A.\ (x,y) \in R \wedge (y,z) \in R \longrightarrow (x,z) \in R\big) \wedge \\
&\big(\forall x \in A.\ \forall y \in A.\ (x,y) \in R \vee (y,x) \in R \vee x = y\big)
\end{aligned}$$

The optimized formulation

$$\begin{aligned}
&\big(\forall x\ y.\ \text{if } (x,y) \in R \text{ then} \\
&\qquad\qquad \{x,y\} \subseteq A \wedge x \neq y \wedge (y,x) \notin R \\
&\qquad\quad \text{else} \\
&\qquad\qquad \{x,y\} \subseteq A \wedge x \neq y \longrightarrow (y,x) \in R\big) \wedge \\
&R^{+} = R
\end{aligned}$$

reduces the number of occurrences of $A$ and $R$, both of which are higher-order arguments that can be specialized to arbitrarily large terms.

## 5.3 Litmus Tests

We evaluated Nitpick on several litmus tests designed to illustrate the semantics of the C++ memory model. Most of these litmus tests had been checked by CPPMEM and the unoptimized version of Nitpick before [16], so it should come as no surprise that our experiments did not reveal any flaws in the C++ standard. We did, however, discover many mistakes in the formalization, such as missing parentheses and typos (e.g., ∀ instead of ∃). These mistakes had been accidentally introduced during maintenance [143, §2] and had gone unnoticed even though the formalization is used as a basis for formal proofs.

Our experience illustrates once again the need to validate complex specifications, to ensure that the formal artifact correctly captures the intended semantics of the informal one (in our case, the C++ standard).

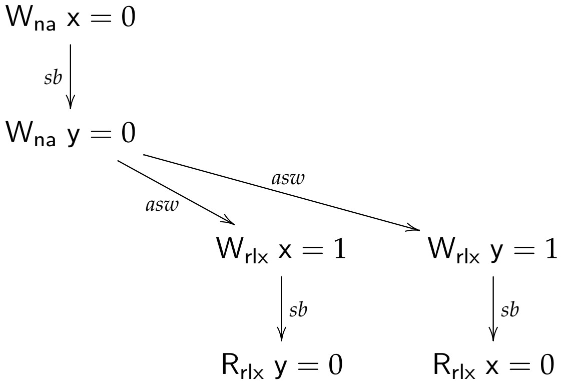
### 5.3.1 Store Buffering

The first test is simply the introductory example from Section 5.2:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
    printf("y: %d\n", y.load(ord_relaxed));
|||
    y.store(1, ord_relaxed);
    printf("x: %d\n", x.load(ord_relaxed));
}}}
```

This program has six actions: two nonatomic initialization writes ($\mathsf{W}_{\mathsf{na}}$), then one relaxed write ($\mathsf{W}_{\mathsf{rlx}}$) and one relaxed read ($\mathsf{R}_{\mathsf{rlx}}$) in each thread. The diagram below shows the relations *sb* and *asw*, which are part of $\mathcal{X}_{\mathsf{opsem}}$ and hence fixed by the program's operational semantics:


$\mathsf{W}_{\mathsf{na}}\ \mathsf{x} = 0$
*sb*
$\mathsf{W}_{\mathsf{na}}\ \mathsf{y} = 0$
*asw*
*asw*
$\mathsf{W}_{\mathsf{rlx}}\ \mathsf{x} = 1$
$\mathsf{W}_{\mathsf{rlx}}\ \mathsf{y} = 1$
*sb*
*sb*
$\mathsf{R}_{\mathsf{rlx}}\ \mathsf{y} = 0$
$\mathsf{R}_{\mathsf{rlx}}\ \mathsf{x} = 0$

Each vertex represents an action, and an $r$-edge from $a$ to $b$ indicates that $(a, b) \in r$. The actions are arranged in three columns corresponding to the threads they belong to. For transitive relations, we omit transitive edges.

To check a litmus test with Nitpick, we must provide $\mathcal{X}_{\mathsf{opsem}}$. We can use CPPMEM's preprocessor to compute $\mathcal{X}_{\mathsf{opsem}}$ from a C++ program's source code, but for simple programs such as this one, we can also translate the code manually. We first declare abbreviations for the test's actions:

**abbreviation** a $\equiv$ Store $\mathsf{a}_0$ $\mathsf{t}_0$ x 0
**abbreviation** b $\equiv$ Store $\mathsf{a}_1$ $\mathsf{t}_0$ y 0
**abbreviation** c $\equiv$ Atomic_store $\mathsf{a}_2$ $\mathsf{t}_1$ Ord_relaxed x 1
**abbreviation** d $\equiv$ Atomic_load $\mathsf{a}_3$ $\mathsf{t}_1$ Ord_relaxed y 0
**abbreviation** e $\equiv$ Atomic_store $\mathsf{a}_4$ $\mathsf{t}_2$ Ord_relaxed y 1
**abbreviation** f $\equiv$ Atomic_load $\mathsf{a}_5$ $\mathsf{t}_2$ Ord_relaxed x 0

The read actions, d and f, specify the value they expect to find as last argument to the constructor. That value is 0 for both threads because we are interested only in non-SC executions, in which the write actions c and e are disregarded by the reads.

Next, we introduce the components of $\mathcal{X}_{\mathsf{opsem}}$ as constants:

**definition** [*nitpick_simp*]: acts $\equiv \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}\}$
**definition** [*nitpick_simp*]: thrs $\equiv \{\mathsf{t}_0, \mathsf{t}_1, \mathsf{t}_2\}$
**definition** [*nitpick_simp*]: lk $\equiv (\lambda\_.\ \mathsf{Atomic})$
**definition** [*nitpick_simp*]: sb $\equiv \{(\mathsf{a}, \mathsf{b}),\ (\mathsf{c}, \mathsf{d}),\ (\mathsf{e}, \mathsf{f})\}$
**definition** [*nitpick_simp*]: asw $\equiv \{(\mathsf{b}, \mathsf{c}),\ (\mathsf{b}, \mathsf{e})\}$
**definition** [*nitpick_simp*]: dd $\equiv \emptyset$
**definition** [*nitpick_simp*]: cd $\equiv \emptyset$

Specialization implicitly propagates these values to where they are needed in the specification. To avoid clutter and facilitate subexpression sharing, we disable unfolding by specifying *nitpick_simp*.

Finally, we look for a model satisfying

$$\mathsf{consistent\_execution\ acts\ thrs\ lk\ sb\ asw\ dd\ cd}\ rf\ mo\ sc$$

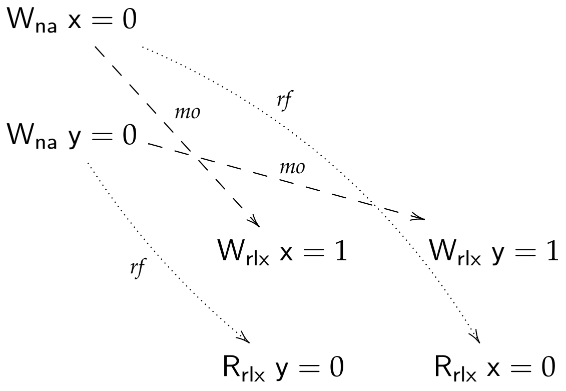
where *rf*, *mo*, and *sc* are free variables corresponding to $\mathcal{X}_{\mathsf{witness}}$. In the interest of academic transparency, the Nitpick call is shown below in all its unpleasantness:

```
nitpick [
  satisfy,              look for a model
  need = a b c d e f,   the necessary actions (Section 3.4.7)
  card act = 6,         six actions (a, b, c, d, e, f)
  card act_id = 6,      six action IDs (a0, a1, a2, a3, a4, a5)
  card thr_id = 3,      three thread IDs (t0, t1, t2)
  card loc = 2,         two locations (x, y)
  card val = 2,         two values (0, 1)
  card = 10,            maximum cardinality for other types
  total_consts,         use lightweight translation (Section 3.4.8)
  finitize act,         pretend act is finite
  dont_box              disable boxing (Section 3.4.2)
]
```

With these options, Nitpick needs 4.7 seconds to find relations that witness a non-SC execution:

$$mo = \{(\mathsf{a},\mathsf{c}),\ (\mathsf{b},\mathsf{e})\} \qquad rf = \{(\mathsf{a},\mathsf{f}),\ (\mathsf{b},\mathsf{d})\} \qquad sc = \emptyset$$

The relations *rf* and *mo* are shown in the diagram below:



The modification-order relation (*mo*) reveals that the assignments $\mathsf{x} = 1$ and $\mathsf{y} = 1$ take place after the initializations to 0, but the read-from relation (*rf*) indicates that the two reads get their values from the initializations and not from the assignments.

If we replace all four relaxed memory accesses with SC atomics, such non-SC behavior is no longer possible. Nitpick verifies this in 4.0 seconds by reporting the absence of suitable witnesses. These results are consistent with our understanding of the C++ standard.

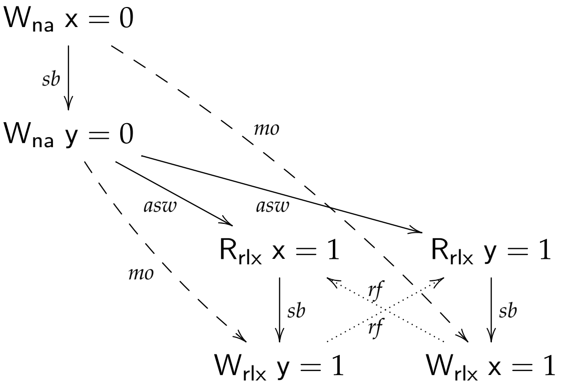
### 5.3.2 Load Buffering

This test is dual to the Store Buffering test. Two threads read from separate locations; then each thread writes to the other location:

```
atomic_int x = 0;
atomic_int y = 0;
```

```
{{{
    printf("x: %d\n", x.load(ord_relaxed));
    y.store(1, ord_relaxed);
|||
    printf("y: %d\n", y.load(ord_relaxed));
    x.store(1, ord_relaxed);
}}}
```

With relaxed atomics, each thread can observe the other thread's "later" write. Nitpick finds the following execution in 4.2 seconds:



Nitpick verifies the absence of a non-SC execution with release/consume, release/acquire, and SC atomics in 4.1 seconds.

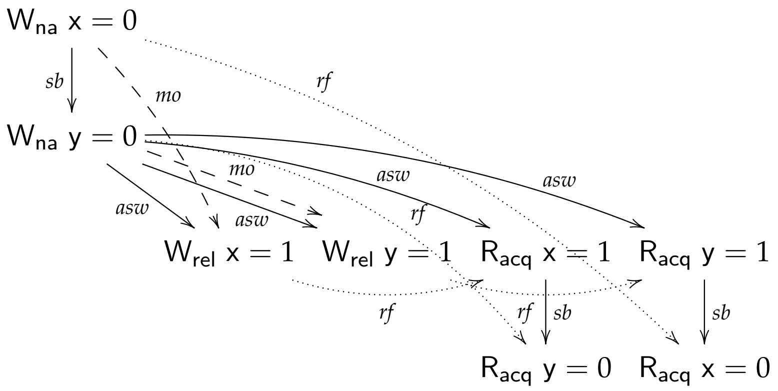
### 5.3.3 Independent Reads of Independent Writes

Two writer threads independently write to x and y, and two readers read from both data locations:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_release);
|||
    y.store(1, ord_release);
|||
    printf("x1: %d\n", x.load(ord_acquire));
    printf("y1: %d\n", y.load(ord_acquire));
|||
    printf("y2: %d\n", y.load(ord_acquire));
    printf("x2: %d\n", x.load(ord_acquire));
}}}
```

With release/acquire, release/consume, and relaxed actions, different readers can observe these writes in opposite order. Nitpick finds an execution in 5.8 seconds:



With SC actions, this behavior is not allowed, and Nitpick verifies the absence of a non-SC execution in 5.2 seconds.

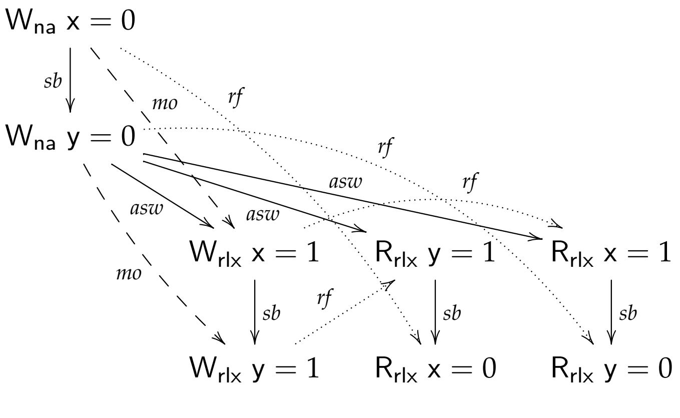
### 5.3.4 Message Passing

We consider a variant of message passing where one thread writes first to a data location x and then to a flag y, while two reading threads read both the flag and the data. There are two initialization writes and two actions in each thread, for a total of eight actions:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
    y.store(1, ord_relaxed);
|||
    printf("y1: %d\n", y.load(ord_relaxed));
    printf("x1: %d\n", x.load(ord_relaxed));
|||
    printf("x2: %d\n", x.load(ord_relaxed));
    printf("y2: %d\n", y.load(ord_relaxed));
}}}
```

Because all non-initialization actions are relaxed atomics, the two readers can observe the writes in opposite order. Nitpick finds a witness execution in 5.7 seconds:



On the other hand, if the flag is accessed via a release/acquire pair, or via SC atomics, the first reader is guaranteed to observe the data written by the writer thread. Nitpick finds such an execution (without the second reader) in 4.0 seconds, and verifies the absence of a non-SC execution also in 4.0 seconds.

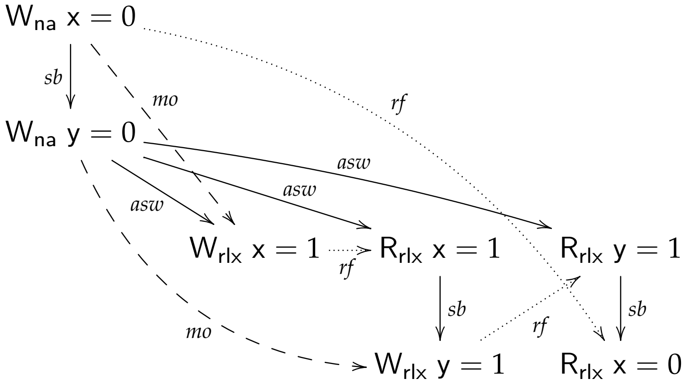
### 5.3.5 Write-to-Read Causality

This test spawns three auxiliary threads in addition to the initialization thread:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
|||
    printf("x1: %d\n", x.load(ord_relaxed));
    y.store(1, ord_relaxed);
|||
    printf("y: %d\n", y.load(ord_relaxed));
    printf("x2: %d\n", x.load(ord_relaxed));
}}}
```

The first auxiliary thread writes to x; the second thread reads from x and writes to y; the third thread reads from y and then from x. With relaxed atomics, the third thread does not necessarily observe the first thread's write to x even if it observes the second thread's write to y and the second thread observes the first thread's write to x. Nitpick finds this execution in 4.2 seconds:



The memory model guarantees write-to-read causality for release/acquire and SC actions. Nitpick verifies the absence of a non-SC execution in 4.4 seconds.

### 5.3.6 Sequential Lock

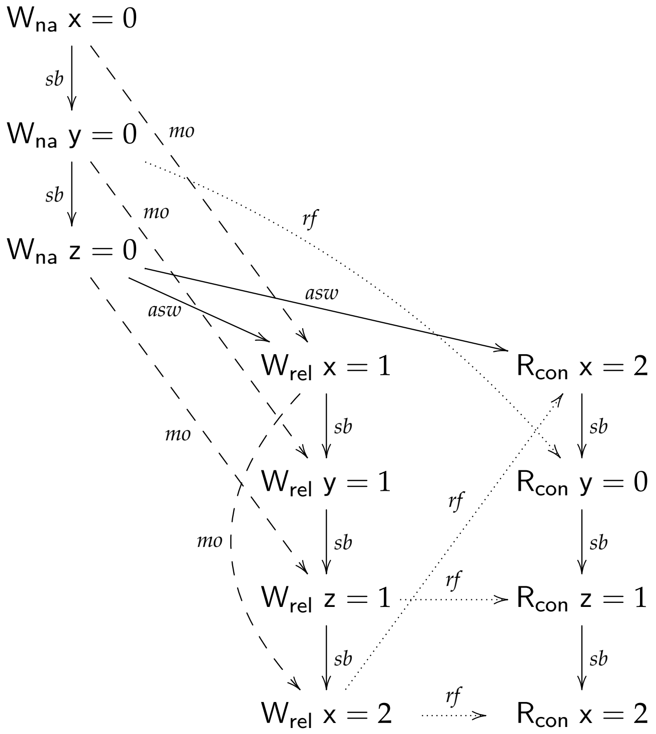
This test is more ambitious. The program models a simple sequential locking algorithm inspired by the Linux kernel's "seqlock" mechanism [109]:

```
atomic_int x = 0;
atomic_int y = 0;
atomic_int z = 0;
```

```
{{{
    for (int i = 0; i < N; i++) {
        x.store(2 * i + 1, ord_release);
        y.store(i + 1, ord_release);
        z.store(i + 1, ord_release);
        x.store(2 * i + 2, ord_release);
    }
|||
    printf("x: %d\n", x.load(ord_consume));
    printf("y: %d\n", y.load(ord_consume));
    printf("z: %d\n", z.load(ord_consume));
    printf("x: %d\n", x.load(ord_consume));
}}}
```

The program spawns a writer and a reader thread. The writer maintains a counter x that gets incremented before and after all writes to the data locations y and z. The data is “locked” whenever the counter x is odd. The reader eventually accesses the data, but not without checking x before and after. A non-SC behavior occurs if the two reads of x yield the same even value (i.e., the lock was free during the two data reads) but y $\neq$ z (i.e., the data was observed while in an inconsistent state).

Already for N $= 1$, Nitpick finds the following non-SC execution, where the reader observes y $= 0$ and z $= 1$, in 15.8 seconds:



With release/acquire instead of release/consume, the algorithm should be free of non-SC behavior. Nitpick takes 15.8 seconds to exhaustively check the N $= 1$ case, 86 seconds for N $= 2$, and 378 seconds for N $= 3$. If we add a second reader thread, it takes 86 seconds for N $= 1$ and 379 seconds for N $= 2$.

Because of the loop, our analysis is incomplete: We cannot prove the absence of non-SC behavior for all bounds `N` (or for an arbitrary number of readers), only its presence. Nonetheless, the small-scope hypothesis, which postulates that "most bugs have small counterexamples" [94, §5.1.3], suggests that the sequential locking algorithm implemented in terms of release/acquire atomics is correct for any number of iterations and reader threads.

### 5.3.7 Generalized Write-to-Read Causality

Nitpick's run-time depends on the size of the search space, which is exponential in the number of actions. To demonstrate how Nitpick scales up to larger litmus tests, we generalize the Write-to-Read Causality test from 2 to $n$ locations. The generalized test consists of $2n$ writes (including $n$ initializations) and $n + 1$ reads, thus $3n + 1$ actions in total. Since the three witness variables are binary relations over actions, the state space is of size $2^{3(3n+1)^2}$.

With relaxed atomics, there is an execution where the last thread does not observe the first thread's write. With SC atomics, no such execution exists. The Nitpick run-times are tabulated below. For comparison, we also include the CPPMEM run-times (on roughly comparable hardware).

| LOCATIONS | ACTIONS | STATES | CPPMEM | | NITPICK | |
|---|---|---|---|---|---|---|
| ($n$) | ($3n+1$) | $\left(2^{3(3n+1)^2}\right)$ | RLX | SC | RLX | SC |
| 2 | 7 | $2^{147}$ | .0 s | .5 s | 4 s | 4 s |
| 3 | 10 | $2^{300}$ | .0 s | 90.5 s | 11 s | 11 s |
| 4 | 13 | $2^{507}$ | .1 s | $> 10^4$ s | 41 s | 40 s |
| 5 | 16 | $2^{768}$ | .2 s | $> 10^4$ s | 132 s | 127 s |
| 6 | 19 | $2^{1083}$ | .7 s | $> 10^4$ s | 384 s | 376 s |
| 7 | 22 | $2^{1452}$ | 2.5 s | $> 10^4$ s | 982 s | 977 s |

Every additional location slows down Nitpick's search by a factor of about 3. Although the search space grows with $2^{n^2}$, the search time grows slightly slower than $k^n$, which is asymptotically better than CPPMEM's $n!$ worst-case complexity. CPPMEM outperforms Nitpick on the relaxed version of the test because its basic constraints reduce the search space to just $2^n$ candidate orders for *rf* and *mo* (Section 5.2.4). On the other hand, CPPMEM scales much worse than Nitpick when the actions are SC, because it naively enumerates all $2^n \cdot (2n + 1)!$ combinations for *rf*, *mo*, and *sc* that meet the basic constraints.

## 5.4 Related Work

The discovery of fatal flaws in the original Java memory model [159] stimulated much research in software memory models. We attempt to cover the most relevant work, focusing on tool support. We refer to Batty et al. [16], Torlak et al. [188], and Yang et al. [206] for more related work.

MemSAT [188] is an automatic tool based on Kodkod specifically designed for debugging axiomatic memory models. It has been used on several memory models from the literature, including the Java memory model. A noteworthy feature of MemSAT is that it produces a minimal unsatisfiable core if the model or the litmus test is overconstrained. MemSAT also includes a component that generates relational constraints from Java programs, akin to CPPMEM's preprocessor. Nitpick's main advantage over MemSAT for our case study is that it understands higher-order logic, as opposed to Kodkod's relational logic.

NemosFinder [206] is another axiomatic memory model checker. Memory models are expressed as Prolog predicates and checked using either constraint logic programming or SAT solving. The tool includes a specification of the Intel Itanium memory model.

Visual-MCM [124] is a generic tool that checks and graphically displays given executions against a memory model specification. The tool was designed primarily as an aid to hardware designers.

While the above tools are generic, many tools target specific models. Manson and Pugh [116] developed two simulators for the Java memory model that enumerate the possible executions of a program. Java RaceFinder [99], an extension to Java PathFinder [193], is a modern successor. Both of these are explicit-state model checkers. Like CPPMEM (and its predecessor `memevents` [166]), they suffer from the state-explosion problem.

## 5.5 Discussion

We applied the model finder Nitpick to an Isabelle formalization of the C++ standard's memory model. Our experiments involved classical litmus tests and did not reveal any flaws in the C++ standard. This is no surprise: The model has already been validated by the CPPMEM simulator on several litmus tests, and the correctness proofs of the proposed Intel x86 and IBM POWER implementations give further evidence that the Isabelle model captures the standard's intended semantics.

Verifying the absence of a consistent non-SC execution for the Independent Reads of Independent Writes test takes about 5.2 seconds using Nitpick, compared with 5 minutes using CPPMEM [16, §7]. Larger SC tests that were never checked with CPPMEM are now checkable within minutes. There will always be cases where more dedicated tools are called for, but it is pleasing when a general-purpose tool outperforms dedicated solutions.

On small litmus tests, Nitpick remains significantly slower than CPPMEM, which takes less than a second on some of the tests. The bottleneck is the translation of the memory model into FORL and SAT. The SAT search is extremely fast for small tests and scales much better than CPPMEM's simplistic enumeration scheme. On the largest problems we considered, Nitpick takes a few seconds; then about 95% of the time is spent in Kodkod, while the rest is spent in MiniSat [74].

For some litmus tests, CPPMEM's basic constraints reduce the search space considerably. We could probably speed up Nitpick by incorporating these constraints into the model—for example, by formalizing *rf* as a map from reads to writes, rather than as a binary relation over actions. However, this would require extensive modifications to the formalization, which is undesirable.

The main challenge for a diagnosis tool such as Nitpick is that users of interactive theorem provers tend to write their specifications so as to make the actual proving easy. In contrast, if the Alloy Analyzer or MemSAT performs poorly on a specification, the tool's developers can put part of the blame on the users, arguing for example that they have "not yet assimilated the relational idiom" [107, p. 7]. We wish we could have applied Nitpick directly on the Isabelle specification of the memory model, but without changes to either Nitpick or the specification our approach would not have scaled to handle even the simplest litmus tests.

We were delighted to see that function specialization, one of the very first optimizations implemented in Nitpick, proved equal to the task. By propagating arguments to where they are needed, specialization ensures that no more than two arguments ever need to be passed at a call site—a dramatic reduction from the 10 or more arguments taken by many of the memory model's functions. Without this crucial optimization, we would have faced the unappealing prospect of rewriting the specification from scratch.

A proof is a proof. What kind of a proof? It's a proof. A proof is a proof, and when you have a good proof, it's because it's proven.

— Jean Chrétien (2002)

# Chapter 6

# Proof Discovery Using Automatic Theorem Provers

Sledgehammer [125, 153] is Isabelle's subsystem for harnessing the power of first-order automatic theorem provers (ATPs). Given a conjecture, it heuristically selects a few hundred relevant lemmas from Isabelle's libraries, translates them to untyped first-order logic along with the conjecture, and sends the resulting problem to ATPs (Section 6.2). Sledgehammer is very effective and has achieved great popularity with users, novices and experts alike.

The Cambridge team that originally developed Sledgehammer included Jia Meng, Lawrence Paulson, Claire Quigley, and Kong Woei Susanto. Over the years, the tool has been improved, tested, and evaluated by members of the Munich team—notably Sascha Böhme, Fabian Immler, Philipp Meyer, Tobias Nipkow, Makarius Wenzel, and the author of this thesis.

This chapter presents some of the latest developments. We extended the tool to invoke satisfiability modulo theories (SMT) solvers, which combine propositional reasoning and decision procedures for equality and arithmetic (Section 6.3). We developed sound and efficient encodings of higher-order features in untyped first-order logic to address long-standing soundness issues (Sections 6.4 and 6.5). We addressed a range of unglamorous but important technical issues (Section 6.6). We evaluated various aspects of the tool on a representative set of Isabelle theories (Section 6.7). Finally, we adapted the Isar proof construction code so that it delivers direct proofs (Section 6.8). Some of these improvements are joint work with Sascha Böhme, Charles Francis, Lawrence Paulson, Nicholas Smallbone, and Nik Sultana; precise credits are given below.

## 6.1 TPTP Syntax

The TPTP World [180] is a well-established infrastructure for supporting the automated reasoning community. It includes a vast problem library, the Thousands of Problems for Theorem Provers (TPTP) [179], as well as specifications of concrete syntaxes to facilitate interchange of problems and solutions: The untyped clause normal form (CNF) and first-order form (FOF) are implemented in dozens of reasoning tools, and a growing number of reasoners can process the "core" typed first-order form (TFF0) [183], which provides simple monomorphic types (sorts) and interpreted arithmetic, or the corresponding higher-order form (THF0) [20]. The TPTP community recently extended TFF0 with rank-1 polymorphism [35].

**CNF: Clause Normal Form.** The TPTP CNF syntax provides an untyped classical first-order logic with equality, which closely corresponds to the internal representation used in most automatic provers. It is parameterized by sets of *function symbols* ($\mathsf{f}$, $\mathsf{g}$, ...), *predicate symbols* ($\mathsf{p}$, $\mathsf{q}$, ...), and *variables* ($X$, $Y$, ...). Function and predicate symbols are assigned fixed arities. The *terms*, *atoms*, *literals*, and *clauses* of CNF have the following syntaxes.[1]

| Terms: | | Atoms: | |
|---|---|---|---|
| $t ::= X$ | variable | $a ::= \mathsf{p}(t_1, \ldots, t_n)$ | predicate atom |
| $\mid \mathsf{f}(t_1, \ldots, t_n)$ | function term | $\mid t_1 = t_2$ | equality |
| Literals: | | Clauses: | |
| $l ::= a$ | positive literal | $c ::= l_1 \vee \cdots \vee l_n$ | |
| $\mid \neg\, a$ | negative literal | | |

Variable names start with an upper-case letter, to distinguish them from function and predicate symbols. Variables are interpreted universally in a clause. Nullary function symbols $\mathsf{c}$ are called *constants* and are normally written without parentheses in terms—i.e., $\mathsf{c}$ rather than $\mathsf{c}()$. The negative literal $\neg\, t = u$ is called *disequality* and is written $t \neq u$.

A CNF problem is a set of clauses, where the conjecture is negated and the axioms appear as is. An interpretation of the function and predicate symbols is a model of a set of clauses iff all ground instances of all clauses evaluate to $\top$. The aim of an automatic prover is to derive the empty clause ($\bot$) from the problem's clauses using some appropriate calculus (typically based on resolution, tableaux, or instantiation) or, equivalently, to show that the problem is unsatisfiable. In case of success, the conjecture is provable from the axioms.

**FOF: First-Order Form.** The FOF syntax extends CNF with the familiar connectives ($\neg$, $\vee$, $\wedge$, $\longrightarrow$, $\longleftrightarrow$) and quantifiers ($\forall$, $\exists$), which can be freely nested. Popular automatic provers such as E, SPASS, and Vampire include clausifiers that translate FOF problems into CNF, introducing Skolem functions to encode existential variables. For example, the FOF formula

$$\forall N.\ N = \mathsf{zero} \vee \exists M.\ N = \mathsf{suc}(M)$$

would be translated into the CNF clause

$$N = \mathsf{zero} \vee N = \mathsf{suc}(\mathsf{m}(N))$$

where $\mathsf{m}$ is a Skolem function.

[1] We take some liberties with the concrete TPTP syntaxes. In particular, we prefer traditional notations ($\neg$, $\vee$, $\neq$, ...) to ASCII symbols (`~`, `|`, `!=`, ...).

**TFF0: Core Typed First-Order Form.** The typed first-order form (TFF) is a recent addition to the TPTP family. The core syntax TFF0 extends FOF with simple monomorphic types (sorts) and interpreted arithmetic. Each $n$-ary function symbol $\mathsf{f}$ and predicate symbol $\mathsf{p}$ must be declared with a type signature:

$$\mathsf{f} : \sigma_1 \times \cdots \times \sigma_n \to \sigma \qquad\qquad \mathsf{p} : \sigma_1 \times \cdots \times \sigma_n \to o$$

The pseudo-type $o$ of Booleans is used for the result of predicates. The type $\iota$ of individuals is predefined but has no particular semantics. An exhaustion rule for (uninterpreted) natural numbers is presented below:

$$\begin{array}{l} \mathsf{zero} : \mathit{nat} \\ \mathsf{suc} : \mathit{nat} \to \mathit{nat} \\ \forall N^{\mathit{nat}}.\ N = \mathsf{zero} \lor \exists M^{\mathit{nat}}.\ N = \mathsf{suc}(M) \end{array}$$

**TFF1: Typed First-Order Form with Polymorphism.** TFF1 is an extension of TFF0 with rank-1 polymorphism. Types are either type variables $\alpha$ or applied type constructors $\bar{\alpha}\ \kappa$. Type declarations are of the forms

$$\mathsf{f} : \forall\alpha_1 \ldots \alpha_m.\ \sigma_1 \times \cdots \times \sigma_n \to \sigma \qquad \mathsf{p} : \forall\alpha_1 \ldots \alpha_m.\ \sigma_1 \times \cdots \times \sigma_n \to o$$

The symbols $\times$, $\to$, and $o$ are part of the type signature syntax; they are not type constructors. Uses of $\mathsf{f}$ and $\mathsf{p}$ in terms require $m$ type arguments specified in angle brackets and $n$ term arguments specified in parentheses:

$$\mathsf{f}\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, t_n) \qquad\qquad \mathsf{p}\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, t_n)$$

The type arguments completely determine the type of the arguments and, for functions, of the result. In a departure from the official TFF1 syntax, we omit the type arguments if they are irrelevant or can be deduced from the context; in addition, we sometimes use superscripts to indicate type constraints.

**THF0: Core Typed Higher-Order Form.** The THF0 syntax is an effective superset of TFF0 similar to Isabelle's higher-order logic but without polymorphism or type classes. The types of THF0 are those of TFF0 extended with a binary type constructor $\to$ that can be arbitrarily nested, to allow curried functions and functions with higher-order arguments, and $o$ is a first-class type. Function application is performed via an explicit left-associative application operator @; thus, THF0 syntactically requires $\mathsf{f} \mathbin{@} X \mathbin{@} Y$ where the first-order TPTP syntaxes have $\mathsf{f}(X, Y)$ and Isabelle/HOL has $\mathsf{f}\ x\ y$. The exhaustion rule for *nat* becomes

$$\begin{array}{l} \mathsf{zero} : \mathit{nat} \\ \mathsf{suc} : \mathit{nat} \to \mathit{nat} \\ \forall N^{\mathit{nat}}.\ N = \mathsf{zero} \lor \exists M^{\mathit{nat}}.\ N = \mathsf{suc} \mathbin{@} M \end{array}$$

## 6.2 Sledgehammer and Metis

Sledgehammer's main processing steps consist of relevance filtering, translation to untyped first-order logic, ATP invocation, proof reconstruction, and proof minimization. For proof reconstruction, it is assisted by the *metis* proof method [92,153].

**Relevance Filtering.** Sledgehammer employs a simple relevance filter [126] to extract a few hundred lemmas that seem relevant to the current conjecture from Isabelle's enormous libraries. The relevance test is based on how many symbols are shared between the conjecture and each candidate lemma. Although crude, this filter greatly improves Sledgehammer's success rate, because most ATPs perform poorly in the presence of thousands of axioms.

**Translation into First-Order Logic.** HOL is much richer than the ATPs' untyped first-order logics (CNF and FOF). Sledgehammer uses various techniques to translate higher-order formulas to first-order logic [125]. The translation is lightweight but unsound; the ATP proofs can be trusted only after they have been reconstructed in Isabelle. Higher-order features complicate the translation: $\lambda$-abstractions are rewritten to combinators, and curried functions are passed varying numbers of arguments by means of an explicit application operator.

**Parallel ATP Invocation.** For a number of years, Isabelle has emphasized parallelism to exploit modern multi-core architectures [203]. Accordingly, Sledgehammer invokes several ATPs in parallel, with great success: Running E [167], SPASS [200], and Vampire [163] together for five seconds solves as many problems as running a single automatic prover for two minutes [44, §8]. The automatic provers are run in the background so that users can keep working during the proof search, although most users find it hard to think while automatic provers are active and prefer to wait for the responses.

**Proof Reconstruction.** As in other LCF-style theorem provers [83], Isabelle theorems can only be generated within a small inference kernel. It is possible to bypass this safety mechanism if some external tool is to be trusted as an oracle, but all oracle inferences are tracked. Sledgehammer performs true proof reconstruction by running Metis, a first-order resolution prover written in ML by Joe Hurd [92]. Although Metis was designed to be interfaced with an LCF-style interactive theorem prover (HOL4), integrating it with Isabelle's inference kernel required significant effort [153]. The resulting *metis* proof method is given the short list of facts referenced in the proof found by the external ATP.[1] The *metis* call is all that remains from the Sledgehammer invocation in the Isabelle theory, which can then be replayed without external provers. Since *metis* is given only a handful of facts, it often succeeds almost instantly.

**Proof Minimization.** Reconstruction using *metis* loses about 5% of ATP proofs because Metis takes too long. One reason is that automatic provers frequently use many more facts than are necessary, making it harder for *metis* to re-find the proof. The minimization tool takes a set of facts found by a prover and repeatedly calls the prover with subsets of the facts to find a minimal set. Depending on the number of initial facts, it relies on either of these two algorithms:

- The naive linear algorithm attempts to remove one fact at a time. This can require as many prover invocations as there are facts in the initial set.

[1] To avoid confusion between the Isabelle proof method and the underlying first-order prover, we consistently write *metis* for the former and Metis for the latter.

- The binary algorithm, due to Bradley and Manna [47, §4.3], recursively bisects the facts. It performs best when a small fraction of the facts are actually required [44, §7].

**Example 6.1.** In the Isabelle proof below, taken from a formalization of the Robbins conjecture [196], four of the five subproofs are discharged by a *metis* call generated by Sledgehammer using an ATP:

```
proof –
  let $z = -(x \sqcup -y)$ and $ky = y \sqcup k \otimes (x \sqcup z)$
  have $-(x \sqcup -ky) = z$ by ($simp\ add$: $copyp0$)
  hence $-(-ky \sqcup -(-y \sqcup z)) = z$ by ($metis\ assms\ sup\_comm$)
  also have $-(z \sqcup -ky) = x$ by ($metis\ assms\ copyp0\ sup\_comm$)
  hence $z = -(-y \sqcup -(-ky \sqcup z))$ by ($metis\ sup\_comm$)
  finally show $-(y \sqcup k \otimes (x \sqcup -(x \sqcup -y))) = -y$ by ($metis\ eq\_intro$)
qed
```

The example is typical of the way Isabelle users employ the tool: If they understand the problem well enough to propose some intermediate properties, all they need to do is state a progression of properties in small enough steps and let Sledgehammer or an automatic Isabelle tactic prove each one.

## 6.3 Extension with SMT Solvers

It is widely recognized that combining automated reasoning systems of different types can deliver huge rewards. First-order ATPs are powerful and general, but they can usefully be complemented by other technologies. Satisfiability modulo theories (SMT) is a powerful technology based on combining a SAT solver with decision procedures for first-order theories, such as equality, integer and real arithmetic, and bit-vector reasoning. SMT solvers are particularly well suited to discharging large proof obligations arising from program verification. Although SMT solvers are automatic theorem provers in a general sense, we find it convenient to reserve the abbreviation ATP for more traditional systems, especially resolution and tableau provers.[1]

In previous work, Böhme integrated the SMT solvers CVC3 [12], Yices [71] and Z3 [66] with Isabelle as oracles and implemented step-by-step proof reconstruction for Z3 [41, 42, 45]. The resulting *smt* proof method takes a list of problem-specific facts that are passed to the SMT solver along with the conjecture (Section 6.3.1). While a motivated user can go a long way with the *smt* proof method [43], the need to specify facts and to guess that a conjecture could be solved by SMT makes it hard to use. As evidence of this, the Isabelle theories accepted in the *Archive of Formal Proofs* [102] in 2010 and early 2011, after the introduction of *smt*, contain 7958 calls to Isabelle's simplifier, 928 calls to its tableau prover, 219 calls to *metis* (virtually all generated using Sledgehammer), but not even one *smt* call.

[1] This distinction is mostly historical. The TPTP and SMT communities are rapidly converging [183, §1], with more and more ATPs supporting typical SMT features such as arithmetic and sorts, and SMT solvers parsing TPTP syntaxes. There is also a strong technological connection between TPTP instantiation-based provers (such as Equinox [57] and iProver [104]) and SMT solvers.

Can typical Isabelle users benefit from SMT solvers? We assumed so and took the obvious next step, namely to have Sledgehammer run SMT solvers in parallel with ATPs, reusing the existing relevance filter and parallel architecture (Section 6.3). This idea seemed promising for a number of reasons:

- ATPs and SMT solvers have complementary strengths. The former handle quantifiers more elegantly; the latter excel on large, mostly ground problems.
- The translation of higher-order constructs and types is done differently for the SMT solvers than for the ATPs—differences that should result in more proved goals.[1]
- Users should not have to guess whether a problem is more appropriate for ATPs or SMT solvers. Both classes of prover should be run concurrently.

The Sledgehammer–SMT integration is, to our knowledge, the first of its kind, and we had no clear idea of how successful it would be as we started the implementation work. Would the SMT solvers only prove conjectures already provable using the ATPs, or would they find original proofs? Would the decision procedures be pertinent to typical interactive goals? Would the SMT solvers scale in the face of hundreds of quantified facts translated en masse, as opposed to carefully crafted axiomatizations?

We were pleased to find that the SMT solvers are up to the task and add considerable power to Sledgehammer. The SMT integration is joint work with Sascha Böhme and Lawrence Paulson [28].

### 6.3.1 The *smt* Proof Method

The *smt* proof method [41,42,45] developed by Böhme interfaces Isabelle with SMT solvers. It translates the conjecture and any user-supplied facts to the SMT solvers' many-sorted first-order logic, invokes a solver, and (depending on the solver) either trusts the result or attempts to reconstruct the proof in Isabelle.

**Translation into First-Order Logic.** The translation maps equality and arithmetic operators to the corresponding SMT-LIB 1.2 [160] concepts.[2] SMT-LIB is a standard syntax supported by most SMT solvers; it corresponds roughly to the monomorphic typed first-order form (TFF0) of TPTP. Many-sorted first-order logic's support for sorts would seem to make it more appropriate to encode HOL type information than untyped first-order logic, but it does not support polymorphism. The solution is to monomorphize the formulas: Polymorphic formulas are iteratively instantiated with relevant monomorphic instances of their polymorphic constants. This process is iterated a bounded number of times to obtain the monomorphized problem. Partial applications are translated using an explicit application operator. In contrast to Sledgehammer's combinator approach, the *smt* method lifts $\lambda$-abstractions into new "supercombinators" [90].

[1] There are also many efficiency, readability, and robustness advantages of obtaining several proofs for the same goal from different sources [182].

[2] Support for SMT-LIB 2.0 [11] is future work.

**Proof Reconstruction.** CVC3 and Z3 provide independently checkable proofs of unsatisfiability. Proof reconstruction is currently supported for Z3, whereas CVC3 and Yices can be invoked as oracles. Reconstruction relies extensively on standard Isabelle proof methods such as the simplifier [134], the tableau prover [147], and the arithmetic decision procedures [53]. Certificates make it possible to store Z3 proofs alongside Isabelle theories, allowing proof replay without Z3; only if the theories change must the certificates be regenerated. Using SMT solvers as oracles requires trusting both the solvers and the *smt* method's translation, so it is generally frowned upon.

**Example 6.2.** The integer recurrence relation $x_{i+2} = |x_{i+1}| - x_i$ has period 9. This property can be proved using the *smt* method as follows [41, §1.2]:

**lemma** $x_3 = |x_2| - x_1 \;\wedge\; x_4 = |x_3| - x_2 \;\wedge\; x_5 = |x_4| - x_3 \;\wedge$
$x_6 = |x_5| - x_4 \;\wedge\; x_7 = |x_6| - x_5 \;\wedge\; x_8 = |x_7| - x_6 \;\wedge$
$x_9 = |x_8| - x_7 \;\wedge\; x_{10} = |x_9| - x_8 \;\wedge\; x_{11} = |x_{10}| - x_9 \Longrightarrow$
$x_1 = x_{10} \;\wedge\; x_2 = x_{11}$
**by** *smt*

SMT solvers find a proof almost instantly, and proof reconstruction (if enabled) takes a few seconds. In contrast, Isabelle's arithmetic decision procedure requires several minutes to establish the same result. This example does not require any problem-specific facts, but these would have been supplied as arguments in the *smt* call just like for *metis* in Example 6.1.

### 6.3.2 Solver Invocation

Extending Sledgehammer with SMT solvers was largely a matter of connecting existing components: Sledgehammer's relevance filter and minimizer with the *smt* method's translation and proof reconstruction. Figure 6.1 depicts the resulting architecture, omitting proof reconstruction and minimization.

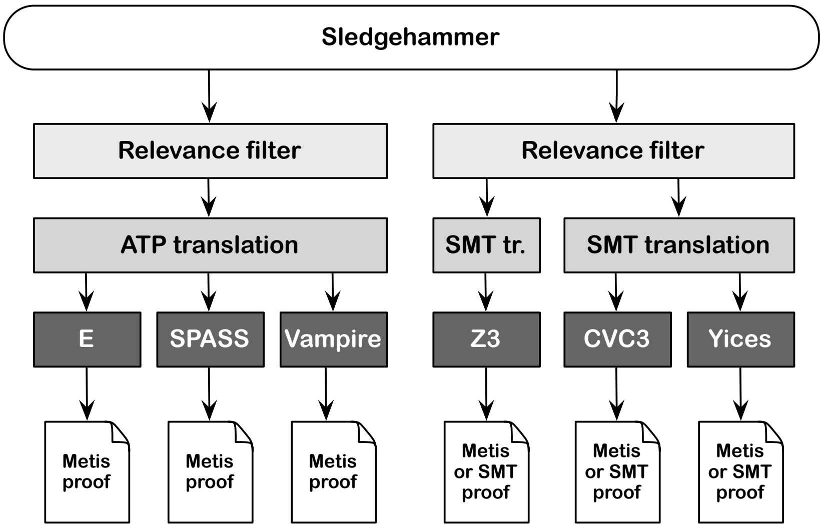


**Figure 6.1:** Sledgehammer's extended architecture

Two instances of the relevance filter run in parallel, to account for different sets of built-in symbols. The relevant facts and the conjecture are translated to the ATP or SMT version of first-order logic, and the resulting problems are passed to the provers. The translation for Z3 is done slightly differently than for CVC3 and Yices to take advantage of the former's support for nonlinear arithmetic.

In our first experiments, we simply invoked Z3 as an oracle with the monomorphized relevant facts, using the same translation as for the *smt* proof method. The results were disappointing. Several factors were to blame:

- The translation of hundreds of facts took many seconds.
- It took us a while to get the bugs out of our translation code. Syntax errors in many generated problems caused Z3 to give up immediately.
- Z3 often ran out of memory after a few seconds or, worse, crashed.

Latent issues both in our translation and in Z3 were magnified by the number of facts involved. Earlier experience with SMT solvers had involved only a few facts.

The bottleneck in the translation was monomorphization. Iterative expansion of a few hundred HOL formulas yielded thousands of monomorphic instances. We reduced the maximum number of iterations from 10 to 3, to great effect.

The syntax errors were typically caused by confusion between formulas and terms or the use of a partially applied built-in symbol (both of which are legal in HOL). These were bugs in the *smt* proof method; we gradually eradicated them.

We reported the segmentation faults to the Z3 developers, who released an improved version. The bug was located in Z3's proof generation facility, which is disabled by default and hence not as well tested as the rest of the solver. To handle the frequent out-of-memory conditions, we modified Sledgehammer to retry aborted solver calls with half the facts. This simple change was enough to increase the success rate dramatically.

### 6.3.3 Proof Reconstruction

In case of success, Sledgehammer extracts the facts referenced in the SMT proof—the unsatisfiable core—and generates an *smt* call with these facts supplied as arguments. For example:

**by** (*smt assms copyp0 sup_comm*)

The proof method invokes Z3 to re-find the proof and replays it step by step. The Z3 proof can also be stored alongside the Isabelle theory as a certificate to avoid invoking Z3 each time the proof is rechecked. Proof minimization can be done as for ATP proofs to reduce the number of facts.

To increase the success rate and reduce the dependency on external solvers or certificates, Sledgehammer first tries *metis* for a few seconds. If this succeeds, Sledgehammer generates a *metis* call rather than an *smt* call. This will of course fail if the proof requires theories other than equality.

One of the less academically rewarding aspects of integrating third-party tools is the effort spent on solving mundane issues. Obtaining an unsatisfiable core from the SMT solvers turned out to be surprisingly difficult:

- CVC3 returns a full proof, but somehow the proof refers to all facts, whether they are actually needed or not, and there is no easy way to find out which facts are really needed. We rely on Sledgehammer's proof minimizer and its binary algorithm to reduce the facts to a reasonable number.
- Yices can output a minimal core, but for technical reasons only when its native input syntax is used rather than the standard SMT-LIB 1.2 format. We tried using off-the-shelf file format converters to translate SMT-LIB 1.2 to 2 then to Yices, but this repeatedly crashed. In the end, we settled for the same solution as for CVC3.
- For Z3, we could reuse our existing proof parser, which we need to reconstruct proofs. The proof format is fairly stable, although new releases often come with various minor changes.

### 6.3.4 Relevance Filtering

The relevance filter gives more precise results if it ignores HOL constants that are translated to built-in constructs. For ATPs, this concerns equality, connectives, and quantifiers. SMT solvers support a much larger set of built-in constructs, notably arithmetic operators. It was straightforward to generalize the filter code so that it performs its task appropriately for SMT solvers. As a result, a fact such as $1 + 1 = 2^{int}$ might be considered relevant for the ATPs but will always be left out of SMT problems, since it involves only built-in symbols. (Indeed, the fact is a tautology of linear arithmetic.)

Observing that some provers cope better with large fact bases than others, we optimized the maximum number of relevant facts to include in a problem independently for each prover. The maxima we obtained are 150 for CVC3 and Yices and 250 for Z3, which is comparable to the corresponding figures for the ATPs.

### 6.3.5 Example

A gratifying example arose on the Isabelle mailing list [138] barely one week after we had enabled SMT solvers in the development version of Sledgehammer. A novice was experimenting with a simple arithmetic datatype:

**datatype** $arith = \mathsf{Z} \mid \mathsf{Succ}\ arith \mid \mathsf{Pred}\ arith$

**inductive** $\mathsf{isvalue}^{arith \to o}$ **where**
$\mathsf{isvalue}\ \mathsf{Z}$
$\mathsf{isvalue}\ M \Longrightarrow \mathsf{isvalue}\ (\mathsf{Succ}\ M)$

**inductive** $\mathsf{step}^{arith \to arith \to o}$ **where**
*s_succ*: $\mathsf{step}\ m\ m' \Longrightarrow \mathsf{step}\ (\mathsf{Succ}\ m)\ (\mathsf{Succ}\ m')$
*s_pred_zero*: $\mathsf{step}\ (\mathsf{Pred}\ \mathsf{Z})\ \mathsf{Z}$
*s_pred*: $\mathsf{step}\ m\ m' \Longrightarrow \mathsf{step}\ (\mathsf{Pred}\ m)\ (\mathsf{Pred}\ m')$
*s_pred_succ*: $\mathsf{isvalue}\ v \Longrightarrow \mathsf{step}\ (\mathsf{Pred}\ (\mathsf{Succ}\ v))\ v$

He wanted to prove the following property but did not know how to proceed:

```
lemma step (Pred Z) m ⟹ m = Z
```

Tobias Nipkow helpfully supplied a structured Isar proof:

```
lemma
  assumes step (Pred Z) m
  shows m = Z
using assms
proof cases
  case s_pred_zero thus m = Z by simp
next
  case (s_pred m′)
  from ‘step Z m′’ have False by cases
  thus m = Z  by blast
qed
```

The proof is fairly simple by interactive proving standards, but it nonetheless represents a few minutes' work to a seasoned user (and, as we saw, was too difficult for a novice). Nipkow then tried the development version of Sledgehammer and found a much shorter proof due to Z3:

```
by (smt arith.simps(2,4,5,8) step.simps)
```

Although it involves no theory reasoning beyond equality, the ATPs failed to find it within 30 seconds because of the presence of too many extraneous facts. The evaluation in Section 6.7 confirms that this is no fluke: SMT solvers often outperform the ATPs even on nonarithmetic problems.

## 6.4 Elimination of Higher-Order Features

A central component of Sledgehammer is its translation module, which encodes Isabelle/HOL formulas in the untyped first-order logic understood by the resolution provers E, SPASS, Vampire, and Metis.[1] Meng and Paulson [125] designed the translation at the heart of Sledgehammer, inspired by Hurd's work in the context of HOL98 and HOL4 [91, 92].

It is convenient to view the translation as a two-step process:

1. The higher-order features of the problem are eliminated, yielding a first-order problem with polymorphic types and type classes: $\lambda$-abstractions are rewritten to combinators, and curried functions are passed varying numbers of arguments via an explicit application operator, hAPP. This step is described in this section.

2. The type information is encoded in the target logic. Meng and Paulson had the TPTP CNF format as their target; our main target is FOF, but we are also interested in TFF0, TFF1, and even THF0. This step is described in Section 6.5.

[1] The translation to the SMT-LIB format also plays an important role, but it lies within the scope of Böhme's Ph.D. thesis [41].

The higher-order features to eliminate are listed below, with an example of each.

1. Higher-order quantification: $\forall x.\, \exists f.\, f\ x = x$
2. Higher-order arguments: $\mathsf{map}\ f\ [x] = [f\ x]$
3. Partial function applications: $f = g \longrightarrow f\ x = g\ x$
4. $\lambda$-abstractions: $(\lambda x\, y.\, y + x) = (+)$
5. Formulas within terms: $\mathsf{p}\ (x^a = x) \longrightarrow \mathsf{p}\ \mathsf{True}$
6. Terms as formulas: $x^o$

After examining several schemes, Meng and Paulson adopted a translation that eliminates higher-order features locally, yielding a smooth transition from purely first-order to heavily higher-order problems. To give a flavor of the translation, here are the six formulas above after their higher-order features have been eliminated, expressed in a TFF1-like syntax:

1. $\forall X^{\alpha}.\, \exists F^{(\alpha,\alpha)\,fun}.\ \mathsf{hAPP}(F,\, X) = X$
2. $\mathsf{map}(F,\, \mathsf{cons}(X, \mathsf{nil})) = \mathsf{cons}(\mathsf{hAPP}(F, X),\, \mathsf{nil})$
3. $F = G \longrightarrow \mathsf{hAPP}(F,\, X) = \mathsf{hAPP}(G,\, X)$
4. $\mathsf{c}(\mathsf{plus}) = \mathsf{plus}$
5. $\mathsf{p}(\mathsf{fequal}(X^a,\, X)) \longrightarrow \mathsf{p}(\mathsf{true})$
6. $\mathsf{hBOOL}(X)$

In these and later examples, we rely on the reader's discernment to identify $+$ in Isabelle/HOL with plus in first-order logic, [] with nil, Cons with cons, True with true, $x$ with $X$, and so on. In the rest of this section, we expand on Meng and Paulson's translation and discuss some alternatives.

### 6.4.1 Arguments and Predicates

Following common practice, Meng and Paulson enforce the first-order distinction between terms and formulas by introducing two special symbols, which they call @ and B in their paper [125, §2.1] and hAPP and hBOOL in their implementation.

- The hAPP function symbol serves as an explicit application operator; it takes a (curried) function and an argument and applies the former on the latter. It permits the application of a variable: $F(\mathsf{c})$ is illegal in first-order logic, but $\mathsf{hAPP}(F,\, \mathsf{c})$ is legal. It also makes it possible to pass a variable number of arguments to a function, as is often necessary if the problem requires partial function applications.
- The hBOOL symbol is a predicate that yields true iff its Boolean term argument equals True. Intuitively, $\mathsf{hBOOL}(t)$ is the same as $t = \mathsf{true}$, where true is the uninterpreted constant symbol corresponding to the HOL constant True.

The special symbols hAPP and hBOOL can hugely burden problems if introduced systematically for all arguments and predicates. To reduce clutter, Meng and Paulson compute the minimum arity $n$ needed for each symbol and pass the first $n$ arguments directly, falling back on hAPP for additional arguments. This optimization works fairly well and has long been the default mode of operation, but it sometimes makes problems unprovable. For example, from the HOL lemmas $\mathsf{Suc}\ 0 = 1$ and $\mathsf{map}\ f\ [x] = [f\ x]$ we would expect the conjecture

$$\exists g^{nat \to nat}.\ \mathsf{map}\ g\ [0] = [1]$$

to be provable, by taking Suc as witness for $g$. However, the first-order problem

$\mathsf{suc}(\mathsf{zero}) = \mathsf{one}$
$\mathsf{map}(F,\ \mathsf{cons}(X,\ \mathsf{nil})) = \mathsf{cons}(\mathsf{hAPP}(F,\ X),\ \mathsf{nil})$
$\mathsf{map}(G,\ \mathsf{cons}(\mathsf{zero},\ \mathsf{nil})) \neq \mathsf{cons}(\mathsf{one},\ \mathsf{nil})$

(where the last clause is the negated conjecture) is unprovable. Confusingly, if we supply an additional fact where Suc appears without argument, say, $\mathsf{Suc} \neq \mathsf{id}$, Suc is translated to a nullary function symbol unifiable with $F$ and $G$, and the problem becomes provable, even though the additional fact is not referenced in the proof.

This incompleteness meant that Sledgehammer would sometimes miss proofs. The same issue affected the translation to SMT-LIB [41, §2.2.3]. A different trade-off was made for *metis* and the proof minimizer: In problems that fell outside a first-order fragment of HOL, hAPP and hBOOL were systematically introduced to ensure that proofs found by an ATP are not lost because of this subtle issue.

Since the issue is connected to higher-order quantifiers, which rarely occur in practice, a better compromise is possible: We can look for essentially universal variables (i.e., non-skolemizable variables) having a function type $\sigma$ in the problem and require each constant of type $\tau_1 \to \cdots \to \tau_n \to \sigma\rho$, where $\sigma\rho$ is an instance of $\sigma$, to take at most their first $n$ arguments directly, using hAPP for any additional arguments. This can burden the translation, but the gain in reliability and predictability is such that we have now made this the preferred mode of operation.

A similar issue affects predicates. Meng and Paulson map HOL functions of type $\sigma \to o$ to predicate symbols if possible and rely on the hBOOL predicate to turn the remaining Boolean terms into formulas. From $\mathsf{null}\ []$, we would expect $\exists p^{\alpha\ list \to o}.\ p\ []$ to be derivable, but this is not the case if null is translated to a predicate symbol. If we make null a nullary function symbol instead, the problem becomes

$\mathsf{hBOOL}(\mathsf{hAPP}(\mathsf{null},\ \mathsf{nil}))$
$\neg\ \mathsf{hBOOL}(\mathsf{hAPP}(P,\ \mathsf{nil}))$

and it can be proved by resolving the two clauses together (with $P = \mathsf{null}$). The general remedy is simple: If the problem contains a universal variable ranging over $\sigma_1 \to \cdots \to \sigma_m \to o$ and a HOL constant of type $\tau_1 \to \cdots \to \tau_n \to \sigma_1\rho \to \cdots \to \sigma_m\rho \to o$, where the types $\sigma_i\rho$ are instances of the corresponding types $\sigma_i$ with respect to the same substitution $\rho$, the constant should be translated to a function symbol of arity $n$ or less, with hAPP and hBOOL inserted as needed, rather than to a predicate.

Although the examples above are artificial, the issues they illustrate arise in real-world examples. As observed elsewhere [152, §5], Sledgehammer's performance on higher-order problems is unimpressive; as a first step in the right direction, we should at the very least make sure that easy higher-order problems are still provable when they reach the first-order provers.

### 6.4.2 Translation of $\lambda$-Abstractions

Meng and Paulson evaluated three schemes for eliminating $\lambda$-abstractions: $\lambda$-lifting [90], the Curry combinators ($\mathsf{I}$, $\mathsf{K}$, $\mathsf{S}$, $\mathsf{B}$, $\mathsf{C}$) [190], and the modified Turner combinators (Curry plus $\mathsf{S}'$, $\mathsf{B}^*$, $\mathsf{C}'$) [97, §16.2; 190]. The combinator approaches enable the automatic provers to synthesize $\lambda$-terms (an extreme example of which is given in Section 6.5.1) but tend to yield bulkier formulas than $\lambda$-lifting: The translation is quadratic in the worst case, and the equational definitions of the combinators are very prolific in the context of resolution. In spite of this, Meng and Paulson found the Curry combinators to be superior to $\lambda$-lifting and the Turner combinators on their benchmarks and kept them as the only supported translation scheme in Sledgehammer and *metis*.

Following the success of the *smt* method, based on $\lambda$-lifting, we carried out new experiments with the tried and tested *smt* implementation of $\lambda$-lifting [41, §2.2.2]. In addition, we introduced a hybrid scheme that characterizes each $\lambda$-lifted constant both using a lifted equation $\mathsf{c}\ x_1\ \ldots\ x_n = t$ and via Curry combinators. The different schemes are evaluated in Section 6.7.3.

### 6.4.3 Higher-Order Reasoning

Beyond the translation of $\lambda$-abstractions, we looked into several ways to improve Sledgehammer's higher-order reasoning capabilities. These are presented below. An alternative to this mixed bag of techniques is to integrate genuine higher-order ATPs as backends; this is considered in Section 6.5.3.

**Extensionality.** The relevance filter previously left out the extensionality axiom

$$(\textstyle\bigwedge x.\ f\ x = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x)$$

because it involves only logical constants, which are ignored for the relevance calculations. However, in its translated form

$$\exists X.\ \mathsf{hAPP}(F,\ X) = \mathsf{hAPP}(G,\ X) \longrightarrow F = G$$

extensionality characterizes the otherwise unspecified $\mathsf{hAPP}$ operator. The axiom is now included in problems that use $\mathsf{hAPP}$.

**Proxies.** First-order occurrences of logical constants can be translated to the corresponding TPTP constructs. The remaining occurrences, such as $=$ and $\mathsf{True}$ in

$$\mathsf{p}\ (x^a = x) \longrightarrow \mathsf{p}\ \mathsf{True}$$

must be translated somehow. The standard solution is to treat higher-order occurrences of Booleans the same way as we would for any other HOL type, mapping $o$ to an uninterpreted (deeply embedded) type *bool*, $\mathsf{True}$ to an uninterpreted constant $\mathsf{true}$, and $\mathsf{False}$ to an uninterpreted constant $\mathsf{false}$. Connectives, quantifiers, and equality can be embedded in the same way. The uninterpreted function symbols that represent the logical constants inside the term language are called *proxies*. By axiomatizing the proxies, we permit a restricted form of higher-order reasoning.

Meng and Paulson handle higher-order occurrences of equality in this way [125, §2.1], by introducing the proxy $\mathsf{fequal}$ (where $\mathsf{f}$ highlights that it is a function and not a predicate) axiomatized as

$$\mathsf{hBOOL}(\mathsf{hAPP}(\mathsf{hAPP}(\mathsf{fequal},\ X^{\alpha}),\ Y)) \longleftrightarrow X = Y$$

Being a nullary function, fequal can be used in any context, by introducing hAPP and hBOOL as appropriate.

We have now extended this approach to the connectives and partially to the ∀ and ∃ quantifiers.[1] For True and False, we include the axioms hBOOL(true) and ¬ hBOOL(false), which connect terms to formulas. If a sound type encoding is used (Section 6.5), we also include the exhaustion rule $X^{bool} = \mathsf{true} \vee X = \mathsf{false}$.

The first-order problem produced for the HOL conjecture $\mathsf{p}\ (x^{a} = x) \longrightarrow \mathsf{p}\ \mathsf{True}$ is shown below:

$$\begin{array}{l}\neg\ \mathsf{hBOOL}(\mathsf{false})\\ \mathsf{hBOOL}(\mathsf{true})\\ X^{bool} = \mathsf{true} \vee X = \mathsf{false}\\ \mathsf{hBOOL}(\mathsf{hAPP}(\mathsf{hAPP}(\mathsf{fequal},\ X^{\alpha}),\ Y^{\alpha})) \longleftrightarrow X = Y\\ \exists X^{a}.\ \mathsf{p}(\mathsf{hAPP}(\mathsf{hAPP}(\mathsf{fequal},\ X),\ X)) \wedge \neg\ \mathsf{p}(\mathsf{true})\end{array}$$

Thanks to the proxies, the problem is provable.

**Induction.** The relevance filter normally leaves out induction rules, since it is unrealistic (if possible at all) for the ATPs to instantiate them with the proper terms, using combinators and proxies to encode formulas. An alternative is to have the filter preinstantiate induction rules based on the conjecture, in the hope that a straightforward induction (requiring no generalization) is possible. In the following example, from a formalization of Huffman's algorithm [26], the proof was discovered by an ATP:

**lemma** finite (alphabet $t$)
**by** (*metis tree.induct* [where $P = (\lambda t.\ \mathsf{finite}\ (\mathsf{alphabet}\ t))$]
*alphabet.simps*(1) *alphabet.simps*(2) *finite.simps finite_UnI*)

This technique is well-known [168, §7.2.1]. Our experiments have so far been inconclusive, because most of the goals that Sledgehammer could prove this way could be proved much more simply with the *induct* proof method followed by *auto*.

## 6.5 Encoding of Polymorphic Types

After translating away the higher-order features of a problem, we are left with first-order formulas in which Isabelle's rich type information, including polymorphism, overloading, and axiomatic type classes, is still present. In contrast, most ATPs support only untyped or monomorphic (many-sorted) formalisms.

The various sound and complete translation schemes for polymorphic types proposed in the literature produce their share of clutter, and lighter approaches are usually unsound (i.e., they do not preserve satisfiability). As a result, application

[1] A mundane issue related to skolemization in *metis* stands in the way of a complete characterization of ∀ and ∃.

authors face a difficult choice between soundness and efficiency. In the context of Sledgehammer, Meng and Paulson [125] considered two main schemes for translating types. Briefly:

- The *fully typed* translation tags every term and subterm with its type using a binary function symbol. Types are represented as first-order terms, with type variables coded as term variables.
- The *constant-typed* translation passes explicit type arguments to the function and predicate symbols corresponding to HOL constants to enforce correct type class reasoning and overload resolution, but not to prevent ill-typed variable instantiations. As a result, it is unsound.

We describe these translations and a few others in more detail in Section 6.5.1. Since the constant-typed translation results in a much higher success rate than the fully typed one [44, §4.2; 125, §3], Meng and Paulson made it the default despite its unsoundness. This state of affairs is unsatisfactory:

- Finite exhaustion rules must be left out because they lead to unsound cardinality reasoning [125, §2.8]. The inability to encode such rules prevents the discovery of proofs by case analysis on finite types. This limitation affects mildly higher-order problems requiring the axiom $x = \mathsf{True} \lor x = \mathsf{False}$, such as the conjecture $P\ \mathsf{True} \Longrightarrow P\ \mathsf{False} \Longrightarrow P\ x$.
- Spurious proofs are distracting and sometimes conceal sound proofs. The seasoned user eventually learns to recognize facts that lead to unsound reasoning and mark them with the *no_atp* attribute to remove them from the scope of the relevance filter, but this remains a stumbling block for the novice.
- It would be desirable to let the ATPs themselves perform relevance filtering, or even use a sophisticated system based on machine learning, such as MaLARea [191, 192], where successful proofs guide subsequent ones. However, such approaches tend to quickly discover and exploit inconsistencies in the large translated axiom set.

This section presents joint work with Sascha Böhme and Nicholas Smallbone that extends earlier work by Claessen, Lillieström, and Smallbone [61]. Claessen et al. designed a family of sound, complete, and efficient translations from monomorphic to untyped first-order logic. The key insight is that monotonic types (types whose domain can always be augmented with new elements while preserving satisfiability) can be simply erased, while the remaining types can be made monotonic by introducing guards (predicates) or tags (functions). Although undecidable, monotonicity can often be inferred using suitable calculi, as we saw in Chapter 4.

In this section, we first generalize this approach to an ML-style polymorphic first-order logic, as embodied by the polymorphic TPTP typed first-order form (TFF1). Unfortunately, the presence of a single polymorphic literal of the form $X^{\alpha} = t$ will lead us to classify every type as potentially nonmonotonic and force the use of guards or tags everywhere, as in the traditional encodings. We solve this issue by our second main contribution, a novel scheme that considerably reduces the clutter associated with nonmonotonic types, based on the observation that guards or tags are only required when translating the particular axioms that make a type nonmonotonic. Consider this simple TFF0 specification of a two-valued *state* type:

$$S^{state} = \mathsf{on} \lor S = \mathsf{off}$$
$$\mathsf{toggle}(S^{state}) \neq S$$

Claessen et al. would classify *state* as nonmonotonic and require systematic annotations with guards or tags, whereas our refined scheme detects that the second axiom is harmless and translates it directly to the untyped formula $\mathsf{toggle}(S) \neq S$.

After a brief review of the traditional polymorphic type encodings (Section 6.5.1), we present the polymorphic monotonicity inference calculus and the related type encodings (Section 6.5.2). Although the focus is on sound encodings, we also consider unsound ones, both as evaluation yardsticks and because applications that certify external proofs can safely employ them for proof search. Furthermore, we explore incomplete versions of the type encodings based on monomorphization (Section 6.5.3). The sound polymorphic encodings are proved sound and complete (Section 6.5.4).

The encodings are now available in Sledgehammer and *metis*. We evaluate the encodings' suitability for the resolution provers E, SPASS, and Vampire and the SMT solver Z3 in Section 6.7. Our comparison includes the traditional type encodings as well as the provers' native support for simple types where available.

From both a conceptual and an implementation point of view, the encodings are all instances of a general framework, in which mostly orthogonal features can be combined in various ways. Defining such a large number of encodings allows us to select the most appropriate encoding for each prover, based on the evaluation. In fact, because of time slicing (Section 6.6.3), it even pays off to have each prover employ a combination of encodings with complementary strengths.

The exposition builds on the following running examples.

**Example 6.3 (Monkey Village).** Imagine a TFF0 village of monkeys where each monkey owns at least two bananas:

$$\mathsf{owns}(M, \mathsf{banana1}(M)) \land \mathsf{owns}(M, \mathsf{banana2}(M))$$
$$\mathsf{banana1}(M) \neq \mathsf{banana2}(M)$$
$$\mathsf{owns}(M_1, B) \land \mathsf{owns}(M_2, B) \longrightarrow M_1 = M_2$$

(The required leap of imagination should not be too difficult for readers who can recall Section 4.1.) The predicate $\mathsf{owns} : \mathit{monkey} \times \mathit{banana} \to o$ associates *monkey*s with their *banana*s, and the functions $\mathsf{banana1}, \mathsf{banana2} : \mathit{monkey} \to \mathit{banana}$ witness the existence of each monkey's minimum supply of bananas. The type *banana* is monotonic, because any model with $k$ bananas can be extended to a model with $k' > k$ bananas. In contrast, *monkey* is nonmonotonic, because there can live at most $n$ monkeys in a village with a finite supply of $2n$ bananas.

**Example 6.4 (Algebraic Lists).** The following TFF1 axioms induce a minimalistic theory of algebraic lists:

$$\mathsf{nil} \neq \mathsf{cons}(X^{\alpha}, Xs)$$
$$Xs = \mathsf{nil} \lor \exists Y\ Ys.\ Xs = \mathsf{cons}(Y^{\alpha}, Ys)$$
$$\mathsf{hd}(\mathsf{cons}(X^{\alpha}, Xs)) = X$$
$$\mathsf{tl}(\mathsf{cons}(X^{\alpha}, Xs)) = Xs$$

We conjecture that cons is injective. Expressed negatively for an unknown but fixed type $b$, the conjecture becomes

$$\exists X\ Y\ Xs\ Ys.\ \mathsf{cons}(X^b,\ Xs) = \mathsf{cons}(Y,\ Ys) \wedge (X \neq Y \vee Xs \neq Ys)$$

Since the problem is unsatisfiable, all types are trivially monotonic.

### 6.5.1 Traditional Type Encodings

Encoding types in an untyped logic is an old problem, and many solutions have nearly folkloric status. Their main interest here is that they lay the foundation for the more efficient encodings introduced in Sections 6.5.2 and 6.5.3.

**Full Type Erasure (e).** The easiest way to translate polymorphic types to untyped first-order logic is to omit, or erase, all type information. This yields the e ("erased") encoding. Type erasure is conspicuously unsound in a logic that interprets equality, because different cardinality constraints can be attached to different types. For example, the e encoding translates the HOL exhaustion rule $u^{unit} = ()$ to the clause $U = \mathsf{unity}$, which forces a universe of cardinality 1 and can be used to derive a contradiction from any disequality $t \neq u$ or any pair of clauses $\mathsf{p}(\bar{t})$ and $\neg\, \mathsf{p}(\bar{u})$, where $\bar{t}$ and $\bar{u}$ are arbitrary. This unsoundness also plagues Hurd's translation [91, 92]. The partial solution proposed and implemented by Meng and Paulson is to leave out exhaustion rules of the forms

$$x = \mathsf{c}_1 \vee \cdots \vee x = \mathsf{c}_n$$

$$(x = \mathsf{c}_1 \Longrightarrow P) \Longrightarrow \cdots \Longrightarrow (x = \mathsf{c}_n \Longrightarrow P) \Longrightarrow P$$

which typically originate from Isabelle's datatype package and account for the vast majority of unsound proofs in practice. Nonetheless, crude syntactic test generally does not suffice to prevent unsound cardinality reasoning.

Type erasure is unsound not only because it allows unsound cardinality reasoning, but also because it confuses distinct monomorphic instances of polymorphic symbols. The HOL property

$$n \neq 0 \Longrightarrow n > 0^{nat}$$

holds for natural numbers, but it would be unsound for integers or real numbers.

**Type Arguments (a).** To ensure sound type class reasoning, we encode type information as additional arguments to the function and predicate symbols. More precisely, we pass one type argument corresponding to each type variable in the most general type for a constant. This is sufficient to reconstruct the constant's type. Following this scheme, $0^{nat}$ and $0^{int}$ are distinguished as $\mathsf{zero}(\mathsf{nat})$ and $\mathsf{zero}(\mathsf{int})$, and the polymorphic function application

$$\mathsf{map}^{(\alpha\to\beta)\to\alpha\ list\to\beta\ list}\ f\ xs$$

is translated to $\mathsf{map}(A,\ B,\ F,\ Xs)$. This approach is readily extended to type classes [125, §2.1]. We call the resulting encoding a; it corresponds to Meng and Paulson's constant-typed translation.

Although Meng and Paulson were careful to supply type information for HOL constants, they neglected to do so for the special operator $\mathsf{hAPP}$. This resulted in fascinating unsound proofs. Let $t \cdot u$ abbreviate $\mathsf{hAPP}(t,\, u)$, and let it associate to the left. From the HOL lemma $\mathsf{Suc}\; n \neq n$ and the definition of the $\mathsf{I}$, $\mathsf{K}$, and $\mathsf{S}$ combinators, the resolution provers can derive a contradiction, showing that something is wrong with the encoding:

1. $\mathsf{suc} \cdot N \neq N$ lemma
2. $\mathsf{i} \cdot X = X$ def. of $\mathsf{I}$
3. $\mathsf{k} \cdot X \cdot Y = X$ def. of $\mathsf{K}$
4. $\mathsf{s} \cdot X \cdot Y \cdot Z = X \cdot Z \cdot (Y \cdot Z)$ def. of $\mathsf{S}$
5. $\mathsf{s} \cdot X \cdot \mathsf{i} \cdot Z = X \cdot Z \cdot Z$ by 2, 4
6. $\mathsf{s} \cdot (\mathsf{k} \cdot X) \cdot Y \cdot Z = X \cdot (Y \cdot Z)$ by 3, 4
7. $\mathsf{s} \cdot (\mathsf{s} \cdot (\mathsf{k} \cdot X)) \cdot \mathsf{i} \cdot Y = X \cdot (Y \cdot Y)$ by 5, 6
8. $\bot$ by 1, 7

The last resolution step requires taking $X = \mathsf{suc}$ and $Y = \mathsf{s} \cdot (\mathsf{s} \cdot (\mathsf{k} \cdot \mathsf{suc})) \cdot \mathsf{i}$ in formula 7. In essence, the proof contradicts the lemma that $\mathsf{suc}$ admits no fixed point (a term $t$ such that $\mathsf{suc} \cdot t = t$) by exhibiting one.[1] It is a well-known result that every function in the untyped $\lambda$-calculus, which is encoded here using combinators, has a fixed point [10, §2.1].

To prevent such unsound proofs, we now treat $\mathsf{hAPP}$ as any other HOL constant of type $(\alpha \to \beta) \to \alpha \to \beta$ and pass type arguments corresponding to $\alpha$ and $\beta$. Despite this change, the $\mathsf{a}$ encoding is unsound, because like $\mathsf{e}$ it permits unsound cardinality reasoning.

**Type Tags (t).** A general approach to preclude unsound cardinality reasoning is to wrap each term and subterm in suitable functions [174, p. 99], which we call *type tags*. The traditional formulation relies on unary functions, one for each simple type to encode. To support polymorphism and $n$-ary type constructors, we use a binary function $\mathsf{t}(\sigma,\, t)$ that tags the term $t$ with its type $\sigma$, where $\sigma$ is encoded as a first-order term [63, §3.1; 125, §2.4].

Using this approach, the HOL lemmas $u = ()$ and $0 \neq \mathsf{Suc}\; n$ are translated to

$$\mathsf{t}(\mathsf{unit},\, U) = \mathsf{t}(\mathsf{unit},\, \mathsf{unity})$$
$$\mathsf{t}(\mathsf{nat},\, \mathsf{zero}) \neq \mathsf{t}(\mathsf{nat},\, \mathsf{suc}(\mathsf{t}(\mathsf{nat},\, N)))$$

Thanks to the type tags, the two formulas are consistent. Because equality is never used directly on variables but only on function applications (of $\mathsf{t}$), problems encoded this way are monotonic [61, §2.3], and consequently no formula can express an upper bound on the cardinality of the universe. Furthermore, since all terms carry full type information, we can safely omit the type arguments for polymorphic constants, as we have done here for $\mathsf{zero}$. This encoding corresponds to Meng and Paulson's fully typed translation; we call it $\mathsf{t}$.

The $\mathsf{t}$ encoding is sound in the sense that if a resolution prover finds a proof, then the problem is also provable in HOL. The resolution proof can still involve

[1] Expressed as an untyped $\lambda$-term, the fixed point found by the resolution prover is $(\lambda x.\; \mathsf{suc}\; (x\; x))$ $(\lambda x.\; \mathsf{suc}\; (x\; x))$, also known as $\mathsf{Y}\; \mathsf{suc}$ [10, §2.1].

type-incorrect inferences, since nothing prevents the prover from instantiating $U$ in $\mathsf{t}(\mathsf{unit}, U) = \mathsf{t}(\mathsf{unit}, \mathsf{unity})$ with a term of the wrong type (e.g., zero), an encoded type (e.g., nat), or even a term in which t tags are not properly nested. This is hardly a problem in practice, because resolution provers heavily restrict paramodulation from and into variables [8,48].

**Type Guards (g).** Type tags are but one way to encode types soundly and completely. A perhaps more intuitive option is to generate *type guards*—predicates that restrict the range of variables. For polymorphic type systems, they take the form of a binary predicate $\mathsf{g}(\sigma, t)$ that indicates whether $t$ has type $\sigma$, where $\sigma$ is encoded as a term. We call this type encoding g. Guard-based encodings yield formulas with a more complex logical structure than with tags, but the terms are correspondingly simpler.

Following the g encoding, the HOL lemmas $u = ()$ and $0 \neq \mathsf{Suc}\ n$ are translated to

$$\mathsf{g}(\mathsf{unit}, U) \longrightarrow U = \mathsf{unity}$$
$$\mathsf{g}(\mathsf{nat}, N) \longrightarrow \mathsf{zero} \neq \mathsf{suc}(N)$$

Each variable occurring in the generated CNF problem is guarded by the g predicate. When generating FOF problems, we must guard bound variables as well, with $\longrightarrow$ as the connective for $\forall$ and $\wedge$ for $\exists$.

To witness the inhabitation (nonemptiness) of all types, we include the axiom

$$\mathsf{g}(A, \mathsf{undefined}(A))$$

which declares the unspecified constant $\mathsf{undefined}^{\alpha}$. In addition, we must provide type information about the function symbols that occur in the problem:

$$\mathsf{g}(\mathsf{unit}, \mathsf{unity}) \qquad \mathsf{g}(\mathsf{nat}, \mathsf{zero}) \qquad \mathsf{g}(\mathsf{nat}, \mathsf{suc}(N))$$

In the third axiom, there is no need to guard $N$ because Suc always returns a natural number irrespective of its argument. This encoding gives a type to some ill-typed terms, such as $\mathsf{suc}(\mathsf{suc}(\mathsf{unity}))$ and $\mathsf{suc}(\mathsf{suc}(\mathsf{nat}))$. Intuitively, this is safe because such terms cannot bring the proof forward (except to witness inhabitation, but even in that role they are redundant). On the other hand, well-typed terms must always be associated with their correct type, as they are in this encoding.

We must include type arguments that occur only in the result type of a constant, to distinguish instances of polymorphic constants, but all other type arguments can be omitted, since they can be deduced from the function's arguments. Thus, the type argument would be kept for nil but omitted for cons (e.g., $\mathsf{nil}(A)$ vs. $\mathsf{cons}(X, Xs)$).

### 6.5.2 Sound Type Erasure via Monotonicity Inference

Type guards and tags significantly increase the size of the problems passed to the automatic provers, with a dramatic impact on their performance. Fortunately, most of the clutter can be removed by inferring monotonicity and (soundly) erasing type information based on a monotonicity analysis.

**Polymorphic Monotonicity Inference.** A type $\sigma$ is *monotonic* in a formula $\varphi$ if any model of $\varphi$ where $\sigma$ has cardinality $k$ can be extended into a model where it has cardinality $k'$, for any $k' > k$ (Chapter 4). Claessen et al. devised an extremely simple (yet quite powerful) calculus to infer monotonicity for monomorphic first-order logic [61, §2.3], based on the observation that a type $\sigma$ must be monotonic if the problem contains no positive literal $X^{\sigma} = t$ (or $t = X^{\sigma}$).[1] We call such an occurrence of $X$ a *positively naked* occurrence. The calculus is powerful enough to infer that *banana* is monotonic in Example 6.3, since the monkey village contains no literal of the form $B^{\mathit{banana}} = t$.

It is not difficult to extend the approach to handle polymorphism. Semantically, a polymorphic type is monotonic iff all of its ground instances are monotonic. The extended calculus computes the set of *possibly nonmonotonic* polymorphic types, which consists of all types $\sigma$ such that there is a positively naked variable of type $\sigma$. Each nonmonotonic ground type is an instance of a type in this set. To infer that a polymorphic type $\sigma$ is monotonic, we check that there is no possibly nonmonotonic type unifiable with $\sigma$.[1] Annoyingly, a single occurrence of a positively naked variable of type $\alpha$, such as $X$ in the equation

$$\mathsf{hd}(\mathsf{cons}(X,\, Xs)) = X^{\alpha}$$

from Example 6.4, is enough to completely flummox the analysis: Since all types are instances of $\alpha$, they are all possibly nonmonotonic.

**Infinity Inference.** We regain some precision by complementing the calculus with an infinity analysis, as suggested by Claessen et al.: By the Löwenheim–Skolem theorem, all types with no finite models are monotonic (with respect to finite and countable models). We call such types *infinite.*

We could employ an approach similar to that implemented in Infinox [60] to automatically infer finite unsatisfiability of types. Infinox relies on various proof principles to show that a set of untyped first-order formulas only has models with infinite domains. For example, given a problem containing the axiom

$$\mathsf{zero} \neq \mathsf{suc}(X) \,\wedge\, (X \neq Y \longrightarrow \mathsf{suc}(X) \neq \mathsf{suc}(Y))$$

Infinox can establish that suc is injective but not surjective and hence (by a well-known lemma) the domain must be infinite. The approach is readily generalizable to polymorphic first-order logic. It can be used to infer that $\alpha$ *list* is infinite in Example 6.4 because cons is injective in its second argument but not surjective.

However, in an interactive theorem prover, it is simpler to exploit the metainformation available through introspection. Inductive datatypes are registered with their constructors; if some of them are recursive, or take an argument of an infinite type, the datatype must be infinite.

[1] Claessen et al. designed a second, more powerful calculus to detect predicates that act as fig leaves for positively naked variables. While the calculus proved fairly successful on a subset of the TPTP benchmark suite [177], we assessed its suitability on about 1000 fairly large problems generated by Sledgehammer and found no improvement on the first calculus. In this thesis, we restrict our attention to the first calculus.

[1] Unification is to be understood in a wider sense, with implicit renaming of variables to avoid name clashes. Hence, $\alpha$ *list* is an instance of $\alpha$ (but $\mathit{nat} \rightarrow \mathit{int}$ is not an instance of $\alpha \rightarrow \alpha$). It may help to think of $\alpha$ as $\forall\alpha.\ \alpha$ and $\alpha$ *list* as $\forall\alpha.\ \alpha\ \mathit{list}$.

Combining infinity inference with the monotonicity inference calculus described above, we get the following rule for inferring monotonicity:

> A polymorphic type is monotonic if, whenever it is unifiable with a possibly nonmonotonic type, the most general unifier is an instance of an infinite type.

Our rule is correct because if we infer a type monotonic, all its ground instances either are infinite or can be inferred monotonic by the monotonicity calculus.

**Type Erasure with Guards (g?, g??).** Claessen et al. observed that monotonic types can be soundly erased when translating from a monomorphic logic to an untyped logic, whereas nonmonotonic types must generally be encoded, typically using guards or tags [61, §3.2]. In particular, type erasure as performed by the type argument encoding a is sound if all types are monotonic. We extend the approach to polymorphism and show how to eliminate even more type information in the monomorphic case.

We first focus on type guards. The following procedure soundly translates problems from polymorphic first-order logic (TFF1) to untyped first-order logic (FOF):

1. Introduce type arguments to all polymorphic function and predicate symbols, as done for the a encoding.
2. Insert guards for the types that cannot be inferred monotonic, and introduce suitable typing axioms.
3. Erase all the types.

By adding guards and typing axioms, step 2 effectively makes the nonmonotonic types monotonic. Once all types are monotonic, step 3 can safely erase them. We call the resulting encoding g?. In contrast to the traditional g encoding, g? generally requires type arguments to compensate for the incomplete type information.

**Example 6.5.** Encoded with g?, the monkey village of Example 6.3 becomes

$$\mathsf{g}(A,\ \mathsf{undefined}(A))$$

$$\begin{aligned}
&\mathsf{g}(\mathsf{monkey},\ M) \longrightarrow \mathsf{owns}(M,\ \mathsf{banana1}(M)) \wedge \mathsf{owns}(M,\ \mathsf{banana2}(M))\\
&\mathsf{g}(\mathsf{monkey},\ M) \longrightarrow \mathsf{banana1}(M) \neq \mathsf{banana2}(M)\\
&\mathsf{g}(\mathsf{monkey},\ M_1) \longrightarrow \mathsf{g}(\mathsf{monkey},\ M_2) \longrightarrow\\
&\qquad \mathsf{owns}(M_1,\ B) \wedge \mathsf{owns}(M_2,\ B) \longrightarrow M_1 = M_2
\end{aligned}$$

Notice that no guard is generated for the variable $B$ of type *banana*. This coincides with Claessen et al. [61, §2.3].

Typing axioms are needed to discharge the guards. We could in principle simply generate typing axioms $\mathsf{g}(\sigma,\ \mathsf{f}(\bar{X}))$ for every function symbol f, as in the g encoding, but some of these axioms are superfluous. We reduce clutter in two ways:

- If $\sigma$ is not unifiable with any of the possibly nonmonotonic types, the typing axiom will never be resolvable against a guard and can be omitted.

- For infinite types $\sigma$, it suffices to generate an axiom $\mathsf{g}(\sigma, X)$ that allows *any* term to be typed as $\sigma$. Such an axiom is sound for any monotonic type $\sigma$, as we will prove in Section 6.5.4. (The purpose of type guards is to prevent instantiating variables of nonmonotonic types with terms of the wrong type; they play no role for monotonic types.)

**Example 6.6.** For the algebraic list problem of Example 6.4, our monotonicity inference reports that $\alpha$ is possibly nonmonotonic, but *$\alpha$ list* is infinite. The g? encoding of the problem follows:

$$\begin{array}{l}
\mathsf{g}(A, \mathsf{undefined}(A)) \\
\mathsf{g}(\mathsf{list}(A), Xs) \\
\mathsf{g}(A, \mathsf{hd}(A, Xs)) \\[1ex]
\mathsf{g}(A, X) \longrightarrow \mathsf{nil}(A) \neq \mathsf{cons}(A, X, Xs) \\
Xs = \mathsf{nil}(A) \lor \exists Y\ Ys.\ \mathsf{g}(A, Y) \land Xs = \mathsf{cons}(A, Y, Ys) \\
\mathsf{g}(A, X) \longrightarrow \mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) = X \\
\mathsf{g}(A, X) \longrightarrow \mathsf{tl}(A, \mathsf{cons}(A, X, Xs)) = Xs \\[1ex]
\exists X\ Y\ Xs\ Ys.\ \mathsf{g}(\mathsf{b}, X) \land \mathsf{g}(\mathsf{b}, Y) \land \\
\qquad\qquad\qquad \mathsf{cons}(\mathsf{b}, X, Xs) = \mathsf{cons}(\mathsf{b}, Y, Ys) \land (X \neq Y \lor Xs \neq Ys)
\end{array}$$

The second typing axiom allows any term to be typed as *$\alpha$ list*, which is sound because *$\alpha$ list* is infinite; alternatively, we could have provided individual typing axioms for $\mathsf{nil}$, $\mathsf{cons}$, and $\mathsf{tl}$. Either way, the axioms are needed to discharge the $\mathsf{g}(A, X)$ guards in case the proof requires reasoning about *$\alpha$ list list*.

The g? encoding treats all variables of the same type uniformly. Hundreds of axioms can be penalized because of one unpleasant formula that uses a type nonmonotonically (or in a way that cannot be inferred monotonic). In Example 6.5, the positively naked variables in

$$\mathsf{owns}(M_1, B) \land \mathsf{owns}(M_2, B) \longrightarrow M_1 = M_2$$

are enough to force the use of guards for all *monkey* variables in the problem. Fortunately, a lighter encoding is possible: Guards are useful only for essentially universal (i.e., non-skolemizable) variables that occur positively naked; they can soundly be omitted for all other variables, irrespective of whether they have a monotonic type, as we will prove in Section 6.5.4. This is related to the observation that only paramodulation from or into a (positively naked) variable can cause ill-typed instantiations in a resolution prover [205, §4]. We call this lighter encoding g??.

**Example 6.7.** Let us return to the monkey village. Encoded with g??, it requires only two type guards, a clear improvement over g? and Claessen et al.:

$$\begin{array}{l}
\mathsf{g}(A, \mathsf{undefined}(A)) \\[1ex]
\mathsf{owns}(M, \mathsf{banana1}(M)) \land \mathsf{owns}(M, \mathsf{banana2}(M)) \\
\mathsf{banana1}(M) \neq \mathsf{banana2}(M) \\
\mathsf{g}(\mathsf{monkey}, M_1) \longrightarrow \mathsf{g}(\mathsf{monkey}, M_2) \longrightarrow \\
\qquad \mathsf{owns}(M_1, B) \land \mathsf{owns}(M_2, B) \longrightarrow M_1 = M_2
\end{array}$$

**Example 6.8.** The g?? encoding of Example 6.4 is identical to g? except that the $\mathsf{nil} \neq \mathsf{cons}$ and $\mathsf{tl}$ axioms do not need any guard (cf. Example 6.6).

**Type Erasure with Tags (t?, t??).** Analogously to g? and g??, we define t? and t?? encodings based on type tags. The t? encoding annotates all terms of a possibly nonmonotonic type that is not infinite. This can result in mismatches—for example, if $\alpha$ is tagged but its instance *α list* is not. The solution is to generate an equation $\mathsf{t}(\sigma, X) = X$ for each infinite type $\sigma$, which allows the prover to add or remove a tag whenever necessary.

The lighter encoding t?? only annotates naked variables, whether positive or negative, and introduces equations $\mathsf{t}(\sigma, \mathsf{f}(\overline{X})) = \mathsf{f}(\overline{X})$ to add or remove tags around each function symbol (or skolemizable variable) f of a possibly nonmonotonic type $\sigma$.[1] For monotonicity, it is not necessary to tag negatively naked variables, but a uniform treatment of naked variables ensures that resolution can be directly applied on equality atoms. This encoding works well in practice, because provers tend to aggressively eliminate type tags using the typing equations as left-to-right rewrite rules.

**Example 6.9.** The t? encoding of Example 6.4 is as follows:

$$\begin{array}{l}
\mathsf{t}(A,\, \mathsf{undefined}(A)) = \mathsf{undefined}(A) \\
\mathsf{t}(\mathsf{list}(A),\, Xs) = Xs \\[1ex]
\mathsf{nil}(A) \neq \mathsf{cons}(A,\, \mathsf{t}(A,\, X),\, Xs) \\
Xs = \mathsf{nil}(A) \lor \exists Y\; Ys.\; Xs = \mathsf{cons}(A,\, \mathsf{t}(A,\, Y),\, Ys) \\
\mathsf{t}(A,\, \mathsf{hd}(A,\, \mathsf{cons}(A,\, \mathsf{t}(A,\, X),\, Xs))) = \mathsf{t}(A,\, X) \\
\mathsf{tl}(A,\, \mathsf{cons}(A,\, \mathsf{t}(A,\, X),\, Xs)) = Xs \\[1ex]
\exists X\; Y\; Xs\; Ys.\; \mathsf{cons}(\mathsf{b},\, \mathsf{t}(\mathsf{b},\, X),\, Xs) = \mathsf{cons}(\mathsf{b},\, \mathsf{t}(\mathsf{b},\, Y),\, Ys) \land \\
\qquad\qquad\quad (\mathsf{t}(\mathsf{b},\, X) \neq \mathsf{t}(\mathsf{b},\, Y) \lor Xs \neq Ys)
\end{array}$$

**Example 6.10.** The t?? encoding of Example 6.4 requires fewer tags, at the cost of more type information (for hd and some of the existential variables):

$$\begin{array}{l}
\mathsf{t}(A,\, \mathsf{undefined}(A)) = \mathsf{undefined}(A) \\
\mathsf{t}(\mathsf{list}(A),\, Xs) = Xs \\
\mathsf{t}(A,\, \mathsf{hd}(A,\, Xs)) = \mathsf{hd}(A,\, Xs) \\[1ex]
\mathsf{nil}(A) \neq \mathsf{cons}(A,\, X,\, Xs) \\
Xs = \mathsf{nil}(A) \lor \exists Y\; Ys.\; \mathsf{t}(A,\, Y) = Y \land Xs = \mathsf{cons}(A,\, Y,\, Ys) \\
\mathsf{hd}(A,\, \mathsf{cons}(A,\, X,\, Xs)) = \mathsf{t}(A,\, X) \\
\mathsf{tl}(A,\, \mathsf{cons}(A,\, X,\, Xs)) = Xs \\[1ex]
\exists X\; Y\; Xs\; Ys.\; \mathsf{t}(\mathsf{b},\, X) = X \land \mathsf{t}(\mathsf{b},\, Y) = Y \land \mathsf{cons}(\mathsf{b},\, X,\, Xs) = \mathsf{cons}(\mathsf{b},\, Y,\, Ys) \land \\
\qquad\qquad\quad (X \neq Y \lor Xs \neq Ys)
\end{array}$$

To conclude this section, we describe an "obviously sound" optimization that is, in fact, unsound. Consider the equational definition of hd in Isabelle/HOL:

$$\mathsf{hd}\ (\mathsf{Cons}\ x^{\alpha}\ xs) = x$$

Clearly, this lemma does not encode any cardinality constraint on the possible instances for $\alpha$; otherwise, it would be easy to derive a contradiction in HOL by

[1] We can optionally provide equations $\mathsf{f}(\overline{X},\, \mathsf{t}(\sigma,\, Y),\, \overline{Z}) = \mathsf{f}(\overline{X},\, Y,\, \overline{Z})$ to add or remove tags around well-typed arguments of a function symbol f, as well as the tag idempotence law $\mathsf{t}(A,\, \mathsf{t}(A,\, X)) = \mathsf{t}(A,\, X)$. Our experiments with them have been inconclusive so far.

instantiating $\alpha$ appropriately. Hence, nothing can go wrong if we omit the type tag around the naked variable in the translation (or so we think):

$$\mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) = X \tag{$*$}$$

From the typing axiom

$$\mathsf{t}(A, \mathsf{hd}(A, Xs)) = \mathsf{hd}(A, Xs)$$

we can derive the instance

$$\mathsf{t}(A, \mathsf{hd}(A, \mathsf{cons}(A, X, Xs))) = \mathsf{hd}(A, \mathsf{cons}(A, X, Xs))$$

By rewriting both sides of the equation with $(*)$, we obtain

$$\mathsf{t}(A, X) = X$$

This last equation is extremely powerful: It can be used to remove the tag around any term (whether the type is correct or not) or to tag any term with any type, defeating the very purpose of type tags.

The nub of the issue is this: Our encoding procedure crucially relies on the problem being monotonic after the tags and type axioms have been introduced, so that the types can be safely erased. The equation $(*)$ is harmless on its own and can be seen as a "monotonic definition," following the ideas presented in Section 4.5.1. However, the equation $(*)$ and the typing axiom, when taken together, compromise monotonicity. For instance, the set of TFF1 formulas

$\mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) = X$
$\mathsf{t}(A, \mathsf{hd}(A, Xs)) = \mathsf{hd}(A, Xs)$
$\mathsf{t}(\mathsf{unit}, X) = \mathsf{t}(\mathsf{unit}, Y)$

has only one model, of cardinality 1: The first two formulas require $\mathsf{t}(\mathsf{unit}, X)$ to act as an identity function on its second argument (since they have $\mathsf{t}(A, X) = X$ as a consequence, as we saw above), whereas the last formula forces $\mathsf{t}(\mathsf{unit}, X)$ to ignore the second argument.

In sum, the envisioned calculus that naively combines the definition handling of Section 4.5.1 with the monotonicity inference of Claessen et al. is unsound. If there is a lesson to be learned here, it must be that theoretical properties should be proved (or at least checked with Nitpick) even when they seem obvious.

**Finiteness Inference (g!, g!!, t!, t!!).** A radical approach is to assume every type is infinite unless we have metainformation to the contrary. Only types that are obviously finite, such as *unit*, $o$, $o \times o$, and $o \to o$, are considered by the monotonicity inference calculus. This is of course unsound; users can encode finiteness constraints as HOL axioms or local assumptions, as in the example below [137]:

**typedecl** *indi*

**axiomatization where**
*finite_indi*: $\mathsf{finite}\ \mathsf{UNIV}^{\mathit{indi} \to o}$

The finiteness of *indi* escapes our simple analysis. Nonetheless, unsound proofs are a rare occurrence with this scheme, and the user can fall back on sound type systems if an unsound proof is found. We identify this family of unsound encodings by the suffix ! instead of ? (e.g., g!, t!!).

### 6.5.3 Monomorphization-Based Encodings

Type variables give rise to term variables in encoded formulas; for example, the type *α list* is encoded as $\mathsf{list}(A)$. These variables dramatically increase the search space and complicate the translation, because type class predicates must be included in the problem to restrict the range of variables [127, §2.1]. An alternative is to monomorphize the problem—that is, to heuristically instantiate the type variables with ground types. This is reminiscent of Nitpick's (and Refute's) treatment of definitions (Section 3.3), but Sledgehammer must handle arbitrary lemmas. Monomorphization was applied successfully in the SMT integration developed by Böhme and briefly described in Section 6.3.1. We were able to reuse his code to preprocess ATP problems.

Monomorphization is necessarily incomplete [40, §2] and often overlooked or derided in the literature; for example, in their work on encoding polymorphism in SMT-LIB's many-sorted logic, Couchot and Lescuyer contend that "monomorphization always returns theories that are much bigger than the original ones, which dramatically slows down provers" [63, p. 3]. Our experience is that monomorphization can dramatically *improve* performance (Section 6.7.2).

**Monomorphization ($\bar{-}$).** The monomorphization algorithm consists of the following three stages [41, §2.2.1]:

1. Separate the monomorphic and the polymorphic formulas, and collect all symbols appearing in the monomorphic formulas (the "mono-symbols").
2. For each polymorphic axiom, stepwise refine a set of substitutions, starting from the singleton set containing only the empty substitution, by matching known mono-symbols against their polymorphic counterparts. As long as new mono-symbols emerge, collect them and repeat this stage.
3. Apply the computed substitutions to the corresponding polymorphic formulas. Only keep fully monomorphic formulas.

To avoid divergence, the implementation limits the iterations performed in stage 2 to a configurable number $K$. To curb the exponential growth, it also enforces an upper bound $\Delta$ on the number of new formulas. Sledgehammer operates with $K = 3$ and $\Delta = 200$ by default, so that a problem with 500 axioms comprises at most 700 axioms after monomorphization. Experiments found these values suitable. Given formulas about *nat* and *α list*, the third iteration already generates *nat list list list* instances—yet another layer of *list* is unlikely to help. Increasing $\Delta$ can help solve more goals, but its potential for clutter is real.

Monomorphization is applicable in conjunction with all the type encodings presented so far except e (which erases all types). We decorate the letter representing a type encoding with $\bar{}$ to indicate monomorphization. Accordingly, $\bar{\mathsf{g}}$? is the monomorphic version of the type guard encoding g?.

One unexpected advantage of monomorphization is that provers detect unprovability of small problems much faster than before. This is useful for debugging and speeds up minimization (Section 6.6.5). Although monomorphization is incomplete [63, p. 265], with a high bound on the number of iterations we can be fairly confident that all necessary monomorphic instances are included.

**Type Mangling (~).** Monomorphization opens the way to an easy optimization: Since all types represented as terms are ground, we can mangle them in the enclosing symbol's name to lighten the translation. In our examples, we mangle with underscores: $\mathsf{zero}(\mathsf{nat})$, $\mathsf{g}(\mathsf{nat}, N)$, and $\mathsf{t}(\mathsf{nat}, N)$ become zero_nat, $\mathsf{g\_nat}(N)$, and $\mathsf{t\_nat}(N)$; in the implementation, we rely on a more sophisticated scheme to avoid name clashes, ensuring that the mangled problem is equisatisfiable to the original monomorphized problem. We identify the mangled, monomorphic version of an encoding with the decoration $\tilde{}$ (e.g., $\tilde{\mathsf{g}}$?).

**Native First-Order Types (n).** Most popular ATPs take their input in the TPTP first-order form (FOF), a first-order logic with equality and quantifiers but no types. A selected few, such as SNARK [175], the metaprovers ToFoF [176] and Monotonox [61], and recent versions of Vampire, also support the monomorphic TPTP typed first-order form (TFF0), which provides simple types (sorts). (The SMT-LIB format [11, 160] offers similar support.) The developers of SPASS recently added support for simple types in an unofficial version of the prover [36], and hence it becomes increasingly important to exploit native support where it is available.

The mangled type guard encoding $\tilde{\mathsf{g}}$ also constitutes a suitable basis for generating TFF0 problems. In $\tilde{\mathsf{g}}$, each variable is guarded by a g predicate mangled with a ground type (e.g., $\mathsf{g\_list\_nat}(Xs)$). In the corresponding TFF0-based encoding, which we call $\tilde{\mathsf{n}}$ ("native"), the variable is declared with the guard's type, and the guard is omitted. For example, the $\tilde{\mathsf{g}}$-encoded FOF formula

$$\forall N.\ \mathsf{g\_nat}(N) \longrightarrow \mathsf{zero} \neq \mathsf{suc}(N)$$

corresponds to the $\tilde{\mathsf{n}}$-encoded TFF0 formula

$$\forall N^{nat}.\ \mathsf{zero} \neq \mathsf{suc}(N)$$

Typing axioms such as

$$\mathsf{g\_nat}(\mathsf{zero\_nat}) \qquad\qquad \mathsf{g}(\mathsf{nat}, \mathsf{suc}(N))$$

are replaced by TFF0 type declarations:

$$\mathsf{zero\_nat} : nat \qquad\qquad \mathsf{suc} : nat \rightarrow nat$$

Thanks to type mangling, this also works for more complex types. For example, the $nat \times int \rightarrow nat$ instance of $\mathsf{fst}^{\alpha \times \beta \rightarrow \alpha}$ is declared as

$$\mathsf{fst\_nat\_int} : prod\_nat\_int \rightarrow nat$$

Unlike type guards, which can be kept or omitted on a per-variable basis, simple types must be kept for all variables. Nonetheless, we can define $\tilde{\mathsf{n}}$? and $\tilde{\mathsf{n}}$! based on $\tilde{\mathsf{g}}$? and $\tilde{\mathsf{g}}$! not by erasing types, but rather by replacing the types we would erase by a shared type of individuals. We use the predefined $\iota$ type from TFF0 for this purpose, but any fresh type would do. Furthermore, since TFF0 disallows overloading, all type arguments must be kept and mangled in the symbol names, as in the fst_nat_int example above.

The polymorphic SMT solver Alt-Ergo [38] can process TFF1 problems, with some help from the Why3 translation tool [39]. It was not hard to extend Sledgehammer to generate TFF1 problems, using $\mathsf{g}$ as a basis, yielding the encodings $\mathsf{n}$, $\overline{\mathsf{n}}$, $\mathsf{n?}$, $\overline{\mathsf{n}}\mathsf{?}$, $\mathsf{n!}$, and $\overline{\mathsf{n}}\mathsf{!}$. Ignoring Isabelle type classes, the plain $\mathsf{n}$ encoding amounts to the identity, but in the implementation it converts Sledgehammer's internal data structures to the concrete TFF1 syntax; the other five encodings are effectively preprocessing steps that perform monomorphization, merge selected types to $\iota$, or both.

The only minor complication concerns type class predicates. The natural encoding assigns them the type signature $\forall\alpha.\ o$ (i.e., a type-indexed family of Booleans), as in the axiom

$$\mathsf{preorder}\langle\alpha\rangle \longrightarrow \forall X\, Y^{\alpha}.\ \mathsf{less}(X,\, Y) \longrightarrow \mathsf{less\_eq}(X,\, Y)$$

where $\langle\alpha\rangle$ is an explicit type argument. The type variable $\alpha$ in $\forall\alpha.\ o$ is called a *phantom type variable* because it does not occur in the type's body. ML-style provers (such as Alt-Ergo and Isabelle/HOL itself) lack $\forall$ binders for types and hence cannot directly cope with phantom types; fortunately, phantoms are easy to preprocess away [35, §5.1].

**Native Higher-Order Types ($\mathsf{N}$).** The core language THF0 of the TPTP typed higher-order form is understood natively by LEO-II [19] and Satallax [9]. To benefit from these and future higher-order ATPs, it is desirable to have Sledgehammer output THF0 problems. This extension is unpublished joint work with Nik Sultana.

Starting from the TFF0-based $\widetilde{\mathsf{n}}$ encoding, we gradually exploited the THF0 format to define a higher-order encoding, which we call $\widetilde{\mathsf{N}}$:

1. Adopt the @ syntax for function application, so that the generated problems comply with THF0.
2. Identify the HOL type $o$ with the homologous THF0 type and eliminate the $\mathsf{hBOOL}$ predicate.
3. Identify the HOL function type with the THF0 function type and replace the explicit application operator $\mathsf{hAPP}$ with @.
4. Map higher-order (unpolarized) occurrences of HOL connectives and quantifiers to the corresponding THF0 constructs, eliminating the need for proxies.
5. Let $\lambda$-abstractions pass through the translation, instead of rewriting them to combinators or lifting them.
6. For Satallax, identify the Hilbert choice constant from HOL with the corresponding THF0 operator. (LEO-II currently does not support Hilbert choice.)

The $\widetilde{\mathsf{N}}$ and $\widetilde{\mathsf{N}}\mathsf{!}$ encodings are higher-order versions of the TFF0-based $\widetilde{\mathsf{n}}$ and $\widetilde{\mathsf{n}}\mathsf{!}$ encodings. These are the last members in our family of type encodings; the whole family is charted in Figure 6.2. An $\widetilde{\mathsf{N}}\mathsf{?}$ encoding is in principle possible, but it would have to be more than a straightforward adaptation of $\widetilde{\mathsf{n}}\mathsf{?}$, whose monotonicity inference calculus is designed for first-order logic.

| | | Typing of Variables | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | All | | | Maybe Fin. | | | Finite | | | None | | |
| | | Polym. | Monom. | Mangled | Polym. | Monom. | Mangled | Polym. | Monom. | Mangled | Polym. | Monom. | Mangled |
| Tags | Heavy | t | t̄ | t̃ | t? | t̄? | t̃? | t! | t̄! | t̃! | | | |
| | Light | | | | t?? | t̄?? | t̃?? | t!! | t̄!! | t̃!! | | | |
| Guards | Heavy | g | ḡ | g̃ | g? | ḡ? | g̃? | g! | ḡ! | g̃! | | | |
| | Light | | | | g?? | ḡ?? | g̃?? | g!! | ḡ!! | g̃!! | | | |
| Native | F.-O. | n | n̄ | ñ | n? | n̄? | ñ? | n! | n̄! | ñ! | | | |
| | H.-O. | | | Ñ | | | | | | Ñ! | | | |
| Args. | | | | | | | | | | | a | ā | ã |
| Fully Erased | | | | | | | | | | | e | | |
| | | Sound | | | | | | Unsound | | | | | |

**Figure 6.2:** Overview of the type encodings

The translation to THF0 is a central aspect of the extension of Sledgehammer with LEO-II and Satallax, but other parts of the machinery also need some adjustments. First, the relevance filter normally leaves out induction rules; however, it makes sense to include them uninstantiated in THF0 problems. Second, proof reconstruction is problematic. As an expedient, we currently extract the referenced facts from LEO-II and Satallax proofs, detecting applications of the extensionality rule, and attempt to reconstruct the proof with a one-line *metis* call.[1]

### 6.5.4 Soundness and Completeness

Despite proof reconstruction, our type encodings deserve a detailed proof of correctness. Obviously, we cannot prove the unsound encodings correct, but this still leaves g, g?, g??, t, t?, t??, n, and n? with and without monomorphization and type mangling (–, ~) and also Ñ. We focus on the mangled-monomorphic and polymorphic versions of g?, g??, t?, and t??; the traditional encodings g and t are accounted for in the literature, monomorphization without mangling (–) is a special case of polymorphism, ñ and Ñ essentially rely on the soundness of type mangling, and the soundness proof for n? is analogous to that for g?.

To cope with the variety of type encodings, we need a modular proof that isolates their various features. We start by proving the mangled encodings sound and complete when applied to already monomorphic problems—i.e., they preserve satisfiability and unsatisfiability. (Monomorphization in itself is obviously sound, although incomplete.) Then we proceed to lift the proof to polymorphic encodings.

[1] As part of his Ph.D. thesis, Sultana is working on a *leo2* proof method that performs step-by-step proof reconstruction for LEO-II, similar in spirit to Böhme's Z3-based *smt* method.

**The Monomorphic, Mangled Case.** To prove $\widetilde{\mathrm{g}}?$, $\widetilde{\mathrm{g}}??$, $\widetilde{\mathrm{t}}?$, and $\widetilde{\mathrm{t}}??$ correct, we follow the two-stage proof strategy put forward by Claessen et al. [61, §3.2]: The first stage adds guards or tags without erasing any types, so that the formulas remain typed, and the second stage erases the types. We call the original problem $\mathrm{A}^\tau$, the intermediate problem $\mathrm{Z}^\tau$, and the final problem Z, as summarized below:

| NAME | DESCRIPTION | LOGIC | EXAMPLE |
|---|---|---|---|
| $\mathrm{A}^\tau$ | Original problem | TFF0 | $V^b = \mathsf{c}$ |
| $\mathrm{Z}^\tau$ | Encoded $\mathrm{A}^\tau$ | TFF0 | $\mathsf{t\_b}(V^b) = \mathsf{c\_b}$ |
| Z | Type-erased $\mathrm{Z}^\tau$ | FOF | $\mathsf{t\_b}(V) = \mathsf{c\_b}$ |

(The $\tau$ superscripts stand for "typed.") The following result, due to Claessen et al. [61, §2.2], plays a key role in the proof:

**Lemma 6.11 (Monotonic Type Erasure).** *Let $\Phi^\tau$ be a monomorphic problem. If $\Phi^\tau$ is monotonic (i.e., all of its types are monotonic), then $\Phi^\tau$ is equisatisfiable to its type-erased variant $\Phi$.*

*Proof.* Let $\mathcal{M}$ be a model of $\Phi^\tau$. By monotonicity, there exists a model $\mathcal{N}$ where all the domains have the cardinality of the largest domain in $\mathcal{M}$. From $\mathcal{N}$, we construct a model of $\Phi$ by identifying all the domains. Conversely, from a model $\mathcal{N}$ of $\Phi$ we construct a model of $\Phi^\tau$ with the same interpretations of functions and predicates as in $\mathcal{N}$ and with $\mathcal{N}$'s unique domain as the domain for every type. □

**Corollary 6.12 (Equisatisfiability Conditions).** *The problems $\mathrm{A}^\tau$ and Z are equisatisfiable if the following conditions hold:*

MONO: *$\mathrm{Z}^\tau$ is monotonic.*

SOUND: *If $\mathrm{A}^\tau$ is satisfiable, then so is $\mathrm{Z}^\tau$.*

COMPLETE: *If $\mathrm{Z}^\tau$ is satisfiable, then so is $\mathrm{A}^\tau$.*

We show the conditions of Corollary 6.12 separately for guards and tags. The proofs rely on the following lemma:

**Lemma 6.13 (Domain Restriction).** *Let $\mathcal{M}$ be a model of $\Phi$, and let $\mathcal{M}'$ be an interpretation constructed from $\mathcal{M}$ by deleting some domain elements while leaving the interpretations of functions and predicates intact. This $\mathcal{M}'$ is a model of $\Phi$ provided that*

(a) *we do not make any domain empty;*

(b) *we do not remove any domain element that is in the range of a function;*

(c) *we do not remove any witness for an existential variable.*

*Proof.* For simplicity, suppose the problem is expressed in CNF, in which case (b) subsumes (c). Conditions (a) and (b) ensure that $\mathcal{M}'$ is well-defined and that ground clauses are interpreted as in $\mathcal{M}$. Since every domain element of $\mathcal{M}'$ is also in $\mathcal{M}$, all clauses that are satisfied in $\mathcal{M}$ are also satisfied in $\mathcal{M}'$. □

**Theorem 6.14 (Correctness of Monomorphic Guards).** *The encodings $\widetilde{\mathrm{g}}?$ and $\widetilde{\mathrm{g}}??$ are sound and complete for monomorphic problems.*

*Proof.* It suffices to show that the three conditions of Corollary 6.12 are fulfilled.

MONO: Infinite types are monotonic. The other types are monotonic if all positively naked variables of their types are guarded [61, §2.4]. Both $\widetilde{\mathrm{g}}$? and $\widetilde{\mathrm{g}}$?? guard all such variables—$\widetilde{\mathrm{g}}$?? guards exactly those variables, while $\widetilde{\mathrm{g}}$? guards more.

SOUND: Given a model of $\mathrm{A}^{\tau}$, we extend it to a model of $\mathrm{Z}^{\tau}$ by giving an interpretation to the type guards. To do this, we simply interpret all type guards by the true predicate (the predicate that is true everywhere).

COMPLETE: A model of $\mathrm{Z}^{\tau}$ is *canonical* if all guards are interpreted by the true predicate. From a canonical model, we obtain a model of $\mathrm{A}^{\tau}$ by the converse construction to SOUND. It then suffices to prove that whenever there exists a model of $\mathrm{Z}^{\tau}$, there exists a canonical model. We appeal to Lemma 6.13 to remove the domain elements that do not satisfy their guard predicate. For this to work, (a) each predicate must be satisfied by at least one element, (b) each function must satisfy its predicate, and (c) each existential variable must satisfy its predicate; this is exactly what our typing axioms ensure. □

**Theorem 6.15 (Correctness of Monomorphic Tags).** *The encodings* $\widetilde{\mathrm{t}}$? *and* $\widetilde{\mathrm{t}}$?? *are sound and complete for monomorphic problems.*

*Proof.* The proof for tags is analogous to that for guards, so we leave out the details. A model of $\mathrm{Z}^{\tau}$ is *canonical* if the type tags are interpreted by the identity function. We construct a canonical model by deleting the domain elements for which the type tag is not the identity. The typing axioms ensure that this gives us a model. □

The above proof goes through even if we tag more terms than are necessary to ensure monotonicity. Hence, it is sound to tag negatively naked variables. We may also add further typing axioms to $\mathrm{Z}^{\tau}$—for example, equations $\mathrm{f}(\overline{U}, \mathrm{t\_}\sigma(X), \overline{V}) = \mathrm{f}(\overline{U}, X, \overline{V})$ to add or remove tags around well-typed arguments of a function f, or the idempotence law $\mathrm{t\_}\sigma(\mathrm{t\_}\sigma(X)) = \mathrm{t\_}\sigma(X)$—provided that they hold for canonical models (where the type tag is the identity function) and preserve monotonicity.

**Extension to Polymorphism.** The next step is to lift the argument to polymorphic encodings and polymorphic problems. Regrettably, it is not possible to adjust the above two-stage proof to polymorphism: Without dependent types, neither the binary g predicate nor the binary t function can be typed, preventing us from constructing the polymorphic intermediate problem corresponding to $\mathrm{Z}^{\tau}$.

Instead, we *reduce* the general, polymorphic case to the already proved monomorphic case. Our Herbrandian motto is: A polymorphic formula is equivalent to the set of its monomorphic instances, which in general will be an infinite set. This complete form of monomorphization is not to be confused with the finitary, heuristic monomorphization algorithm presented in Section 6.5.3. Our proof exploits a form of commutativity between our encodings and complete monomorphization.

More specifically, given a polymorphic problem $\mathrm{A}^{\alpha}$, the following two routes are possible (among others).

1. *Encode, then "monomorphize"*: Generate an untyped problem X from $\mathrm{A}^{\alpha}$ using a polymorphic encoding; then generate all possible "monomorphic"

$$\begin{tikzcd}
\mathrm{A}^\alpha \arrow[r, "\text{encode}"] \arrow[dd, "\text{mono-}\atop\text{morphize}"'] & \mathrm{X}^\tau \arrow[r, "\text{erase types}"] \arrow[d, "\text{``mono-}\atop\text{morphize''}"'] & \mathrm{X} \\
 & \mathrm{Y}^\tau \arrow[r, "\text{mangle}"] & \mathrm{Y} \\
\mathrm{A}^\tau \arrow[r, "\text{encode}"] & \mathrm{Z}^\tau \arrow[r, "\text{erase types}"] & \mathrm{Z}
\end{tikzcd}$$

**(a)** Transformations

$$\begin{tikzcd}
\mathrm{A}^\alpha \arrow[dd, leftrightarrow, "\text{Herbrand}"] & \mathrm{X}^\tau \arrow[r, squiggly, dash, "\text{Lem. 6.11}"] \arrow[d, leftrightarrow, "\text{Herbrand}"'] & \mathrm{X} \\
 & \mathrm{Y}^\tau \arrow[r, squiggly, dash, "\text{Lem. 6.16}"] & \mathrm{Y} \arrow[d, squiggly, dash, "\text{Lem. 6.17}"'] \\
\mathrm{A}^\tau \arrow[rr, squiggly, dash, "\text{Thms. 6.14, 6.15}"] & & \mathrm{Z}
\end{tikzcd}$$

**(b)** Equisatisfiability

**Figure 6.3:** Relationships between problems

instances of the problem's formulas by instantiating the encoded type variables with all possible "types" and mangle the resulting (generally infinite) set to obtain the problem Y. By our motto, X and Y are equisatisfiable.

2. *Monomorphize, then encode*: Compute the (generally infinite) set of monomorphic formulas $\mathrm{A}^\tau$ by instantiating all type variables in $\mathrm{A}^\alpha$ with all possible ground types; then translate $\mathrm{A}^\tau$ to Z using the mangled-monomorphic variant of the chosen encoding. $\mathrm{A}^\alpha$ and Z are equisatisfiable by Theorem 6.15 and our motto.

As in the monomorphic case, where we distinguished between the intermediate, typed problem $\mathrm{Z}^\tau$ and the final, untyped problem Z, we find it useful to oppose $\mathrm{X}^\tau$ to X and $\mathrm{Y}^\tau$ to Y. Although the protectors g and t cannot be typed polymorphically, a crude typing is possible, with encoded types assigned the type $\vartheta$ and all other terms assigned $\iota$. This avoids mixing types and terms in the encoded problems. The table below summarizes the situation:

| NAME | DESCRIPTION | LOGIC | EXAMPLE |
|---|---|---|---|
| $\mathrm{A}^\alpha$ | Original problem | TFF1 | $V^\alpha = \mathsf{c}$ |
| $\mathrm{X}^\tau$ | Encoded $\mathrm{A}^\alpha$ | TFF0 | $\mathsf{t}(A^\vartheta, V^\iota) = \mathsf{c}(A^\vartheta)$ |
| X | Type-erased $\mathrm{X}^\tau$ | FOF | $\mathsf{t}(A, V) = \mathsf{c}(A)$ |
| $\mathrm{Y}^\tau$ | "Monomorphized" $\mathrm{X}^\tau$ | TFF0 | $\mathsf{t}(\mathsf{b}^\vartheta, V^\iota) = \mathsf{c}(\mathsf{b}^\vartheta)$ |
| Y | Mangled $\mathrm{Y}^\tau$ | FOF | $\mathsf{t\_b}(V) = \mathsf{c\_b}$ |
| $\mathrm{A}^\tau$ | Monomorphized $\mathrm{A}^\alpha$ | TFF0 | $V^b = \mathsf{c}$ |
| $\mathrm{Z}^\tau$ | Encoded $\mathrm{A}^\tau$ | TFF0 | $\mathsf{t\_b}(V^b) = \mathsf{c\_b}$ |
| Z | Type-erased $\mathrm{Z}^\tau$ | FOF | $\mathsf{t\_b}(V) = \mathsf{c\_b}$ |

Figure 6.3(a) presents the situation visually.

Intuitively, the problems Y and Z obtained by following routes 1 and 2 should be very similar. If we can show that they are in fact equisatisfiable, the desired equisatisfiability of $\mathrm{A}^\alpha$ and X follows by transitivity, thereby proving g?, g??, t?, and t?? sound and complete. Figure 6.3(b) sketches the equisatisfiability proof. The missing equisatisfiabilities $\mathrm{Y}^\tau \sim \mathrm{Y} \sim \mathrm{Z}$ are proved below.

**Lemma 6.16 (Correctness of Type Mangling).** *The problems* $\mathrm{Y}^{\tau}$ *and* Y *are equisatisfiable.*

*Proof.* The difference between $\mathrm{Y}^{\tau}$ and Y is that the former has ground arguments of type $\vartheta$, while the latter mangles them into the symbol names—for example, $\mathsf{p}(\mathsf{b}^{\vartheta}, V^{\iota})$ vs. $\mathsf{p\_b}(V)$. Mangling is generally incomplete in an untyped logic; for example, the formula $X = Y \wedge \mathsf{q}(\mathsf{a}, U) \wedge \neg\, \mathsf{q}(\mathsf{b}, V)$ is unsatisfiable (since it implies $\mathsf{a} = \mathsf{b}$), but its mangled variant $X = Y \wedge \mathsf{q\_a}(U) \wedge \neg\, \mathsf{q\_b}(V)$ is satisfiable. In our two-typed setting, since there are no equations relating $\vartheta$ terms, mangling is easy to prove correct by considering (equality) Herbrand interpretations of the nonmangled and mangled formulas. □

**Lemma 6.17 (Commutativity of Encoding and Monomorphization).** *The problems* Y *and* Z *are equisatisfiable.*

*Proof.* We start with an example that illustrates the reasoning behind the proof. As polymorphic problem $\mathrm{A}^{\alpha}$, we simply take the polymorphic list axiom

$$\mathsf{hd}(\mathsf{cons}(X^{\alpha}, Xs)) = X$$

from Example 6.4. We suppose that *α list* is infinite (and hence monotonic) but the base type *b* is possibly nonmonotonic.

Following route 1, we apply the two-typed variant of t?? directly to the polymorphic formula $\mathrm{A}^{\alpha}$. This yields the set $\mathrm{X}^{\tau}$, where the second axiom below repairs mismatches between tagged and untagged terms with the infinite type *α list*:

$$\begin{array}{l}\mathsf{hd}(A, \mathsf{cons}(A^{\vartheta}, X^{\iota}, Xs^{\iota})) = \mathsf{t}(A, X)\\ \mathsf{t}(\mathsf{list}(A^{\vartheta}), Xs^{\iota}) = Xs\end{array}$$

This set would also contain a typing axiom for hd, which we omit here. The constant b and the unary function list are the only function symbols with a result of type $\vartheta$. Next, we instantiate the variables $A$ with all possible ground terms of type $\vartheta$ (of which there are infinitely many), yielding $\mathrm{Y}^{\tau}$. Finally, we mangle $\mathrm{Y}^{\tau}$, transforming $\mathsf{hd}(\mathsf{b}, t)$ into $\mathsf{hd\_b}(t)$ and so on. This gives Y:

$$\begin{array}{l}\mathsf{hd\_b}(\mathsf{cons\_b}(X, Xs)) = \mathsf{t\_b}(X)\\ \mathsf{hd\_list\_b}(\mathsf{cons\_list\_b}(X, Xs)) = \mathsf{t\_list\_b}(X)\\ \quad\vdots\\ \mathsf{t\_list\_b}(Xs) = Xs\\ \mathsf{t\_list\_list\_b}(Xs) = Xs\\ \quad\vdots\end{array}$$

In contrast, with route 2 we fully monomorphize $\mathrm{A}^{\alpha}$ to $\mathrm{A}^{\tau}$. Then we use a mangled-monomorphic encoding, say, $\widetilde{\mathsf{t}}$??, to translate it into a set Z of untyped formulas

$$\begin{array}{l}\mathsf{hd\_b}(\mathsf{cons\_b}(X, Xs)) = \mathsf{t\_b}(X)\\ \mathsf{hd\_list\_b}(\mathsf{cons\_list\_b}(X, Xs)) = X\\ \quad\vdots\end{array}$$

Notice that the treatment of $X$ in the right-hand sides above differs, since *b* is possibly nonmonotonic but *b list* is infinite.

Are Y and Z equisatisfiable? The first member of Y is also the first member of Z. The second formula of Y, however, does not appear in Z: the second formula of Z is the closest but its right-hand side is $X$ instead of $\mathsf{t\_list\_b}(X)$. Fortunately, Y also contains the axiom $\mathsf{t\_list\_b}(Xs) = Xs$, so Y must imply the second formula of Z. Conversely, problem Z does not mention the symbol $\mathsf{t\_list\_}\sigma$ for any $\sigma$, so we can add, for all ground types $\sigma$, the axiom $\mathsf{t\_list\_}\sigma(Xs) = Xs$ to Z while preserving satisfiability. This new set implies all members of Y—including the second formula—so Y and Z are equisatisfiable.

We now generalize the above argument. Y contains axioms $\mathsf{g\_}\sigma(X)$ or $\mathsf{t\_}\sigma(X) = X$ for each infinite type $\sigma$, whereas Z does not mention $\mathsf{g\_}\sigma$ or $\mathsf{t\_}\sigma$ for these types because they are monotonic; we can add the corresponding axioms to Z while preserving satisfiability. Otherwise, Y and Z contain the same formulas, except when $\mathrm{A}^{\alpha}$ quantifies over a variable $X$ of a possibly nonmonotonic type with an infinite instance $\sigma$. Z will not protect the $\sigma$ instances of $X$, but Y might; however, since $\sigma$ is infinite, Y must contain the axiom $\mathsf{g\_}\sigma(X)$ or $\mathsf{t\_}\sigma(X) = X$, allowing us to remove the guard or tag. Hence, the two sets of formulas are equisatisfiable. □

**Theorem 6.18 (Correctness of Polymorphic Encodings).** *The encodings* g?, g??, t?, *and* t?? *are sound and complete.*

*Proof.* This follows from Lemmas 6.11, 6.16, and 6.17, Theorems 6.14 and 6.15, and Herbrand's theorem (for terms and for types), as depicted in Figure 6.3(b). The application of Lemma 6.11 to erase $\vartheta$ and $\iota$ in $\mathrm{X}^{\tau}$ requires $\mathrm{X}^{\tau}$ to be monotonic; this can be proved either in the style of MONO in the proof of Theorem 6.14 or by observing that monotonicity is preserved along the equisatisfiability chain $\mathrm{Z}^{\tau} \sim \mathrm{Z} \sim \mathrm{Y} \sim \mathrm{Y}^{\tau} \sim \mathrm{X}^{\tau}$. □

## 6.6 Further Technical Improvements

This section describes various technical improvements that we made to Sledgehammer. These improvements have not always been of academic interest, but they are valuable to the users and maintainers.

### 6.6.1 Full First-Order Logic Output

In the previous Sledgehammer architecture, the available lemmas were rewritten to CNF using a naive exponential application of distributive laws before the relevance filter was invoked. To avoid clausifying thousands of lemmas on each Sledgehammer invocation, the CNF clauses were kept in a cache. This design was technically incompatible with the (cache-unaware) *smt* method, and it was already unsatisfactory for ATPs, which nowadays include custom clausifiers that generate a polynomial number of clauses [141].

We adjusted the relevance filter so that it operates on arbitrary HOL formulas, trying to simulate the old behavior. To mimic the penalty associated with Skolem functions in the CNF-based code, we keep track of polarities and detect quantifiers that give rise to Skolem functions.

A minor disadvantage of generating FOF formulas is that the Skolem constants introduced by the ATPs are applied directly (e.g., $\mathsf{y}(X)$) instead of via the application operator (e.g., $\mathsf{hAPP}(\mathsf{y}, X)$). As a result, the lemma

$$(\forall x.\ \exists y.\ p\ x\ y) \Longrightarrow \exists f.\ \forall x.\ p\ x\ (f\ x)$$

(a weak form of the HOL axiom of choice), which was previously translated to the inconsistent pair of clauses

$$\begin{array}{l}\mathsf{hBOOL}(\mathsf{hAPP}(\mathsf{hAPP}(P, X), \mathsf{hAPP}(\mathsf{y}, X))) \\ \neg\ \mathsf{hBOOL}(\mathsf{hAPP}(\mathsf{hAPP}(P, \mathsf{x}(F)), \mathsf{hAPP}(F, \mathsf{x}(F))))\end{array}$$

can no longer be proved by the ATPs without additional facts.

### 6.6.2 Fine-Tuned Relevance Filter

The relevance filter is controlled by a threshold, a convergence parameter, and a prover-dependent desired number of facts. We identified several issues: The number of selected facts varied greatly between problems, and the threshold and convergence parameters affected it in counterintuitive ways. No facts would be selected if the threshold was not reached in the first iteration. For first-order goals, all higher-order facts were crudely filtered out.

We started with the existing code base but made the selection more fluid by performing more iterations, each selecting fewer facts. If no facts are selected in the first iteration, the threshold is automatically lowered. We also made the desired number of facts a hard limit and tuned the other parameters so that it is normally reached. Inspired by the world's best-run financial institutions, we introduced a complex array of bonuses: Local facts are preferred to global ones, first-order facts are preferred to higher-order ones, and so on. These parameters were optimized by running the relevance filter with different values on a benchmark suite and comparing the output against a database of known proofs.

To help debugging, we ensured that the facts returned by the relevance filter are output in the order in which they were selected. We discovered later that this can be exploited by some provers:

- Z3 supports weights as extralogical annotations on universal quantifiers. The greater the weight of the quantifier, the fewer instantiations are allowed. We can give a weight of 0 to the quantifiers in the most relevant fact included, $N$ to the quantifiers in the least relevant fact, and interpolate in between.
- Motivated by Sledgehammer, Stephan Schulz extended E with two weight functions, `FunWeight` and `SymOffsetWeight`, that let us associate weights with function and predicate symbols. We give lower weights to the symbols that occur in the most relevant facts, so that clauses containing these symbols are preferred to other clauses.
- Also motivated by our application, Daniel Wand extended SPASS's input syntax with *ranks* attached to facts, indicating their likely relevance [36]. The ranks influence the clause selection strategy. This enhancement is expected to be part of SPASS 3.8.

### 6.6.3 Time Slicing

Some automatic provers, notably Vampire, use internal strategy scheduling when attacking a problem. Each strategy is given a time slice of at most a few seconds. Time slicing usually results in significantly higher success rates than letting an automatic prover run undisturbed for *n* seconds with a fixed strategy.

Applications can implement a form of time slicing for any automatic prover, by invoking it repeatedly with different options for a fraction of the total time limit. In Sledgehammer, we vary not only the prover options but also the number of facts and the type encodings. For E, SPASS, and Vampire, we computed optimal strategies of three 10-second slices based on exhaustive evaluations of the main type encodings and prover options. For example, the best combination of three slices we found for E were 50 facts with the $\widetilde{\mathtt{g}}$?? encoding and E's `FunWeight` clause weighting function, 500 facts with $\widetilde{\mathtt{t}}$?? and `FunWeight`, and 1000 facts with $\widetilde{\mathtt{t}}$?? and `SymOffsetWeight`.

### 6.6.4 Additional Provers

Our goal with Sledgehammer is to help as many Isabelle users as possible. Third-party provers should ideally be bundled with Isabelle and ready to be used without requiring configuration. Today, Isabelle includes CVC3, E, SPASS, and Z3 executables. Users can download Vampire and Yices, whose licenses forbid redistribution, but most simply run Vampire remotely on SystemOnTPTP [178].

We have undertaken experiments with several other ATPs:

- SInE, the Sumo Inference Engine [88], is a metaprover designed to cope with large axiom bases. SystemOnTPTP provides an older version as a wrapper for E. We pass E-SInE more facts than can be handled by the other ATPs, and it sometimes surprises us with original proofs.
- Waldmeister [87] is a highly optimized prover for the unit equality fragment of first-order logic. Its main strength is that it can form long chains of equational rewrites, reordering the given equations if necessary (unlike Isabelle's simplifier). Sledgehammer can use it if the goal is an unconditional equation.
- Instantiation provers such as Equinox [57] and iProver [104] are promising, but in case of success they currently do not deliver a proof, not even the list of used axioms.
- Z3 natively supports the TPTP FOF and TFF0 syntaxes. We can pretend it is an ATP and have it compete head-to-head with E, SPASS, and Vampire.

We already mentioned the SMT solvers CVC3, Yices, and Z3 in Section 6.3 and the higher-order ATPs LEO-II and Satallax in Section 6.5.3.

By default, Sledgehammer now runs E, E-SInE, SPASS, Vampire, Z3, and (for equational goals) Waldmeister in parallel, either locally or remotely via SystemOnTPTP. Remote servers are satisfactory for proof search, at least when they are up and running and the user has Internet access. They also help distribute the load: Unless the user's machine has eight processor cores, it would be reckless to launch six automatic provers locally and expect the user interface to remain responsive.

### 6.6.5 Fast Minimization

Proof minimization draws on either a naive linear algorithm or a binary algorithm (Section 6.2). Given an $n$-fact proof, the linear algorithm always needs $n$ calls to the external prover, whereas the binary algorithm requires anywhere between $\log_2 n$ and $2n$ calls, depending on how many facts are actually needed [44, §7.1]. We currently select the binary algorithm iff $n > 20$. In particular, the binary algorithm is used for provers that do not produce proofs or unsatisfiable cores, in which case it starts with the complete set of facts given to the prover.

Because of the multiple prover invocations (many if not most of which with unprovable problems), minimization often consumes more time than the proof search itself. An obvious improvement to the minimization algorithms is to inspect the proofs and eliminate any fact that is not referenced in it. Another improvement is to use the time required by the last successful proof as the timeout for the next one, instead of a fixed, necessarily liberal timeout (5 seconds). We have implemented these ideas in both algorithms. Together with a more detailed output, they have greatly contributed to making minimization fast and enjoyable.

The adaptations to Bradley and Manna's binary algorithm [47, §4.3] required some thought. Our new algorithm is presented below in a syntax close to Standard ML:

```
fun split [] lrs = lrs
  | split [x] (ls, rs) = (x :: ls, rs)
  | split (x1 :: x2 :: xs) (ls, rs) = split xs (x1 :: ls, x2 :: rs)

fun binary_minimize p t xs =
  let
    fun bm t sup (xs as _ :: _ :: _) =
        let val (l0, r0) = split xs ([], []) in
          case p t (sup @ l0) of
            Success (t, ys) ⇒ bm t (sup ∩ ys) (l0 ∩ ys)
          | Failure ⇒
            case p t (sup @ r0) of
              Success (t, ys) ⇒ bm t (sup ∩ ys) (r0 ∩ ys)
            | Failure ⇒
              let
                val (t, sup_r0, l) = bm t (sup @ r0) l0
                val sup = sup ∩ sup_r0
                val r0 = r0 ∩ sup_r0
                val (t, sup_l, r) = bm t (sup @ l) r0
                val sup = sup ∩ sup_l
              in (t, sup, l @ r) end
        end
      | bm t sup xs = (t, sup, xs)
  in
    case bm t [] xs of
      (t, _, [x]) ⇒ (case p t [] of Success _ ⇒ [] | Failure ⇒ [x])
    | (_, _, xs) ⇒ xs
  end
```

The main function, binary_minimize, takes the following arguments: a proving function $p$ that expects a timeout and a list of facts, a timeout $t$, and a list of facts $xs$. The result of $p$ is either Failure or Success $(t', xs')$, where $t' \leq t$ is the time taken to find the proof and $xs' \subseteq xs$ lists the referenced facts in the proof. If $t \neq \infty$, the proving function may return Failure when a Success value would be possible, in which case the result of minimization is not guaranteed to be minimal.

The inner function bm takes a timeout $t$, a support set $sup$, and a list of facts $xs$ to minimize, and returns a new triple $(t', sup', xs')$, where $xs'$ is the minimized set of facts, whereas $t' \leq t$ and $sup' \subseteq sup$ are a smaller timeout and a more precise support set to use for further calls to $p$.

### 6.6.6 Revamped User Experience

Sledgehammer was designed to run in the background, so that users can keep working on a manual proof while the automatic provers are active. However, what usually happened is that users waited for the tool to return, which took up to one minute. Since the vast majority of proofs are found in the first 30 seconds [44, §4], we halved the time limit. If no proof is found by then, Sledgehammer simply returns. Otherwise, it uses up to 30 seconds for postprocessing the proof; this is tolerable, because users tend to be patient when they know a proof has been found.

Postprocessing is a new concept that involves automatic proof minimization and proof preplay. If the proof search took $t$ seconds and yielded an $n$-fact proof, the linear algorithm can minimize it within $nt$ seconds. If $nt$ is below an acceptable threshold (currently 5 seconds), the proof is minimized behind the scenes before it is presented to the user. Automatic minimization also takes place if $n$ is absurdly large, which happens if the prover claims provability without giving any details.

Users waste precious time on *metis* calls that fail or take too long. Proof preplay addresses this by testing the *metis* call for a few seconds. In fact, three *metis* calls are tried in sequence, with different type encodings (a, g?, and e), followed by an *smt* call. The first call that succeeds is shown to the user together with the time it took. If several automatic provers independently found a proof, the user can choose the fastest one and insert it in the theory text.

With so many tools at their disposal, users run the risk of forgetting to invoke them at the right point. For this reason, Sledgehammer can now be configured to run automatically for a few seconds on all newly entered conjectures. It is also part of the **try** tool, which launches the main Isabelle proof and disproof tools with a more liberal time limit [30, §1].

In keeping with our belief that users should be given complete control to override default values and heuristics, we devised, implemented, and documented a rich option syntax [27], complemented by clear messages for common error scenarios. Many of the options cater for power users, allowing them to influence the facts to select, the $\lambda$-abstraction translation scheme or type encoding to use, and so on.

### 6.6.7 Skolemization without Choice

The step-by-step reconstruction of Metis proofs by the *metis* method is only possible for problems expressed in CNF. Since we cannot reasonably expect users to put their formulas in this form, it is the *metis* method's task to clausify the problem, via Isabelle's inference kernel.

The main difficulty here is simulating skolemization in Isabelle. A naive treatment of skolemization would require the introduction of fresh constants in the middle of a proof, which is technically impossible. The approach taken by Paulson and Susanto [153] when they implemented *metis* was to represent Skolem constants by fresh nonschematic variables and define them locally using so-called metahypotheses. A HOL lemma of the form $\forall x.\, \exists y^{\alpha}.\; P\; x\; y$ was transformed into

$$y^{\alpha} = (\lambda x.\, \varepsilon y.\; P\; x\; y) \;\vdash\; P\; x\; (y\; x)$$

where $\vdash$ separates the metahypothesis from the HOL clause. (Recall that $\varepsilon$ is the indefinite description operator, or Hilbert choice.) Once the proof by contradiction was completed, the metahypothesis could be removed.

However, because of restrictions in Isabelle's logical framework, variables introduced this way cannot have polymorphic types. In the formula above, if $\alpha$ is a schematic type variable, the schematicity is lost in the process. As a result, it was possible to use $\forall x.\, \exists y^{\alpha}.\; P\; x\; y$ to prove $\forall x.\, \exists y^{\alpha}.\; P\; x\; y$, but not to prove the equivalent formula $\forall x.\, \exists y^{\beta}.\; P\; x\; y$, with $\beta$ instead of $\alpha$.

The partial workaround in the old code was to introduce theory-level constants, which can be polymorphic, instead of nonschematic variables. This was done automatically for all the theorems of a theory at the end of the theory, and the resulting clauses were kept in a cache. It then became possible to prove $\forall x.\, \exists y^{\beta}.\; P\; x\; y$ from $\forall x.\, \exists y^{\alpha}.\; P\; x\; y$ if the latter belonged to an ancestor theory, but not if it had been introduced in the current, unfinished theory. The clause cache was also very inelegant and somewhat expensive in terms of time and memory, and as we mentioned in Section 6.6.1 it did not interact well with SMT solvers.

Our solution to these issues is to avoid the introduction of free variables or constants altogether and use the term $\lambda x.\, \varepsilon y.\; P\; x\; y$ directly instead. This only impacts the HOL side of things: When the clauses are translated for Metis, these $\varepsilon$ terms are replaced by fresh Skolem function symbols, and any occurrence of such a symbol in a Metis proof is mapped back to the corresponding $\varepsilon$ term in HOL. This change allowed us to eradicate the clause cache once and for all.

However, there are two drawbacks to this approach:

- For large formulas with many nested existential quantifiers, the HOL formulas explode in size and slow down *metis*. As a side effect, this prevents us from replacing the naive exponential clausifier by a polynomial approach based on definitional CNF, as this would introduce too many existential quantifiers.
- The reliance on the indefinite choice operator means that *metis* cannot be used in theories that are loaded before $\varepsilon$ is available, such as Isabelle's basic arithmetic theories. It also prevents the use of *metis* (and Sledgehammer) for other object logics, such as ZF [145, 146].

A solution to these issues was proposed by de Nivelle [67] for Coq. The idea is to recast Skolem functions into "Skolem predicates," which can be defined using the unproblematic definite description operator $\iota$ instead of $\varepsilon$. We considered this approach but found that Isabelle, with its schematic variables, offers an easier alternative. To keep the presentation simple, we restrict ourselves to the case where there is only one existential quantifier in one lemma, focusing first on the abstract problem before we review a specific example. In the interest of clarity, we distinguish schematics from nonschematics by a ? prefix in this discussion.

Our ideal would be to transform the lemma $\forall x.\ \exists y.\ P\ x\ y$ into the clause $P\ ?x\ (y\ ?x)$ where $y$ is some fresh constant or nonschematic variable, but this is denied to us: As we saw above, we cannot produce such a $y$ out of thin air. The intuition is to fall back on $P\ ?x\ (?y\ ?x)$, where $?y$ is schematic. This property generally does not hold unless $P$ is uniformly true; clearly, $?y$ must be constrained somehow. Indeed, the condition it must respect is precisely $P\ ?x\ (?y\ ?x)$, so we add that as an assumption and obtain the tautology

$$P\ ?x\ (?y\ ?x) \Longrightarrow P\ ?x\ (?y\ ?x)$$

This formula is trivial to derive, literally, using the aptly named kernel function $\mathsf{Thm.trivial}$. What interests us first is the conclusion $P\ ?x\ (?y\ ?x)$—the clause proper. In the translation for Metis, the assumption is ignored, the schematic variable $?x$ is mapped to a term variable, and $?y$ is mapped to a fresh Skolem function symbol. Once we have a Metis proof, we simulate it step by step, carrying the $P\ ?x\ (?y\ ?x)$ assumption along. At the end, when Metis has derived the empty clause, we are left with a theorem of the form

$$P\ t_1\ (?y_1\ t_1) \Longrightarrow \cdots \Longrightarrow P\ t_n\ (?y_n\ t_n) \Longrightarrow \mathsf{False}$$

where each assumption corresponds to a use of the clause $P\ ?x\ (?y\ ?x)$. In general, $?x$ is instantiated with some terms $t_j$, but $?y$ is left uninstantiated, since Metis sees it as a function symbol and not as the variable it really is. With some massaging, each assumption can be discharged using the original lemma $\forall x.\ \exists y.\ P\ x\ y$, resulting in the theorem $\mathsf{False}$ (with the negated conjecture as metahypothesis).

Let us make this more concrete by looking at an example. Assume we want to use the lemma

$$\forall x.\ x = 0 \lor \exists y.\ x = \mathsf{Suc}\ y$$

to prove the nearly identical goal

$$a = 0 \lor \exists b.\ a = \mathsf{Suc}\ b$$

Our clausifier first produces the three (generalized) HOL clauses

$$\begin{array}{cc} \multicolumn{2}{c}{?x = 0 \lor ?x = \mathsf{Suc}\ (?y\ ?x) \Longrightarrow ?x = 0 \lor ?x = \mathsf{Suc}\ (?y\ ?x)} \\ a \neq 0 & a \neq \mathsf{Suc}\ ?b \end{array}$$

The first clause corresponds to the lemma, whereas the last two are derived from the negated conjecture. Because of the negation, the existential variable $b$ is essentially universal and becomes a schematic variable $?b$. Inversely, the nonschematic variable $a$ is effectively a Skolem constant.

The HOL formulas are translated to the first-order clauses

$$X = \mathsf{zero} \lor X = \mathsf{suc}(\mathsf{y}(X)) \qquad \mathsf{a} \neq \mathsf{zero} \qquad \mathsf{a} \neq \mathsf{suc}(B)$$

where $\mathsf{y}$ is a Skolem function. Metis finds a contradiction by taking $X = \mathsf{a}$ and $B = \mathsf{y}(\mathsf{a})$, and the needed resolution steps are easy to emulate in Isabelle. Because the first clause has an additional assumption, at the end of the resolution proof we have derived the following HOL theorem:

$$a = 0 \lor a = \mathsf{Suc}\ (?y\ a) \Longrightarrow \mathsf{False}$$

Let us call this intermediate result *nearly_there*. Notice that $?x$ in the assumption is instantiated with $a$ (since Metis took $X = \mathsf{a}$), whereas $?y$ is left uninstantiated.

To conclude the proof, we must derive False from *nearly_there*. This is convoluted. To give the general flavor of the approach, we express it in the Isar syntax, but naturally it is implemented as a general ML tactic. The current goal state is displayed between each pair of Isar commands:

**show** False

1. False

**apply** (*insert* '$\forall x.\ \exists y.\ x = 0 \lor x = \mathsf{Suc}\ y$')

1. $\forall x.\ \exists y.\ x = 0 \lor x = \mathsf{Suc}\ y \Longrightarrow \mathsf{False}$

**apply** (*erule_tac* $x = a$ in *allE*)

1. $\exists y.\ a = 0 \lor a = \mathsf{Suc}\ y \Longrightarrow \mathsf{False}$

**apply** (*erule exE*)

1. $\bigwedge y.\ a = 0 \lor a = \mathsf{Suc}\ y \Longrightarrow \mathsf{False}$

**apply** (*rule nearly_there*)

1. $\bigwedge y.\ a = 0 \lor a = \mathsf{Suc}\ y \Longrightarrow a = 0 \lor a = \mathsf{Suc}\ (?y\ y\ a)$

**by** *assumption*

In a first step, we insert the prenex normal form of our lemma as an assumption in the proof goal. Then we eliminate the quantifier prefix, by instantiating the universal quantifier and transforming the existential into a schematic variable, applying the appropriate HOL elimination rules (*allE* and *exE*). We are left with a trivial implication modulo the unification goal $y \overset{?}{=} ?y\ y\ a$. This unification is no match for Isabelle's higher-order unifier, which finds the solution $?y = (\lambda y\ a.\ y)$.

The approach is generalizable to arbitrary quantifier prefixes, but it requires careful synchronization when discharging the assumptions. In addition, we cannot always expect the higher-order unifier to succeed and must perform the required instantiations explicitly. Since the new approach works well with definitional CNF and Isabelle already has some support for it (developed for the SAT solver integration [198, §6]), we now have a clausifier that generates a polynomial number of clauses in the worst case, making it possible for *metis* to reconstruct proofs that were previously beyond its effective reach.

## 6.7 Evaluation

According to Lawrence Paulson [150]:

> Experimental science is about empirical results. You set up an experiment, you run it and you report what happened. In computer science, we seem to have turned things around, so that the mere act of building a big system counts as experimental science. In my opinion, it isn't.

We wholeheartedly agree. Our changes to Sledgehammer might not be improvements, unless we can exhibit convincing empirical evidence that the tool is now more powerful than it was before our involvement with it.

Ideally, we would show the contribution of each improvement in isolation to the overall result. This is technically difficult, because in parallel with Sledgehammer's development, Isabelle itself, its lemma libraries, the evaluation framework (Mirabelle), the theories used for the evaluation, and the third-party automatic provers kept evolving. Nonetheless, we can get a fairly accurate idea by tracking the evaluations presented in three papers:

- In their original "Judgment Day" study, conducted in November 2009 before we took over Sledgehammer's development, Böhme and Nipkow [44] evaluated E, SPASS, and Vampire in parallel on 1240 proof goals arising in seven representative Isabelle theories. They found an overall success rate of 47% for a 30 second timeout and 48% for 120 seconds.
- In September 2010, we ran the Judgment Day benchmark suite on the same hardware but with more recent versions of the software and theories [152]. In particular, Sledgehammer now communicated with the ATPs using FOF instead of CNF, added E-SInE to the collection of ATPs, and employed the latest versions of SPASS and Vampire. The success rate had climbed from 48% to 52% for a 120 second timeout.
- In April 2011, we ran a superset of the Judgment Day benchmarks after a further round of improvements, including the addition of SMT solvers [28]. On the Judgment Day benchmarks, the success rate with a 30 second timeout was 52% for the ATPs (E, E-SInE, SPASS, and Vampire), 55% for the SMT solvers (CVC3, Yices, and Z3), and 60% for all provers combined.

Here we evaluate the latest version of Sledgehammer and the automatic provers, comparing the performance of the automatic provers and their unique contributions (Section 6.7.4). But first, we present two smaller evaluations that focus on specific aspects of the translation to first-order logic: the encoding of polymorphic types (Section 6.7.2) and the translation of $\lambda$-abstractions (Section 6.7.3).

### 6.7.1 Experimental Setup

The Judgment Day benchmarks consist of all the proof goals arising in seven theories from the Isabelle distribution and the *Archive of Formal Proofs* [102], listed below under short names. The third column lists the number of goals from each theory, and the last column specifies the features it contains, where A means arithmetic, I means induction/recursion, L means $\lambda$-abstractions, and S means sets.

| Thy. | Description | Goals | Feats. |
|---|---|---|---|
| *Arr* | Arrow's impossibility theorem | 101 | L S |
| *FFT* | Fast Fourier transform | 146 | A L |
| *FTA* | Fundamental theorem of algebra | 424 | A |
| *Hoa* | Completeness of Hoare logic with procedures | 203 | A I L |
| *Jin* | Type soundness of a subset of Java | 182 | I L |
| *NS* | Needham–Schroeder shared-key protocol | 100 | I |
| *SN* | Strong normalization of the typed $\lambda$-calculus | 115 | A I |

We added two theories (*QE* and *S2S*) that rely heavily on arithmetic to exercise the SMT decision procedures and a third one (*Huf*) that combines all four features:

| | | | |
|---|---|---|---|
| *QE* | DNF-based quantifier elimination | 193 | A LS |
| *S2S* | Sum of two squares | 130 | A |
| *Huf* | Optimality of Huffman's algorithm | 284 | AILS |

We believe these ten theories to be representative of typical applications of Isabelle, with a bias toward arithmetic reasoning.

We used the following prover versions as backends: CVC3 2.2, E 1.4, E-SInE 0.4, LEO-II 1.2.8, Satallax 2.2, SPASS 3.7, Vampire 1.8, Waldmeister 710, Yices 1.0.28, and Z3 3.0. E-SInE and Waldmeister were run remotely via SystemOnTPTP [178]; the other provers were installed and run on Böhme and Nipkow's 32-bit Linux servers (with 3.06 GHz Dual-Core Intel Xeon processors). For the evaluations of the translation of types and $\lambda$-abstractions, we invoked each prover once per problem, with a fixed set of options. For some provers, we deviated from the default setup, based on earlier experiments:

- The E setup was suggested by Stephan Schulz and included the `SymOffsetWeight` weight function (Section 6.6.2).
- We invoked LEO-II with `-sos`.
- We invoked SPASS with `-Auto`, `-SOS=1`, `-VarWeight=3`, and `-FullRed=0`.
- We invoked Vampire with `-mode casc`, `-forced_options propositional_to_bdd=off`, and `-thanks "Andrei and Krystof"`.
- We invoked Z3 with `MBQI=true` for TPTP problems.

For evaluating the combination of provers, we enabled time slicing (Section 6.6.3). Each time slice of each prover then specifies its own options.

### 6.7.2 Type Encodings

To evaluate the type encodings described in Section 6.5, we put together two sets of 1000 problems generated from the ten selected Isabelle theories, with 100 problems per theory, using combinators to encode $\lambda$-abstractions. The problems in the first benchmark set include up to 150 heuristically selected axioms (before monomorphization); that number is increased to 500 for the second set, to reveal how well the encodings scale with the problem size.

We evaluated the main type encodings with the resolution provers E, SPASS, and Vampire and the SMT solver Z3 in TPTP mode. The provers were granted 20 seconds of CPU time per problem. Most proofs were found within a few seconds; a higher time limit would have had little impact on the success rate [44]. To avoid giving the unsound encodings an unfair advantage, the proof search phase is followed by a certification phase that attempts to re-find the proof using a combination of sound encodings.

Figures 6.4 and 6.5 give, for each combination of prover and encoding, the number of solved problems from each set. Rows marked with ~ concern the mangled-monomorphic encodings. For the second benchmark set, Figure 6.6 presents the average number of clauses, literals per clause, symbols per atom, and symbols for clausified problems (using E's clausifier), to measure each encoding's overhead.

The monomorphic versions of our refined monotonicity-based translation scheme, especially g̃!! and g̃??, performed best overall. This confirms the intuition that clutter (whether type arguments, guards, or tags) slows down automatic provers. Surprisingly, some of our monomorphic encodings outperformed Vampire's and Z3's native types (ñ? and ñ). Part of the explanation is that the type support in Vampire 1.8 is unsound (leading to many rejected proofs) and interferes with the prover's internal strategy scheduling.

Polymorphic encodings lag behind, but our approach nonetheless constitutes a substantial improvement over the traditional polymorphic schemes. The best unsound encodings performed very slightly better than the best sound ones, but not enough to justify making them the default.

### 6.7.3 Translation of $\lambda$-Abstractions

Orthogonally to the encoding of types, it is important to choose an appropriate translation of $\lambda$-abstractions. In Section 6.4.3, we presented two main translation schemes into first-order logic—the Curry combinators and $\lambda$-lifting—as well as a hybrid scheme. For good measure, we also evaluate two more radical translation schemes: One consists of "hiding" the $\lambda$-abstractions by replacing them by unspecified fresh constants, effectively disabling all reasoning under $\lambda$-abstractions; the other, which is available only with the higher-order ATPs, is simply to keep the $\lambda$s in the generated THF0 problems.

We evaluated these translation schemes for the first-order resolution provers E, SPASS, and Vampire with the type encoding g̃??, the SMT solver Z3 in TPTP mode with ñ, and the higher-order provers LEO-II and Satallax with Ñ. The provers were given 30 seconds per problem. For each combination of prover and translation scheme, we generated 1000 problems from the ten selected Isabelle theories, with 100 problems per theory. The first-order provers were passed 300 axioms; the (less scalable) higher-order provers were passed only 100 axioms. Figure 6.7 gives the number of solved problems for each set of 1000 problems.

| | UNSOUND | | | | | | SOUND | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | e | a | g!! | t!! | g! | t! | g?? | t?? | g? | t? | g | t | n? | n |
| E | 316 | 362 | 350 | 353 | 343 | 338 | 344 | 351 | 345 | 302 | 255 | 295 | – | – |
| ~ | – | 344 | ***390*** | 389 | 382 | 372 | 388 | ***390*** | 386 | 373 | 355 | 334 | – | – |
| SPASS | 275 | 324 | 305 | 308 | 309 | 293 | 291 | 309 | 290 | 242 | 247 | 267 | – | – |
| ~ | – | 267 | ***337*** | 334 | 334 | 322 | ***344*** | 341 | 340 | 333 | 321 | 311 | – | – |
| VAMPIRE | 291 | 376 | 328 | 326 | 333 | 333 | 331 | 313 | 325 | 294 | 240 | 211 | – | – |
| ~ | – | 357 | ***385*** | 368 | 376 | 379 | ***381*** | 374 | 365 | 364 | 303 | 238 | 366 | 376 |
| Z3 | 295 | 365 | 345 | 347 | 328 | 313 | 329 | 333 | 307 | 260 | 253 | 319 | – | – |
| ~ | – | 339 | ***366*** | 360 | 362 | 362 | 353 | 352 | 356 | 348 | 349 | 314 | ***364*** | 361 |

**Figure 6.4:** Number of solved problems with 150 facts

| | Unsound | | | | | | Sound | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | e | a | g!! | t!! | g! | t! | g?? | t?? | g? | t? | g | t | n? | n |
| E | 193 | 339 | 309 | 308 | 312 | 300 | 313 | 310 | 304 | 245 | 199 | 216 | – | – |
| ~ | – | 309 | ***368*** | 365 | 363 | 354 | 364 | ***366*** | 364 | 350 | 314 | 308 | – | – |
| SPASS | 186 | 306 | 287 | 285 | 284 | 274 | 274 | 294 | 264 | 204 | 198 | 243 | – | – |
| ~ | – | 247 | 310 | 304 | ***312*** | 284 | ***308*** | 306 | 300 | 276 | 239 | 231 | – | – |
| Vampire | 180 | 350 | 282 | 292 | 298 | 286 | 282 | 288 | 283 | 279 | 185 | 148 | – | – |
| ~ | – | 315 | 344 | 344 | ***354*** | 334 | 339 | 342 | 339 | 331 | 251 | 171 | ***355*** | 346 |
| Z3 | 175 | 339 | 319 | 323 | 311 | 296 | 306 | 303 | 262 | 222 | 200 | 219 | – | – |
| ~ | – | 302 | 343 | 343 | ***350*** | 337 | 343 | 340 | ***344*** | 327 | 329 | 254 | 335 | 337 |

**Figure 6.5:** Number of solved problems with 500 facts

| | Unsound | | | | | | Sound | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Avg. Num. | e | a | g!! | t!! | g! | t! | g?? | t?? | g? | t? | g | t |
| Clauses | ***749*** | 808 | 896 | 835 | 896 | 835 | 974 | 867 | 974 | 867 | 1107 | ***827*** |
| ~ | – | 1080 | 1139 | 1105 | 1139 | 1105 | 1176 | 1105 | 1176 | 1105 | 1649 | 1105 |
| Literals | 2.32 | 2.55 | 2.70 | 2.60 | 3.00 | 2.55 | 2.84 | 2.59 | 3.88 | 2.49 | 5.41 | 2.56 |
| per Clause ~ | – | 2.28 | 2.35 | ***2.22*** | 2.35 | 2.29 | 2.57 | ***2.15*** | 2.57 | 2.29 | 4.74 | 2.29 |
| Symbols | ***6.7*** | 9.3 | 15.8 | 16.6 | 14.5 | 18.3 | 14.8 | 16.4 | 11.9 | 25.6 | 6.3 | 28.1 |
| per Atom ~ | – | 7.2 | 6.9 | 7.4 | 7.0 | 8.0 | 6.3 | 7.6 | 6.5 | 8.3 | ***4.4*** | 14.0 |
| Symbols | ***11.7*** | 19.2 | 38.2 | 38.6 | 39.0 | 38.9 | 40.9 | 41.5 | 45.1 | 55.2 | 38.0 | 59.5 |
| ('000) ~ | – | 17.8 | 18.5 | 18.6 | 18.7 | 20.1 | ***19.0*** | 19.2 | 19.6 | 21.0 | 34.4 | 35.3 |

**Figure 6.6:** Average size of clausified problems with 500 facts

| | Hidden | Combs. | Lifting | Hybrid | Lambdas |
|---|---|---|---|---|---|
| E | 373 | 393 | 390 | ***394*** | – |
| SPASS | 338 | 345 | ***347*** | 325 | – |
| Vampire | 391 | 403 | 411 | ***415*** | – |
| Z3 | 342 | ***367*** | 357 | 360 | – |
| LEO-II | 209 | 196 | ***221*** | 207 | 205 |
| Satallax | 137 | 138 | 137 | 133 | ***139*** |
| All Provers | 446 | 482 | 478 | ***489*** | – |

**Figure 6.7:** Solved problems per prover and $\lambda$-abstraction translation scheme

These results show that it is appropriate to try to encode $\lambda$-abstractions—just hiding them does not help. On the benchmarks, E and Z3 preferred combinators, whereas SPASS and Vampire preferred lifting. The hybrid scheme worked well for E and Vampire but had a terrible impact on SPASS, for which it performed even worse than simply hiding the $\lambda$s.

As expected, Satallax performs about as well if the $\lambda$s appear as such in the problem or if they are lifted. (In higher-order logic, $\lambda$-lifting is a simple syntactic transformation that does not affect provability.) On the other hand, the $\lambda$s give LEO-II more than enough rope to hang itself.

Based on these results, we have now made the hybrid scheme the default for Vampire and $\lambda$-lifting for SPASS and LEO-II. However, we suspect our first-order translation schemes are far from optimal. We would need to implement and evaluate more elaborate translation schemes—such as Obermeyer's combinator-based approach [142] and explicit substitutions [69]—to gain clarity on this point, and perhaps choose a more appropriate set of benchmarks.

### 6.7.4 Combination of Automatic Provers

Last but not least, we applied the main ATPs and SMT solvers supported by Sledgehammer to all 1876 goals from the ten selected theories with a 30 second time limit, followed in case of success by a 30 second postprocessing phase and a 30 second proof reconstruction phase.

| | *FFT* | *QE* | *Huf* | *NS* | *Jin* | *Arr* | *Hoa* | *SN* | *FTA* | *S2S* | All | Uniq. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | ***25*** | ***30*** | 30 | 27 | 46 | 46 | 44 | 68 | ***67*** | 55 | 45.8 | 1.1 |
| E-SInE | 20 | 30 | 22 | 22 | 18 | 29 | 42 | 57 | 60 | 52 | 37.5 | .3 |
| SPASS | 18 | 26 | 30 | 31 | 41 | 44 | 50 | 67 | 62 | 50 | 43.6 | .5 |
| Vampire | 19 | 28 | 25 | 27 | 43 | 37 | 44 | ***70*** | 67 | 48 | 43.2 | .4 |
| Waldmeister | 4 | 6 | 2 | 4 | 1 | 0 | 0 | 1 | 6 | 2 | 3.1 | .0 |
| LEO-II | 13 | 13 | 20 | 7 | 23 | 7 | 25 | 43 | 48 | 35 | 27.0 | .1 |
| Satallax | 3 | 7 | 6 | 2 | 14 | 10 | 25 | 30 | 25 | 22 | 15.6 | .1 |
| CVC3 | 21 | 27 | 31 | 23 | 37 | 52 | ***56*** | 53 | 57 | ***62*** | 43.1 | .4 |
| Yices | 18 | 27 | 24 | 27 | 42 | 36 | 50 | 55 | 54 | 58 | 40.2 | .2 |
| Z3 | 24 | 25 | ***33*** | ***41*** | ***50*** | ***55*** | 54 | 62 | 62 | 58 | ***47.2*** | ***2.0*** |
| F.-O. ATPs | 28 | 33 | 36 | 40 | 47 | 50 | 54 | 74 | 71 | 65 | 51.3 | 5.3 |
| H.-O. ATPs | 14 | 16 | 21 | 7 | 27 | 15 | 35 | 50 | 51 | 37 | 30.5 | .1 |
| SMT Solvers | 29 | 28 | 36 | 44 | 50 | 62 | 65 | 63 | 68 | 67 | 51.9 | 6.8 |
| All Provers | 34 | 41 | 42 | 46 | 54 | 66 | 70 | 74 | 75 | 76 | 58.8 | – |

**Figure 6.8:** Success rates (%) on all goals per prover and theory

Figure 6.8 gives the success rate for each prover (or class of prover) on each theory together with the unique contributions of each prover. Sledgehammer solves 58.8% of the goals, compared with 52.0% without SMT. Much to our surprise, the best SMT solver, Z3, beats the best ATP, E, with 47.2% versus 45.8%. Z3 also contributes by far the most unique proofs: 2.0% of the goals are proved only by it, a figure that climbs to 5.9% if we exclude CVC3 and Yices.

While it might be tempting to see this evaluation as a direct comparison of provers, recall that even provers of the same class (first-order ATP, higher-order ATP, or SMT solver) are not given the same number of facts or the same options. Sledgehammer is not so much a competition as a *combination* of provers.

If we consider only the seven Judgment Day theories, the success rate is 56.2% for the first-order ATPs, 34.2% for the higher-order ATPs, 57.5% for the SMT solvers, and 63.6% for all provers combined. This is a substantial improvement over the success rate of 47% obtained by Böhme and Nipkow (for E, SPASS, and Vampire).

| | *FFT* | *QE* | *Huf* | *NS* | *Jin* | *Arr* | *Hoa* | *SN* | *FTA* | *S2S* | All | Uniq. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | ***20*** | ***17*** | 19 | 14 | 41 | 44 | 27 | 56 | ***44*** | 21 | 29.6 | 1.7 |
| E-SInE | 15 | ***17*** | 13 | 8 | 18 | 24 | 27 | 53 | 37 | 16 | 22.4 | .4 |
| SPASS | 15 | 17 | 21 | 21 | 36 | 37 | 35 | 56 | 37 | 25 | 28.7 | .9 |
| Vampire | 15 | 14 | 17 | 16 | 39 | 31 | 30 | ***59*** | 44 | 16 | 27.7 | .7 |
| Waldmeister | 5 | 5 | 1 | 0 | 1 | 0 | 0 | 0 | 7 | 0 | 2.4 | .0 |
| LEO-II | 11 | 4 | 14 | 0 | 18 | 3 | 14 | 34 | 24 | 9 | 14.0 | .1 |
| Satallax | 0 | 3 | 3 | 0 | 16 | 2 | 13 | 25 | 9 | 5 | 7.4 | .1 |
| CVC3 | 16 | 13 | 20 | 10 | 32 | 42 | 43 | 37 | 37 | 28 | 27.6 | .7 |
| Yices | 13 | 13 | 15 | 17 | 37 | 21 | 39 | 41 | 32 | 23 | 24.9 | .3 |
| Z3 | 19 | 10 | ***24*** | ***32*** | ***48*** | ***45*** | ***46*** | 46 | 44 | ***28*** | ***33.6*** | ***3.2*** |
| F.-O. ATPs | 23 | 21 | 26 | 30 | 41 | 45 | 40 | 64 | 51 | 35 | 36.3 | 5.8 |
| H.-O. ATPs | 11 | 7 | 14 | 0 | 25 | 5 | 19 | 36 | 25 | 9 | 16.0 | .2 |
| SMT Solvers | 24 | 13 | 26 | 36 | 48 | 52 | 54 | 47 | 46 | 32 | 37.0 | 7.5 |
| All Provers | 28 | 23 | 32 | 38 | 50 | 56 | 60 | 64 | 57 | 47 | 44.3 | – |

**Figure 6.9:** Success rates (%) on “nontrivial” goals per prover and theory

About 40% of the goals from the chosen Isabelle theories are “trivial” in the sense that they can be solved directly by standard Isabelle tactics invoked with no arguments. If we ignore these and focus on the remaining 1144 “nontrivial” goals, which users are especially keen on seeing solved by Sledgehammer, the success rates are lower, as shown in Figure 6.9: The ATPs solve 36.8% of these harder goals, and SMT solvers increase the success rate to 44.3%. In contrast, the success rate on the “trivial” goals for all provers combined is about 81%.

It may surprise some that E outperforms Vampire with such a clear margin, when the latter has been winning the FOF division of CASC for the last 10 years. The explanation is that Sledgehammer greatly benefits from E’s support for weight functions, a nonstandard feature with no equivalent in Vampire.

Since LEO-II relies on E for its first-order reasoning and most of our problems are either first-order or very mildly higher-order, we could have expected similar results for both provers. The wide gap between LEO-II and E is likely attributable to its inefficient encoding of types [18] and could be addressed by adopting one of our monotonicity-based encodings.

We expected Satallax to perform at least as well as LEO-II, but two factors worked against it: Its internal strategy scheduling is calibrated for a much higher time limit than 30 seconds, and a debilitating bug on Linux leads it to find spurious proofs, which are then rejected by its unsatisfiable core extractor.

## 6.8 Structured Proof Construction

Proofs found by ATPs are reconstructed in Isabelle either using a single *metis* call or as Isar proofs following the structure of the ATP proofs [153]. The latter option is useful for more difficult proofs, which *metis* fails to re-find within a reasonable time. But most users find the generated Isar proofs unpalatable and are little inclined to insert them in their theory text.

As illustration, consider the conjecture length (tl *xs*) $\leq$ length *xs*, which states that the tail of a list (the list from which we remove its first element, or the empty list if the list is empty) is at most as long as the original list. The proof found by Vampire, translated to Isar, is as follows:

```
proof neg_clausify
  assume length (tl xs) ≰ length xs
  hence drop (length xs) (tl xs) ≠ [] by (metis drop_eq_Nil)
  hence tl (drop (length xs) xs) ≠ [] by (metis drop_tl)
  hence ∀u. xs @ u ≠ xs ∨ tl u ≠ [] by (metis append_eq_conv_conj)
  hence tl [] ≠ [] by (metis append_Nil2)
  thus False by (metis tl.simps(1))
qed
```

The *neg_clausify* proof method puts the Isabelle conjecture into negated clause form, ensuring that it has the same shape as the corresponding ATP conjecture. The negation of the clause is introduced by the **assume** keyword, and a series of intermediate facts introduced by **hence** lead to a contradiction.

There is a considerable body of research about making resolution proofs readable. Earlier work focused on translating detailed resolution proofs into natural deduction calculi [128, 155]. Although they are arguably more readable, these calculi still operate at the logical level, whereas humans reason mostly at the assertion level, invoking definitions and lemmas without providing the full logical details. A line of research focused on transforming natural deduction proofs into assertion-level proofs [6, 89], culminating with the systems TRAMP [122] and Otterfier [208].

We would have liked to try out TRAMP and Otterfier, but these are large pieces of unmaintained software that are hardly installable on modern machines and that only support older ATPs. Regardless, the problem looks somewhat different in the context of Sledgehammer. Because the ATPs are given hundreds of lemmas, they tend to find short proofs, typically involving only a handful of lemmas. Moreover, Sledgehammer can be instructed to merge each sequence of *n* inferences if short proofs are desired. Replaying the inference is a minor issue, thanks to *metis*.

The first obstacle to readability is that the Isar proof, like the underlying ATP proof, is by contradiction. As a first step toward more intelligible proofs, we looked for a method to turn contradiction proofs around. From the onset, we decided that the method should not be tied to any one logic (as long as it is classical) or calculus. In particular, it should work on the Isar proofs generated by Sledgehammer or directly on first-order TPTP/TSTP proofs [184]. The direct proof should be expressible in Isar, Tisar (our ad hoc extension of TPTP/TSTP with Isar-like case analysis and nested subproofs), or similar block-structured languages.

We present a method for transforming resolution proofs into direct proofs, or *redirecting* resolution proofs, in a simple Isar-like syntax (Section 6.8.1). We demonstrate the method on a few examples (Section 6.8.2) before we present the main algorithm (Section 6.8.3).

For examples with a simple linear structure, such as the Isar proof above, the proof can be turned around by applying contraposition repeatedly:

```
proof –
  have tl [] = [] by (metis tl.simps(1))
  hence ∃u. xs @ u = xs ∧ tl u = [] by (metis append_Nil2)
  hence tl (drop (length xs) xs) = [] by (metis append_eq_conv_conj)
  hence drop (length xs) (tl xs) = [] by (metis drop_tl)
  thus length (tl xs) ≤ length xs by (metis drop_eq_Nil)
qed
```

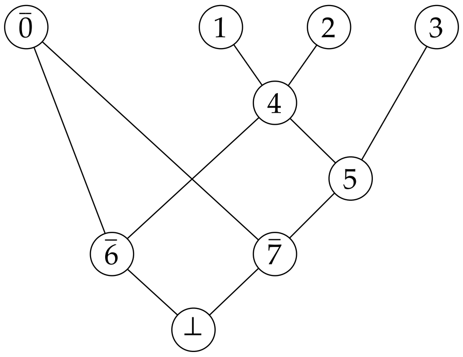
Our approach works on arbitrary proofs by contradiction. An early prototype demonstrated at PAAR-2010 [151] and IWIL-2010 [152] sometimes exhibited exponential behavior. This issue has been resolved: Each step in the resolution proof now gives rise to exactly one step in the direct proof. A linear number of additional steps are introduced in the direct proof, but these correspond to applications of basic logical rules, such as modus ponens. Charles Francis implemented a prototype as a student project, and the algorithm is now part of Sledgehammer.

While the output is designed for replaying proofs, it also has a pedagogical value: Unlike Isabelle's automatic tactics, which are black boxes, the proofs delivered by Sledgehammer can be inspected and understood. The direct proof also forms a good basis for further development.

### 6.8.1 Proof Notations

We first fix some notations for representing proofs.

**Proof Graph.** A proof graph is a directed acyclic graph where an edge $a \rightarrow b$ indicates that $a$ is used to derive $b$. We adopt the convention that derived nodes appear lower than their parent nodes in the graph and omit the arrowheads:



ATP proofs identify formulas by numbers: The conjecture is called 0, the axioms are numbered $1, 2, \ldots, n$, and the derivations performed during proof search (whether or not they participate in the final proof) are numbered sequentially from $n+1$. We abstract the ATP proofs by ignoring the formulas and keeping only the numbers. Since we are not interested in the internal structure of the formulas found in the ATP proofs, we call them *atoms*.

An atom is *tainted* if it is the negated conjecture or one of its direct or indirect consequences. For our convenience, we decorate tainted atom numbers with a bar, denoting negation. Removing the bar unnegates the conjecture but negates the other tainted atoms. For the last step, we write $\bot$ rather than $\overline{\top}$.

**Proof Trees.** Proof trees are the standard notation for natural deduction proofs. For example, the proof graph above is captured by the following proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\overline{0}] \qquad \dfrac{\dfrac{1 \quad 2}{1 \wedge 2} \quad 1 \wedge 2 \longrightarrow 4}{4}}{\overline{0} \wedge 4} \qquad \overline{0} \wedge 4 \longrightarrow \overline{6}}{\overline{6}} \qquad \dfrac{\dfrac{[\overline{0}] \qquad \dfrac{\dfrac{3 \qquad \dfrac{\dfrac{1 \quad 2}{1 \wedge 2} \quad 1 \wedge 2 \longrightarrow 4}{4}}{3 \wedge 4} \qquad 3 \wedge 4 \longrightarrow 5}{5}}{\overline{0} \wedge 5} \qquad \overline{0} \wedge 5 \longrightarrow \overline{7}}{\overline{7}}}{\overline{6} \wedge \overline{7}} \qquad \overline{6} \wedge \overline{7} \longrightarrow \bot}{\bot}$$

The implications, such as $1 \wedge 2 \longrightarrow 4$, are tautologies that should be proved as well. The proof graph notation is more compact, not least because it enables the sharing of subproofs; consequently, we prefer proof graphs to proof trees.

**Isar Proofs.** Isar proofs [139, 202] are a linearization of natural deduction proofs, but unlike proof trees they allow the sharing of common derivations:

```
proof neg_clausify
  assume $\overline{0}$
  have 4 by (metis 1 2)
  have 5 by (metis 3 4)
  have $\overline{6}$ by (metis $\overline{0}$ 4)
  have $\overline{7}$ by (metis $\overline{0}$ 5)
  show $\bot$ by (metis $\overline{6}$ $\overline{7}$)
qed
```

The above proof is by contradiction. The corresponding direct proof relies on a 2-way case analysis:

```
proof –
  have 4 by (metis 1 2)
  have 5 by (metis 3 4)
  have 6 ∨ 7 by metis
  moreover
  { assume 6
    have 0 by (metis 4 6) }
  moreover
  { assume 7
    have 0 by (metis 5 7) }
  ultimately show 0 by metis
qed
```

**Shorthand Proofs.** The last proof format is an ad hoc shorthand notation for a subset of Isar. In their simplest form, these shorthand proofs are a list of derivations

$$a_1, \ldots, a_m \;\rhd\; b_1 \vee \cdots \vee b_n$$

whose intuitive meaning is that "from the hypotheses $a_1$ and … and $a_m$, we conclude $b_1$ or … or $b_n$." Both the commas on the left-hand side and the disjunctions on the right-hand sides are associative, commutative, and idempotent. If a hypothesis $a_i$ is the previous derivation's conclusion, we can omit it and write $\blacktriangleright$ instead of $\rhd$. This notation mimics Isar, with $\rhd$ for **have** or **show** and $\blacktriangleright$ for **hence** or **thus**. Depending on whether we use the abbreviated format, our running example becomes

$$\begin{array}{rcl@{\qquad\qquad\qquad}rcl}
1,2 & \rhd & 4 & 1,2 & \rhd & 4 \\
3,4 & \rhd & 5 & 3 & \blacktriangleright & 5 \\
\overline{0},4 & \rhd & \overline{6} & \overline{0},4 & \rhd & \overline{6} \\
\overline{0},5 & \rhd & \overline{7} & \overline{0},5 & \rhd & \overline{7} \\
\overline{6},\overline{7} & \rhd & \bot & \overline{6} & \blacktriangleright & \bot
\end{array}$$

Each derivation $\Gamma \rhd b$ is essentially a sequent with $\Gamma$ as the antecedent and $b$ as the succedent. For proofs by contradiction, the atoms in the antecedent are either the negated conjecture ($\overline{0}$), facts that were proved elsewhere (1, 2, and 3), or atoms that were proved in preceding sequents (4, 5, $\overline{6}$, and $\overline{7}$). The succedent of the last sequent is always $\bot$.

Direct proofs can be presented in the same way, but the negated conjecture $\overline{0}$ may not appear in any of the sequents' antecedents, and the last sequent must have the conjecture 0 as its succedent. In some of the direct proofs, we find it useful to introduce case analyses. For example:

$$\begin{array}{rcl}
1,2 & \rhd & 4 \\
3 & \blacktriangleright & 5 \\
 & \rhd & 6 \lor 7
\end{array}$$
$$\left[\begin{array}{rcl|rcl}
 & & [6] & & & [7] \\
4 & \blacktriangleright & 0 & 5 & \blacktriangleright & 0
\end{array}\right]$$

In general, case analysis blocks have the form

$$\left[\begin{array}{rcl|c|rcl}
 & & [a_1] & \cdots & & & [a_n] \\
\Gamma_{11} & \rhd & b_{11} & \cdots & \Gamma_{n1} & \rhd & b_{n1} \\
 & \vdots & & & & \vdots & \\
\Gamma_{1k_1} & \rhd & b_{1k_1} & \cdots & \Gamma_{nk_n} & \rhd & b_{nk_n}
\end{array}\right]$$

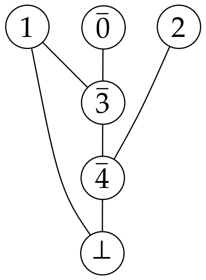
with the requirement that a sequent with the succedent $a_1 \lor \cdots \lor a_n$ has been proved immediately above the case analysis. Each of the branches must be a valid proof. The assumptions $[a_j]$ may be used to discharge hypotheses in the same branch, as if they had been sequents $\rhd\ a_j$. Taken as a unit, the case analysis block is indistinguishable from the sequent

$$a_1 \lor \cdots \lor a_n,\ \big(\textstyle\bigcup_{i,j} \Gamma_{ij} - \bigcup_j a_j - \bigcup_{i,j} b_{ij}\big) \ \rhd\ b_{1k_1} \lor \cdots \lor b_{nk_n}$$

### 6.8.2 Examples of Proof Redirection

Before we present the proof redirection algorithm, we consider four examples of proofs by contradiction and redirect them to produce a direct proof. The first example has a simple linear structure, the second and third examples involve a "lasso," and the last example has no apparent structure.

**A Linear Proof.** We start with a simple proof by contradiction expressed as a proof graph and in our shorthand notation:



$$\begin{aligned} \overline{0}, 1 &\rhd \overline{3} \\ 2, \overline{3} &\rhd \overline{4} \\ 1, \overline{4} &\rhd \bot \end{aligned}$$

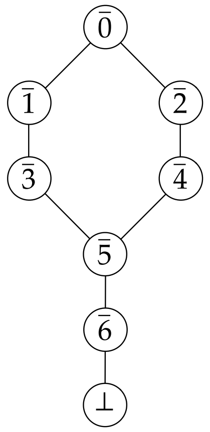
We redirect the sequents using (sequent-level) contraposition to eliminate all negations. This gives

$$\begin{aligned} 1, 3 &\rhd 0 \\ 2, 4 &\rhd 3 \\ 1 &\rhd 4 \end{aligned}$$

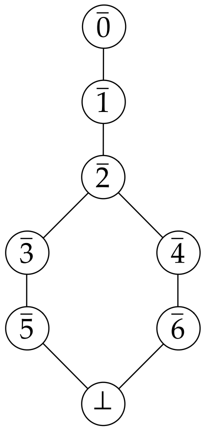
We then obtain the direct proof by reversing the order of the sequents, and introduce ▶ wherever possible:

```
proof –
  have 4 by (metis 1)          1 ▷ 4
  hence 3 by (metis 2)         2 ▶ 3
  thus 0 by (metis 1)          1 ▶ 0
qed
```

**Lasso-Shaped Proofs.** The next two examples look superficially like lassos:



$$\begin{aligned} \overline{0} &\rhd \overline{1} \\ \overline{0} &\rhd \overline{2} \\ \overline{1} &\rhd \overline{3} \\ \overline{2} &\rhd \overline{4} \\ \overline{3}, \overline{4} &\rhd \overline{5} \\ \overline{5} &\rhd \overline{6} \\ \overline{6} &\rhd \bot \end{aligned}$$



$$\begin{aligned} \overline{0} &\rhd \overline{1} \\ \overline{1} &\rhd \overline{2} \\ \overline{2} &\rhd \overline{3} \\ \overline{2} &\rhd \overline{4} \\ \overline{3} &\rhd \overline{5} \\ \overline{4} &\rhd \overline{6} \\ \overline{5}, \overline{6} &\rhd \bot \end{aligned}$$

We first focus on the example on the left-hand side. Starting from $\bot$, it is easy to redirect the proof segment up to the lasso cycle:

$$\begin{aligned} &\rhd 6 \\ 6 &\rhd 5 \\ 5 &\rhd 3 \vee 4 \end{aligned}$$

When applying the contrapositive to eliminate the negations in $\overline{3}, \overline{4} \rhd \overline{5}$, we obtain a disjunction in the succedent: $5 \rhd 3 \lor 4$. To continue from there, we introduce a case analysis. In each branch, we can finish the proof:

$$\left[\begin{array}{rcl|rcl} & & [3] & & & [4] \\ 3 & \rhd & 1 & 4 & \rhd & 2 \\ 1 & \rhd & 0 & 2 & \rhd & 0 \end{array}\right]$$

In the second lasso example, the cycle occurs near the end of the contradiction proof. A disjunction already arises when we redirect the last derivation. Naively finishing each branch independently leads to a fair amount of duplication:

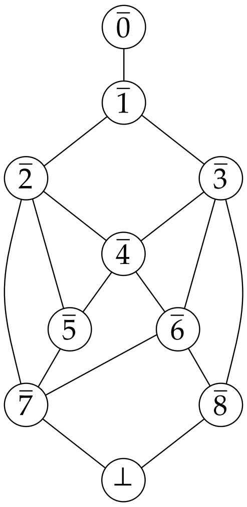
$$\begin{array}{c} \rhd\ 5 \lor 6 \\ \left[\begin{array}{rcl|rcl} & & [5] & & & [6] \\ 5 & \rhd & 3 & 6 & \rhd & 4 \\ 3 & \rhd & 2 & 4 & \rhd & 2 \\ 2 & \rhd & 1 & 2 & \rhd & 1 \\ 1 & \rhd & 0 & 1 & \rhd & 0 \end{array}\right] \end{array}$$

The key observation is that the two branches can share the last two steps. This yields the following proof (without and with $\blacktriangleright$):

$$\begin{array}{c} \rhd\ 5 \lor 6 \\ \left[\begin{array}{rcl|rcl} & & [5] & & & [6] \\ 5 & \rhd & 3 & 6 & \rhd & 4 \\ 3 & \rhd & 2 & 4 & \rhd & 2 \end{array}\right] \\ 2 \rhd 1 \\ 1 \rhd 0 \end{array} \qquad\qquad \begin{array}{c} \rhd\ 5 \lor 6 \\ \left[\begin{array}{r|r} [5] & [6] \\ \blacktriangleright\ 3 & \blacktriangleright\ 4 \\ \blacktriangleright\ 2 & \blacktriangleright\ 2 \end{array}\right] \\ \blacktriangleright\ 1 \\ \blacktriangleright\ 0 \end{array}$$

Here we were fortunate that the branches were joinable on 2. To avoid duplication, we must sometimes join on a disjunction $b_1 \lor \cdots \lor b_n$, as in the next example.

**A Spaghetti Proof.** The final example is diabolical (and slightly unrealistic):



$$\begin{array}{rcl} \overline{0} & \rhd & \overline{1} \\ \overline{1} & \rhd & \overline{2} \\ \overline{1} & \rhd & \overline{3} \\ \overline{2}, \overline{3} & \rhd & \overline{4} \\ \overline{2}, \overline{4} & \rhd & \overline{5} \\ \overline{3}, \overline{4} & \rhd & \overline{6} \\ \overline{2}, \overline{5}, \overline{6} & \rhd & \overline{7} \\ \overline{3}, \overline{6} & \rhd & \overline{8} \\ \overline{7}, \overline{8} & \rhd & \bot \end{array}$$

We start with the contrapositive of the last sequent:

$$\rhd\ 7 \vee 8$$

We perform a case analysis on $\rhd\ 7 \vee 8$. Since we want to avoid duplication in the two branches, we first determine which nodes are reachable in the refutation graph by navigating upward from either $\overline{7}$ or $\overline{8}$ but not from both. The only such nodes here are $\overline{5}$, $\overline{7}$, and $\overline{8}$. In each branch, we can perform derivations of the form $\Gamma \rhd b$ where $\Gamma \cap \{5,7,8\} \neq \emptyset$ without fearing duplication. Following this rule, we can only perform one inference in the right branch before we must stop:

$$\begin{array}{c} {[8]} \\ 8\ \rhd\ 3 \vee 6 \end{array}$$

Any further inferences would need to be repeated in the left branch, so it is indeed a good idea to stop. The left branch starts as follows:

$$\begin{array}{c} {[7]} \\ 7\ \rhd\ 2 \vee 5 \vee 6 \end{array}$$

We would now like to perform the inference $5\ \rhd\ 2 \vee 4$. This would certainly not lead to any duplication, because $\overline{5}$ is not reachable from $\overline{8}$ by navigating upward in the refutation graph. However, we cannot discharge the hypothesis 5, having established only the disjunction $2 \vee 5 \vee 6$. We need a case analysis on the disjunction to proceed:

$$\left[\begin{array}{c|c|c} & {[5]} & \\ {[2]} & 5\ \rhd\ 2 \vee 4 & {[6]} \end{array}\right]$$

The 2 and 6 subbranches are left alone, because there is no node that is reachable only by $\overline{2}$ or $\overline{6}$ but not by the other two nodes from $\{\overline{2},\overline{5},\overline{6}\}$ by navigating upward in the refutation graph. Since only one branch is nontrivial, we find it more aesthetically pleasing to abbreviate the entire case analysis to

$$2 \vee 5 \vee 6\ \rhd\ 2 \vee 4 \vee 6$$

Putting this all together, the outer case analysis becomes

$$\left[\begin{array}{c|c} {[7]} & \\ \blacktriangleright\ 2 \vee 5 \vee 6 & {[8]} \\ \blacktriangleright\ 2 \vee 4 \vee 6 & \blacktriangleright\ 3 \vee 6 \end{array}\right]$$

The left branch proves $2 \vee 4 \vee 6$, the right branch proves $3 \vee 6$; hence, both branches together prove $2 \vee 3 \vee 4 \vee 6$. Next, we perform the inference $6 \rhd 3 \vee 4$. This requires a case analysis on $2 \vee 3 \vee 4 \vee 6$:

$$\left[\begin{array}{c|c|c|c} & & & {[6]} \\ {[2]} & {[3]} & {[4]} & 6\ \rhd\ 3 \vee 4 \end{array}\right]$$

This proves $2 \vee 3 \vee 4$. Since only one branch is nontrivial, we can abbreviate the case analysis to

$$2 \vee 3 \vee 4 \vee 6\ \rhd\ 2 \vee 3 \vee 4$$

It may help to think of such derivation steps as instances of rewriting modulo associativity, commutativity, and idempotence. Here, 6 is rewritten to $3 \lor 4$ in $2 \lor 3 \lor 4 \lor 6$, resulting in $2 \lor 3 \lor 4$. Similarly, the sequent $4 \rhd 2 \lor 3$ gives rise to the case analysis

$$\left[\begin{array}{c|c|c} & & {[4]} \\ {[2]} & {[3]} & 4 \rhd 2 \lor 3 \end{array}\right]$$

which we abbreviate to

$$2 \lor 3 \lor 4 \rhd 2 \lor 3$$

We are left with $2 \lor 3$. The rest is analogous to the second lasso-shaped proof:

$$\begin{array}{c} \left[\begin{array}{c|c} {[2]} & {[3]} \\ 2 \rhd 1 & 3 \rhd 1 \end{array}\right] \\ 1 \rhd 0 \end{array}$$

Putting all of this together, we obtain the following proof, expressed in Isar and in shorthand using ►:

```
proof –
  have 7 ∨ 8 by metis
  moreover
  { assume 7
    hence 2 ∨ 5 ∨ 6 by metis
    hence 2 ∨ 4 ∨ 6 by metis }
  moreover
  { assume 8
    hence 3 ∨ 6 by metis }
  ultimately have 2 ∨ 3 ∨ 4 ∨ 6 by metis
  hence 2 ∨ 3 ∨ 4 by metis
  hence 2 ∨ 3 by metis
  moreover
  { assume 2
    hence 1 by metis }
  moreover
  { assume 3
    hence 1 by metis }
  ultimately have 1 by metis
  thus 0 by metis
qed
```

$$\begin{array}{c} \rhd 7 \lor 8 \\ \left[\begin{array}{c|c} {[7]} & \\ \blacktriangleright 2 \lor 5 \lor 6 & {[8]} \\ \blacktriangleright 2 \lor 4 \lor 6 & \blacktriangleright 3 \lor 6 \end{array}\right] \\ \blacktriangleright 2 \lor 3 \lor 4 \\ \blacktriangleright 2 \lor 3 \\ \left[\begin{array}{c|c} {[2]} & {[3]} \\ \blacktriangleright 1 & \blacktriangleright 1 \end{array}\right] \\ \blacktriangleright 0 \end{array}$$

The result is quite respectable, considering what we started with.

### 6.8.3 The Redirection Algorithm

The process we applied in the examples above can be generalized into an algorithm. The algorithm takes an arbitrary proof by contradiction expressed as a set of sequents as input, and produces a proof in our Isar-like shorthand notation, with sequents and case analysis blocks. The proof is constructed one inference at a time starting from $\top$ until the conjecture is proved. The algorithm relies on a few auxiliary notions that we present first.

A fundamental operation is sequent-level contraposition. As in the examples, we conveniently assume that the negated conjecture and all the nodes that are tainted by it are negated atoms while the other atoms are positive. A proof by contradiction then consists of three kinds of sequent:

1. $a_1, \ldots, a_k, \overline{a_{k+1}}, \ldots, \overline{a_m} \ \rhd\ \bot$
2. $a_1, \ldots, a_k, \overline{a_{k+1}}, \ldots, \overline{a_m} \ \rhd\ \overline{b}$
3. $a_1, \ldots, a_m \ \rhd\ b$

The *contrapositive* of each kind of sequent is given below:

1. $a_1, \ldots, a_k \ \rhd\ a_{k+1} \vee \cdots \vee a_m$
2. $a_1, \ldots, a_k, b \ \rhd\ a_{k+1} \vee \cdots \vee a_m$
3. $a_1, \ldots, a_m \ \rhd\ b$

In all three cases, the contrapositive has the general form

$$a_1,\ \ldots,\ a_m \ \rhd\ b_1 \vee \cdots \vee b_n$$

We call the contrapositives of the sequents in the proof by contradiction the *redirected sequents*.

Based on the set of redirected sequents, we define the *atomic inference graph* (AIG) with, for each sequent of the above general form, an edge from each $a_i$ to each $b_j$, and no additional edges. The AIG is acyclic. Navigating forward in this graph corresponds roughly to navigating backward, or upward, in the refutation graph.

Given a set of atoms $\{b_1, \ldots, b_n\}$, the *zone* of an atom $b_j$ is the set of descendants of $b_j$ in the AIG that are not descendants of any of the other atoms in the set. Being a descendant of itself, $b_j$ may (but need not) belong to its own zone.

The algorithm keeps track of the *last-proved clause* (initially $\top$), the set of *already proved atoms* (initially the set of facts taken as axioms), and the set of *remaining sequents* to use (initially all the redirected sequents provided as input). It performs the following steps:

1. If there are no remaining sequents, stop.
2. If the last-proved clause is $\top$ or a single atom $b$:
   - 2.1. Pick a sequent $\Gamma \rhd c$ among the remaining sequents that can be proved using only already proved atoms, preferring sequents with a single atom in their succedent.
   - 2.2. Append $\Gamma \rhd c$ to the proof.
   - 2.3. Make $c$ the last-proved clause, add $c$ to the already proved atoms if it is an atom, and remove $\Gamma \rhd c$ from the remaining sequents.
   - 2.4. Go to step 1.

3. Otherwise, the last-proved succedent is of the form $b_1 \vee \cdots \vee b_n$. An $n$-way case analysis is called for:[1]
   - 3.1. Compute the zone of each atom $b_j$ with respect to $\{b_1, \ldots, b_n\}$.
   - 3.2. For each $b_j$, compute the set $\mathcal{S}_j$ of sequents $(\Gamma, c)$ such that $\Gamma$ consists only of already proved atoms or atoms within $b_j$'s zone.
   - 3.3. Recursively invoke the algorithm $n$ times, once for each $b_j$, each time with $b_j$ as the last-proved clause, with $b_j$ added to the already proved atoms and $\mathcal{S}_j$ as the set of remaining sequents. This step yields $n$ (possibly empty) subproofs $\pi_1, \ldots, \pi_n$.
   - 3.4. Append the following case analysis block to the proof:

$$\left[\begin{array}{c|c|c} [b_1] & \cdots & [b_n] \\ \pi_1 & \cdots & \pi_n \end{array}\right]$$

   - 3.5. Make the succedent $c_1 \vee \cdots \vee c_k$ of the case analysis block the last-proved clause, add $c_1$ to the already proved atoms if $k = 1$, and remove all sequents belonging to any of the sets $\mathcal{S}_j$ from the remaining sequents.
   - 3.6. Go to step 1.

To make this description more concrete, the algorithm is presented in Standard ML pseudocode below. The pseudocode is fairly faithful to the description above. Atoms are represented by integers and literals by sets (lists) of integers. Go-to statements are implemented by tail-recursion, and the state is threaded through recursive calls as three arguments (*last*, *earlier*, and *seqs*).

One notable difference, justified by a desire to avoid code duplication, is that the set of already proved atoms, called *earlier*, excludes the last-proved clause *last*. Hence we take *last* ∪ *earlier* to obtain the already proved atoms, where *last* is either the empty list (representing $\top$) or a singleton list (representing a single atom).

Shorthand proofs are represented as lists of *inference*s:

```
datatype inference =
  Have of int list × int list |
  Cases of (int × inference list) list
```

The main function implementing the algorithm follows:

```
fun redirect last earlier seqs =
  if null seqs then
    []
  else if length last ≤ 1 then
    let
      val provable = filter (fn (Γ, _) ⇒ Γ ⊆ last ∪ earlier) seqs
      val horn_provable = filter (fn (_, [_]) ⇒ true | _ ⇒ false) provable
      val (Γ, c) = hd (horn_provable @ provable)
```

[1] A generalization would be to perform a $m$-way case analysis, with $m < n$, by keeping some disjunctions. For example, we could perform a 3-way case analysis with $b_1 \vee b_2$, $b_3$, and $b_4$ as the assumptions instead of breaking all the disjunctions in a 4-way analysis. This could potentially lead to cleaner proofs if the disjuncts are carefully chosen.

```
    in Have $(\Gamma, c)$ :: redirect $c$ $(last \cup earlier)$ (filter $((\neq)\ (\Gamma, c))$ $seqs$) end
  else
    let
      val $zs$ = zones_of (length $last$) (map (descendants $seqs$) $last$)
      val $\mathcal{S}$ = map (fn $z \Rightarrow$ filter (fn $(\Gamma, \_) \Rightarrow \Gamma \subseteq earlier \cup z$) $seqs$) $zs$
      val $cases$ = map (fn $(b, ss) \Rightarrow (b,$ redirect $[b]$ $earlier\ ss$)) (zip $last$ $\mathcal{S}$)
    in Cases $cases$ :: redirect (succedent_of_cases $cases$) $earlier$ $(seqs - \bigcup \mathcal{S})$ end
```

The code uses familiar ML functions, such as map, filter, and zip. It also relies on a descendants function that returns the descendants of the specified node in the AIG associated with *seqs;* its definition is omitted. Finally, the code depends on the following straightforward functions:

```
fun zones_of 0 _ = []
  | zones_of $n$ $(B :: Bs)$ = $(B - \bigcup Bs)$ :: zones_of $(n - 1)$ $(Bs$ @ $[B])$

fun succedent_of_inf (Have $(\_, c)$) = $c$
  | succedent_of_inf (Cases $cases$) = succedent_of_cases $cases$
and succedent_of_case $(a, [])$ = $[a]$
  | succedent_of_case $(\_, infs)$ = succedent_of_inf (last $infs$)
and succedent_of_cases $cases$ = $\bigcup$ (map succedent_of_case $cases$)
```

As a postprocessing step, we abbreviate case analyses in which only one branch is nontrivial, transforming

$$\left[\begin{array}{c|c|c|r|c|c|c} & & & [c_i] & & & \\ & & & a_{11}, \ldots, a_{1k_1} \ \rhd\ d_1 & & & \\ & & & \vdots & & & \\ [c_1] & \cdots & [c_{i-1}] & a_{n1}, \ldots, a_{nk_n} \ \rhd\ d_n & [c_{i+1}] & \cdots & [c_m] \end{array}\right]$$

into

$$\begin{array}{r} c_1 \lor \cdots \lor c_m,\ a_{11}, \ldots, a_{1k_1} \ \rhd\ \widetilde{d_1} \\ \widetilde{a_{21}}, \ldots, \widetilde{a_{2k_2}} \ \rhd\ \widetilde{d_2} \\ \vdots \\ \widetilde{a_{n1}}, \ldots, \widetilde{a_{nk_n}} \ \rhd\ \widetilde{d_n} \end{array}$$

where the function $\tilde{}$ is the identity except for the $d_i$'s:

$$\widetilde{d_i} = c_1 \lor \cdots \lor c_{i-1} \lor d_i \lor c_{i+1} \lor \cdots \lor c_m$$

Finally, we clean up the proof by introducing ▶ and translate it to Isar.

It is not hard to convince ourselves that the proof output by redirect is correct by inspecting the code. A Have $(\Gamma, c)$ sequent is appended only if all the atoms in $\Gamma$ have been proved (or assumed) already, and a case analysis on $b_1 \lor \cdots \lor b_n$ always follows a sequent with the succedent $b_1 \lor \cdots \lor b_n$. Whenever a sequent is output, it is removed from the set *seqs*. The function returns only if *seqs* is empty, at which point we can be assured that the conjecture has been proved.

Termination is not quite as obvious. The recursion is well-founded, because the pair $\langle$length *seqs*, length *last*$\rangle$ becomes strictly smaller with respect to the lexicographic extension of $<$ for each of the three syntactic recursive calls. For the first recursive call, the list filter $((\neq)\ (\Gamma,\ c))$ *seqs* is strictly shorter than *seqs* since $(\Gamma,\ c) \in$ *seqs*. The second call is performed for each branch of a case analysis; the *ss* argument is a (not necessarily strict) subset of the caller's *seqs*, and the list $[b]$ is strictly shorter than *last*, which has length 2 or more. For the third call, the key property is that at least one of the zones is nonempty, from which we obtain $seqs - \bigcup \mathcal{S} \subset seqs$. (If all the zones were empty, then $b_1$ would be a descendant of $b_2$ in the AIG and vice versa, pointing to a cycle in the refutation graph.)

As for run-time exceptions, the only worrisome construct is the hd call in redirect's second branch. We must convince ourselves that there exists at least one sequent $(\Gamma,\ c) \in seqs$ such that $\Gamma \subseteq last \cup earlier$. Intuitively, this is unsurprising, because *seqs* is initialized from a well-formed refutation graph: The nonexistence of such a sequent would indicate a gap or a cycle in the graph.

## 6.9 Related Work

Sledgehammer is just one of many proof tools that bridge automatic and interactive theorem provers. We review the main integrations of ATPs and SMT solvers as well as the type translation schemes proposed in the literature.

**ATP Integrations.** There have been several attempts to integrate ATPs in interactive provers, usually with proof reconstruction. The most notable integrations are probably Otter in ACL2 [119]; Bliksem and Zenon in Coq [23,46]; Gandalf in HOL98 [91]; DISCOUNT, SETHEO, and SPASS in ILF [64]; Zenon in Isabelle/TLA$^+$ [55]; Otter, PROTEIN, SETHEO, SPASS, and $_3T\!A\!P$ in KIV [2,161]; and Bliksem, EQP, LEO, Otter, PROTEIN, SPASS, TPS, and Waldmeister in ΩMEGA [122,171]. Regrettably, few if any of these appear to have withstood the test of time. Sledgehammer's success ostensibly inspired the new Miz𝔸ℝ web service for Mizar [164], based on Vampire and SInE. An integration of the equational prover Waldmeister with Agda is under development [78].

**SMT Solver Integrations.** There have also been several attempts to combine interactive theorem provers with SMT solvers, either as oracles or with proof reconstruction. ACL2 and PVS employ UCLID and Yices as oracles [115, 165]. HOL Light integrates CVC Lite and reconstructs its proofs [121]. Isabelle/HOL enjoys two oracle integrations [13,76] and two tactics with proof reconstruction [42,45,77]. HOL4 includes an SMT tactic [42,45,199], and an integration of veriT in Coq, based on SMT proof validation, is in progress [5].

**Type Translations.** The earliest descriptions of type guards and type tags we are aware of are due to Enderton [75, §4.3] and Stickel [174, p. 99]. Wick and McCune [205, §4] compare full type erasure, guards, and tags. Type arguments are reminiscent of System F; they are described by Meng and Paulson [125], who also present a translation of axiomatic type classes. The intermediate verification language and tool Boogie 2 [111] supports a restricted form of higher-rank polymorphism (with polymorphic maps), and its cousin Why3 [39] supports rank-1 (ML-style) polymorphism. Both define translations to a monomorphic logic and rely on proxies to handle interpreted types [40,63,111]. One of the Boogie translations [111, §3.1] ingeniously uses SMT triggers to prevent ill-typed variable instantiations in conjunction with type arguments; however, this approach is risky in the absence of a semantics for triggers.

An alternative to encoding polymorphic types or monomorphizing them away is to support them directly in the prover. This is ubiquitous in interactive theorem provers, but perhaps the only automatic prover with native support for polymorphism is Alt-Ergo [38].

Do I contradict myself?
Very well then I contradict myself.
— Walt Whitman (1855)

# Chapter 7

# Conclusion

Interactive theorem proving is notoriously laborious, even for highly specialized experts. This thesis presented work on two tools, Nitpick and Sledgehammer, that make interactive proving in Isabelle/HOL more productive and enjoyable.

## 7.1 Results

Isabelle/HOL offers a wide range of automatic tools for proving and disproving conjectures. Some are built into the prover, but increasingly these activities are delegated to external tools, such as resolution provers, SAT solvers, and SMT solvers. While there have been several attempts at integrating external provers and disprovers in other interactive theorem provers, Isabelle is probably the only interactive prover where external tools play such a prominent role, to the extent that they are now seen as indispensable by many if not most users.

Nitpick is a new higher-order model finder that supports both inductive and coinductive predicates and datatypes. It translates higher-order formulas to first-order relational logic (FORL) and invokes the SAT-based Kodkod model finder [187] to solve these. Compared with Quickcheck, which is restricted to executable formulas, Nitpick shines by its generality—the hallmark of SAT-based model finding.

Nitpick's translation to FORL exploits Kodkod's strengths. Datatypes are encoded following an Alloy idiom extended to mutually recursive and coinductive datatypes. FORL's relational operators provide a natural encoding of partial application and $\lambda$-abstraction, and the transitive closure plays a crucial role in the encoding of inductive datatypes. Our main contributions have been to isolate three ways to translate (co)inductive predicates to FORL, based on wellfoundedness, polarity, and linearity, and to devise optimizations—notably function specialization, boxing, and monotonicity inference—that dramatically increase scalability.

Unlike Nitpick, Sledgehammer was already a popular tool when we took over its development. To existing users, the addition of SMT solvers as backends means that they obtain even more proofs without effort. The SMT solvers compete advantageously with the resolution-based ATPs on all kinds of problem. This came as a surprise to Sledgehammer's first developer, Lawrence Paulson, who admits he "never imagined we would get any mileage out of such tools" [149]. Running the

SMT solvers in parallel with the ATPs is entirely appropriate, for how is the user supposed to know which class of prover will perform best?

To users of SMT solvers, the Sledgehammer–SMT integration eases the transition from automatic proving in first-order logic to interactive proving in higher-order logic. Isabelle/HOL is powerful enough for the vast majority of hardware and software verification efforts, and its LCF-style inference kernel provides a trustworthy foundation. Even the developers of SMT solvers profit from the integration: It helps them reach a larger audience, and proof reconstruction brings to light bugs in their tools, including soundness bugs, which might otherwise go undetected.[1]

Support for the TPTP typed first-order form has recently been added to Vampire, and it is expected that E and SPASS will follow suit. We extended Sledgehammer to generate typed problems for Vampire. To interact with ATPs with no support for types, we implemented a family of sound and efficient translations in Sledgehammer and the companion proof method *metis*, thereby addressing a recurring user complaint. Although Isabelle certifies external proofs, unsound proofs are annoying and may conceal sound proofs.

In terms of usefulness, Sledgehammer is arguably second only to the simplifier and tableau prover. But Nitpick also provides invaluable help and encourages a lightweight explorative style to formal specification development, as championed by Alloy. We frequently receive emails from users grateful to have been spared "several hours of hard work."

An explanation for Nitpick's and Sledgehammer's success is that they are included with Isabelle and require no additional installation steps. External tools necessary to their operation are either included in the official Isabelle packages or accessible as online services. Multi-core architectures and remote servers help to bear the burden of (dis)proof, so that users can continue working on a manual proof while the tools run in the background.

Another important design goal for both tools was one-click invocation. Users should not be expected to preprocess the goals or specify options. Even better than one-click invocation is zero-click invocation, whereby the tools spontaneously run on newly entered conjectures. A more flexible user interface, such as the experimental jEdit-based PIDE [204], could help further here, by asynchronously dispatching the tools to tackle any unfinished proofs in the current proof document, irrespective of the text cursor's location.

Guttmann, Struth, and Weber's recent work on a large Isabelle/HOL repository of relational and algebraic methods for modeling computing systems [85] constitutes an unanticipated validation of our work: They relied almost exclusively on Sledgehammer and the SMT integration to prove over 1000 propositions, including intricate refinement and termination theorems. To their surprise, they found that

[1] Indeed, we discovered a soundness bug in Yices and another in Z3 while evaluating them. On the ATP side of things, proof reconstruction with *metis* drew our attention to unsoundnesses in Satallax and in Vampire's newly introduced type support, and Nitpick helped uncover soundness issues in the higher-order provers LEO-II and TPS as well as severe SAT solver integration issues in Kodkod. Our experience vividly illustrates the dangers of uncritically accepting proofs from external sources. Remarkably, no critical bugs were found in Isabelle or any SAT solver.

Sledgehammer can often automate algebraic proofs at the textbook level. Echoing Lakatos [108], they remarked that "counterexample generators such as Nitpick complement the ATP systems and allow a proof and refutation game which is useful for developing and debugging formal specifications" [85, p. 15].

Interactive theorem proving remains very challenging, but thanks to a new generation of automatic proof and disproof tools and the wide availability of multi-core processors with spare cycles, it is much easier and more enjoyable now than it was only a few years ago.

## 7.2 Future Work

Nitpick and Sledgehammer suffer from a number of limitations, most of which could be addressed in future work.

### 7.2.1 Counterexample Generation with Nitpick

**Additional Backends.** Nitpick takes part in the first-order model finding division of the annual CADE ATP System Competition (CASC). The 2011 competition results [181] suggest that Nitpick could scale slightly better by employing Paradox [62] as a backend. The main difficulty with such an integration is that Paradox (like the other model finders that compete at CASC) is based on the untyped TPTP FOF syntax, which is a poor match for FORL. Moreover, we do not expect multiple backends to be as beneficial as for proving, because of the systematic nature of finite model finding: If Kodkod exhaustively checked that there is no model of a given cardinality, Paradox will not find any either.

**Exhaustive LCF-Style Proofs.** Nitpick was primarily designed as a countermodel finder, but if the problem involves only types with bounded finite cardinalities, the absence of countermodels up to the cardinality bounds implies that the conjecture holds. We relied on this observation when we exhaustively checked litmus tests against the C++ memory model. Kodkod can provide unsatisfiability proofs, and we could in principle replay them in Isabelle's inference kernel to obtain a trustworthy decision procedure for a generalization of effectively propositional problems.

**Enhanced Interrogation Techniques.** Large axiomatic specifications, such as that of the C++ memory model, are often overconstrained, and it is extremely tedious for the user to reduce the specification manually to identify the culprit. Kodkod can provide a minimal unsatisfiable core; it should be possible to translate it into a set of inconsistent HOL formulas that explains why no models exist. Other useful diagnosis tools would be a facility for inspecting the value of any term occurring in the problem and a graphical output similar to that of the Alloy Analyzer. More generally, it would be desirable for Nitpick to give more insight into what it does—for example, whether and where approximation takes place. The C++ memory model case study was within our reach only because we understood how the tool worked behind the scenes and could make sense of the generated Kodkod problems.

**Hybrid Approaches.** Random testing, narrowing, data-flow analysis, and code equations have proved useful in the context of Quickcheck, but we currently know of no good way of integrating these approaches with SAT solving. Perhaps a tight integration with an SMT solver, as was done recently for a fragment of Scala [185], would help combine ground and symbolic reasoning in a meaningful way.

### 7.2.2 Proof Discovery with Sledgehammer

**Structured Proof Construction.** On the Sledgehammer side of things, more work is needed on the construction of readable, concise, and reliable Isar proofs from the resolution proofs produced by the ATPs. There is plenty of inspiring prior art, notably TRAMP [122] and Otterfier [208]. Ideally, Isar proof construction should be extended to SMT solvers.

**Translation of $\lambda$-Abstractions.** Sledgehammer's combinator-based translation of $\lambda$-abstractions performs poorly in practice; $\lambda$-lifting is slightly superior for some provers, but it is hopelessly incomplete. Obermeyer recently demonstrated that "combinatory algebra / combinatory logic is not as prohibitively unreadable and combinatorially explosive as is commonly thought" [142, p. 17] if one supplements the combinators' definitions with redundant equations relating them to one another. A completely different translation scheme would be to embed the $\lambda$-calculus with de Bruijn indices in first-order logic, with an explicit substitution operator [69].

**Type Encodings.** Efficient type encodings are crucial to Sledgehammer's performance. A promising direction for future research would be to look into strengthening the monotonicity analysis. Our work in the context of Nitpick showed how to detect definitions and handle them specially, but we have yet to try this out in Sledgehammer. Type arguments clutter our polymorphic translations; they can often be omitted soundly, but we lack an inference to find out precisely when.

**Arithmetic Support.** Sledgehammer's ATP translation currently makes no provision for arithmetic. SPASS+T [158] and recent versions of Vampire support arithmetic natively, and it would be desirable to map Isabelle's arithmetic constants to the corresponding TPTP constructs. Proof reconstruction is a challenge here, because Metis does not know about arithmetic.

**Relevance Filtering.** The relevance filter is simplistic. It might be preferable to let the automatic theorem provers perform relevance filtering or use a sophisticated system based on machine learning, such as MaLARea [191, 192], where successful proofs guide later proofs.

**Metainformation Export.** Much metainformation is lost in the translation from Isabelle/HOL to first-order logic: whether a symbol is a free constructor, whether an equation is an oriented simplification rule, and so on. Automatic provers could exploit this information in various ways, but we must agree on a syntax and semantics. Work has commenced in Saarbrücken to handle Isabelle metainformation meaningfully [36].

# Bibliography

[1] *ISO/IEC 14882:2011: Information Technology—Programming Languages—C++*. ISO IEC JTC1/SC22, 2011.

[2] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reif, G. Schellhorn, and P. H. Schmitt. Integrating automated and interactive theorem proving. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, pages 97–116. Kluwer, 1998.

[3] A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, N. Santoro, and S. Whitesides, editors, *WADS 1993*, volume 709 of *LNCS*, pages 61–70. Springer, 1993.

[4] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic*. Springer, second edition, 2002.

[5] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *CPP 2011*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

[6] S. Autexier, C. Benzmüller, A. Fiedler, H. Horacek, and Q. B. Vo. Assertion-level proof representation with under-specification. *Electr. Notes Theor. Comput. Sci.*, 93:5–23, 2004.

[7] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 19–99. Elsevier, 2001.

[8] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Inf. Comput.*, 121(2):172–192, 1995.

[9] J. Backes and C. E. Brown. Analytic tableaux for higher-order logic with choice. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNAI*, pages 76–90. Springer, 2010.

[10] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[11] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard—Version 2.0. In A. Gupta and D. Kroening, editors, *SMT 2010*, 2010.

[12] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[13] D. Barsotti, L. P. Nieto, and A. Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. *Formal Asp. Comput.*, 19(3):321–341, 2007.

[14] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In J. Field and M. Hicks, editors, *POPL 2012*, pages 509–520. ACM, 2012.

[15] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency: The post-Rapperswil model. Technical Report N3132, ISO IEC JTC1/SC22/WG21, 2010.

[16] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *POPL 2011*, pages 55–66. ACM, 2011.

[17] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. *ACM SIGPLAN Notices*, 32(8):25–37, 1997.

[18] C. Benzmüller. Private communication, Aug. 2011.

[19] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II—A cooperative automatic theorem prover for higher-order logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *LNAI*, pages 162–170. Springer, 2008.

[20] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0—The core of the TPTP language for higher-order logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *LNAI*, pages 491–506. Springer, 2008.

[21] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *SEFM 2004*, pages 230–239. IEEE C.S., 2004.

[22] S. Berghofer and M. Wenzel. Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs '99*, volume 1690 of *LNCS*, pages 19–36, 1999.

[23] M. Bezem, D. Hendriks, and H. de Nivelle. Automatic proof construction in type theory using resolution. *J. Autom. Reasoning*, 29(3-4):253–275, 2002.

[24] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *TACAS '99*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[25] J. C. Blanchette. Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. To appear in *Softw. Qual. J.*

[26] J. C. Blanchette. Proof pearl: Mechanizing the textbook proof of Huffman's algorithm in Isabelle/HOL. *J. Autom. Reasoning*, 43(1):1–18, 2009.

[27] J. C. Blanchette. Hammering away: A user's guide to Sledgehammer for Isabelle/HOL.
`http://www21.in.tum.de/dist/Isabelle/doc/sledgehammer.pdf`, 2011.

[28] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNAI*, pages 207–221. Springer, 2011.

[29] J. C. Blanchette, S. Böhme, and N. Smallbone. Monotonicity or how to encode polymorphic types safely and efficiently. Submitted.

[30] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[31] J. C. Blanchette and K. Claessen. Generating counterexamples for structural inductions by exploiting nonstandard models. In C. G. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNAI*, pages 117–141. Springer, 2010.

[32] J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNAI*, pages 91–106. Springer, 2010.

[33] J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. *J. Autom. Reasoning*, 47(4):369–398, 2011.

[34] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

[35] J. C. Blanchette and A. Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. Submitted.

[36] J. C. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. More SPASS with Isabelle—Superposition with hard sorts and configurable simplification. Submitted.

[37] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *PPDP 2011*, pages 113–124. ACM Press, 2011.

[38] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In C. Barrett, L. de Moura, D. Babic, and A. Goel, editors, *SMT/BPR '08*, ICPS, pages 1–5. ACM, 2008.

[39] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In K. R. M. Leino and M. Moskal, editors, *Boogie 2011*, pages 53–64, 2011.

[40] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNAI*, pages 87–102. Springer, 2011.

[41] S. Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. Ph.D. thesis, Dept. of Informatics, T.U. München, 2012.

[42] S. Böhme, A. C. J. Fox, T. Sewell, and T. Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In J.-P. Jouannaud and Z. Shao, editors, *CPP 2011*, volume 7086 of *LNCS*, pages 183–198. Springer, 2011.

[43] S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie—An interactive prover-backend for the Verifying C Compiler. *J. Autom. Reasoning*, 44(1-2):111–144, 2010.

[44] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNAI*, pages 107–121. Springer, 2010.

[45] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[46] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *LPAR 2007*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.

[47] A. R. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20:379–405, 2008.

[48] D. Brand. Proving theorems with the modifiction method. *SIAM J. Comput.*, 4(4):412–430, 1975.

[49] L. Bulwahn. Smart testing of functional programs in Isabelle. In N. Bjørner and A. Voronkov, editors, *LPAR-18*, volume 7180 of *LNCS*, pages 153–167. Springer, 2012.

[50] L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 38–53. Springer, 2007.

[51] R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*, volume 31 of *Applied Logic*. Springer, 2004.

[52] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In R. De Nicola, editor, *ESOP 2007*, volume 4421 of *LNCS*, pages 331–346. Springer, 2007.

[53] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Autom. Reasoning*, 41(1):33–59, 2008.

[54] H. R. Chamarthi, P. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. `http://arxiv.org/pdf/1105.4394`, 2011.

[55] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA$^+$ proof system. In P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, editors, *LPAR 2008 Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[56] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[57] K. Claessen. Equinox, a new theorem prover for full first-order logic with equality. Presentation at the Dagstuhl Seminar on Deduction and Applications, 2005.

[58] K. Claessen. Private communication, Apr. 2009.

[59] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00*, pages 268–279. ACM, 2000.

[60] K. Claessen and A. Lillieström. Automated inference of finite unsatisfiability. *J. Autom. Reasoning*, 47(2):111–132, 2011.

[61] K. Claessen, A. Lillieström, and N. Smallbone. Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNAI*, pages 207–221. Springer, 2011.

[62] K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *MODEL*, 2003.

[63] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNAI*, pages 263–278. Springer, 2007.

[64] B. Dahn, J. Gehne, T. Honigmann, and A. Wolf. Integration of automated and interactive theorem proving in ILF. In W. McCune, editor, *CADE-14*, volume 1249 of *LNCS*, pages 57–60. Springer, 1997.

[65] A. L. de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. Ph.D. thesis, C.S. Dept., University of Glasgow, 1995.

[66] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[67] H. de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. *Inf. Comput.*, 199(1-2):24–54, 2005.

[68] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[69] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions (extended abstract). In *LICS '95*, pages 366–374. IEEE, 1995.

[70] A. Dunets, G. Schellhorn, and W. Reif. Automated flaw detection in algebraic specifications. *J. Autom. Reasoning*, 45(4):359–395, 2010.

[71] B. Dutertre and L. de Moura. The Yices SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[72] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. A. Basin and B. Wolff, editors, *TPHOLs 2003*, volume 2758 of *LNCS*, pages 188–203. Springer, 2003.

[73] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[74] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[75] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[76] L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In J. Rushby and N. Shankar, editors, *AFM08*, pages 3–13, 2008.

[77] P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In H. Hermanns and J. Palsberg, editors, *TACAS 2006*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.

[78] S. Foster and G. Struth. Integrating an automated theorem prover into Agda. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NFM 2011*, volume 6617 of *LNCS*, pages 116–130, 2011.

[79] M. F. Frias, C. G. L. Pombo, and M. M. Moscato. Alloy Analyzer + PVS in the analysis and verification of Alloy specifications. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 587–601. Springer, 2007.

[80] S. Friedrich. More on lazy lists. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Lazy-Lists-II.shtml`, 2004.

[81] M. Gebser, O. Sabuncu, and T. Schaub. An incremental answer set programming based system for finite model computation. *AI Comm.*, 24(2):195–212, 2011.

[82] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[83] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[84] E. L. Gunter. Why we can't have SML style datatype declarations in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *TPHOLs '92*, volume A-20 of *IFIP Transactions*, pages 561–568. North-Holland, 1993.

[85] W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In S. Qin and Z. Qiu, editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 617–632. Springer, 2011.

[86] J. Harrison. Inductive definitions: Automation and application. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *TPHOLs 1995*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.

[87] T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. WALDMEISTER—High-performance equational deduction. *J. Autom. Reasoning*, 18(2):265–270, 1997.

[88] K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNAI*, pages 299–314. Springer, 2011.

[89] X. Huang. Translating machine-generated resolution proofs into ND-proofs at the assertion level. In N. Y. Foo and R. Goebel, editors, *PRICAI '96*, volume 1114 of *LNCS*, pages 399–410. Springer, 1996.

[90] R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *LFP 1982*, pages 1–10. ACM Press, 1982.

[91] J. Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs '99*, volume 1690 of *LNCS*, pages 311–321, 1999.

[92] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

[93] D. Jackson. Nitpick: A checkable specification language. In *FMSP '96*, pages 60–69, 1996.

[94] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[95] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *ESEC/FSE 2001*, pages 62–73, 2001.

[96] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bull. EATCS*, 62:222–259, 1997.

[97] S. P. Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[98] J. Jürjens and T. Weber. Finite models in FOL-based crypto-protocol verification. In P. Degano and L. Viganò, editors, *ARSPA-WITS 2009*, volume 5511 of *LNCS*, pages 155–172. Springer, 2009.

[99] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *ASE 2009*, pages 495–499. IEEE, 2009.

[100] S. C. Kleene. On notation for ordinal numbers. *J. Symb. Log.*, 3(4):150–155, 1938.

[101] S. C. Kleene. Representation of events in nerve nets and finite automata. In J. McCarthy and C. Shannon, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[102] G. Klein, T. Nipkow, and L. Paulson, editors. *The Archive of Formal Proofs.* `http://afp.sf.net/`.

[103] S. Klingenbeck. *Counter Examples in Semantic Tableaux.* Infix, 1997.

[104] K. Korovin. Instantiation-based automated reasoning: From theory to practice. In R. A. Schmidt, editor, *CADE-22*, volume 5663 of *LNAI*, pages 163–166. Springer, 2009.

[105] D. C. Kozen. *Automata and Computability.* Undergraduate Texts in Computer Science. Springer, 1997.

[106] A. Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2009.

[107] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In M. Wermelinger and H. Gall, editors, *ESEC/FSE 2005.* ACM, 2005.

[108] I. Lakatos. *Proofs and Refutations.* Cambridge University Press, 1976. Posthumous edition by J. Worrall and E. Zahar.

[109] C. Lameter. Effective synchronization on Linux/NUMA systems. Presentation at the Gelato Conference, 2005.

[110] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[111] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

[112] F. Lindblad. Property directed generation of first-order test data. In M. Morazán, editor, *TFP 2007*, pages 105–123. Intellect, 2008.

[113] A. Lochbihler. Formalising FinFuns—Generating code for functions as data from Isabelle/HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 310–326. Springer, 2009.

[114] A. Lochbihler. Coinduction. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Coinductive.shtml`, 2010.

[115] P. Manolios and S. K. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *ICCAD '05*, pages 855–862. IEEE, 2005.

[116] J. Manson and W. Pugh. The Java memory model simulator. In *FTfJP 2002*, 2002.

[117] W. McCune. A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.

[118] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2010.

[119] W. McCune and O. Shumsky. System description: IVY. In D. McAllester, editor, *CADE-17*, volume 1831 of *LNAI*, pages 401–405. Springer, 2000.

[120] A. McIver and T. Weber. Towards automated proof support for probabilistic distributed systems. In G. Sutcliffe and A. Voronkov, editors, *LPAR 2005*, volume 3835 of *LNAI*, pages 534–548. Springer, 2005.

[121] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. *Electr. Notes Theor. Comput. Sci.*, 144(2):43–51, 2006.

[122] A. Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level (system description). In D. McAllester, editor, *CADE-17*, volume 1831 of *LNAI*, pages 460–464. Springer, 2000.

[123] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer, 1989.

[124] A. C. Melo and S. C. Chagas. Visual-MCM: Visualising execution histories on multiple memory consistency models. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *ACPC 1999*, volume 1557 of *LNCS*, pages 500–509. Springer, 1999.

[125] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[126] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[127] J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.

[128] D. Miller and A. Felty. An integration of resolution and natural deduction theorem proving. In *AAAI-86*, volume I: Science, pages 198–202. Morgan Kaufmann, 1986.

[129] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, 1997.

[130] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[131] L. Momtahan. Towards a small model theorem for data independent systems in Alloy. *Electr. Notes Theor. Comput. Sci.*, 128(6):37–52, 2005.

[132] A. Nadel. *Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and Innovations*. M.Sc. thesis, Inst. of C.S., Hebrew University of Jerusalem, 2002.

[133] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Sys.*, 1(2):245–257, 1979.

[134] T. Nipkow. Term rewriting and beyond—Theorem proving in Isabelle. *Formal Asp. Comput.*, 1:320–338, 1989.

[135] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[136] T. Nipkow. Verifying a hotel key card system. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC 2006*, volume 4281 of *LNCS*, pages 1–14. Springer, 2006.

[137] T. Nipkow. Social choice theory in HOL: Arrow and Gibbard–Satterthwaite. *J. Autom. Reasoning*, 43(3):289–304, 2009.

[138] T. Nipkow. Re: [isabelle] A beginner's questionu [sic]. `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2010-November/msg00097.html`, Nov. 2010.

[139] T. Nipkow. A tutorial introduction to structured Isar proofs. `http://www21.in.tum.de/dist/Isabelle/doc/isar-overview.pdf`, 2011.

[140] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[141] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 335–367. Elsevier, 2001.

[142] F. H. Obermeyer. *Automated Equational Reasoning in Nondeterministic $\lambda$-Calculi Modulo Theories $\mathcal{H}^*$*. Ph.D. thesis, Dept. of Mathematics, Carnegie Mellon University, 2009.

[143] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP 2011*, volume 6898 of *LNCS*, pages 363–369. Springer, 2011.

[144] S. Owre. Random testing in PVS. In *AFM '06*, 2006.

[145] L. C. Paulson. Set theory for verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.

[146] L. C. Paulson. Set theory for verification: II. Induction and recursion. *J. Autom. Reasoning*, 15(2):167–215, 1995.

[147] L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, pages 23–47. MIT Press, 1997.

[148] L. C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction—Essays in Honour of Robin Milner*, pages 187–212. MIT Press, 2000.

[149] L. C. Paulson. Private communication, Nov. 2010.

[150] L. C. Paulson. Private communication, Dec. 2010.

[151] L. C. Paulson. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. PAAR-2010, 2010.

[152] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, E. Ternovska, and S. Schulz, editors, *IWIL-2010*, 2010.

[153] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

[154] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[155] F. Pfenning. Analytic and non-analytic proofs. In R. E. Shostak, editor, *CADE-7*, volume 170 of *LNCS*, pages 393–413. Springer, 1984.

[156] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Inf. Comput.*, 178(1):279–293, 2002.

[157] R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *TYPES '93 (Selected Papers)*, volume 806 of *LNCS*, pages 313–332. Springer, 1994.

[158] V. Prevosto and U. Waldmann. SPASS+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *ESCoR 2006*, volume 192 of *CEUR Workshop Proceedings*, pages 18–33. CEUR-WS.org, 2006.

[159] W. Pugh. The Java memory model is fatally flawed. *Concurrency—Practice and Experience*, 12(6):445–455, 2000.

[160] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. `http://goedel.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`, 2006.

[161] W. Reif and G. Schellhorn. Theorem proving in large theories. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume III: Applications, pages 225–241. Kluwer, 1998.

[162] W. Reif, G. Schellhorn, and A. Thums. Flaw detection in formal specifications. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR 2001*, volume 2083 of *LNCS*, pages 642–657. Springer, 2001.

[163] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Comm.*, 15(2-3):91–110, 2002.

[164] P. Rudnicki and J. Urban. Escape to ATP for Mizar. In P. Fontaine and A. Stump, editors, *PxTP-2011*, 2011.

[165] J. M. Rushby. Tutorial: Automated formal methods with PVS, SAL, and Yices. In D. V. Hung and P. Pandya, editors, *SEFM 2006*, page 262. IEEE, 2006.

[166] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In Z. Shao and B. C. Pierce, editors, *POPL 2009*, pages 379–391. ACM, 2009.

[167] S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *IJCAR 2004*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[168] J. M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.

[169] H. Schütz and T. Geisler. Efficient model generation through compilation. In M. A. McRobbie and J. K. Slaney, editors, *CADE-13*, volume 1104 of *LNAI*, pages 433–447. Springer, 1996.

[170] J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In J. Vitek, editor, *ECOOP 2008*, volume 5142 of *LNCS*, pages 27–51. Springer, 2008.

[171] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, and M. Pollet. Proof development with ΩMEGA: The irrationality of $\sqrt{2}$. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, volume 28 of *Applied Logic*, pages 271–314. Springer, 2003.

[172] J. K. Slaney. FINDER: Finite domain enumerator system description. In A. Bundy, editor, *CADE-12*, volume 814 of *LNAI*, pages 798–801. Springer, 1994.

[173] G. Snelting and D. Wasserrab. A correctness proof for the Volpano/Smith security typing system. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/VolpanoSmith.shtml`, 2008.

[174] M. E. Stickel. Schubert's steamroller problem: Formulations and solutions. *J. Autom. Reasoning*, 2(1):89–101, 1986.

[175] M. E. Stickel, R. J. Waldinger, M. R. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *CADE-12*, volume 814 of *LNAI*, pages 341–355. Springer, 1994.

[176] G. Sutcliffe. ToFoF. `http://www.cs.miami.edu/~tptp/ATPSystems/ToFoF/`.

[177] G. Sutcliffe. The TPTP problem library: TPTP v5.0.0. `http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml`.

[178] G. Sutcliffe. System description: SystemOnTPTP. In D. McAllester, editor, *CADE-17*, volume 1831 of *LNAI*, pages 406–410. Springer, 2000.

[179] G. Sutcliffe. The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[180] G. Sutcliffe. The TPTP World—Infrastructure for automated reasoning. In E. M. Clarke and A. Voronkov, editors, *LPAR-16*, volume 6355 of *LNAI*, pages 1–12. Springer, 2010.

[181] G. Sutcliffe. The CADE-23 automated theorem proving system competition—CASC-23. *AI Comm.*, 25(1):49–63, 2012.

[182] G. Sutcliffe, C. Chang, L. Ding, D. McGuinness, and P. P. da Silva. Different proofs are good proofs. In D. McGuinness, A. Stump, G. Sutcliffe, and C. Tinelli, editors, *EMSQMS 2010*, pages 1–10, 2010.

[183] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP typed first-order form with arithmetic. In N. Bjørner and A. Voronkov, editors, *LPAR-18*, volume 7180 of *LNCS*, pages 406–419. Springer, 2012.

[184] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP data-exchange formats for automated theorem proving tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, volume 112 of *Frontiers in Artificial Intelligence and Applications*, pages 201–215. IOS Press, 2004.

[185] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In E. Yahav, editor, *SAS 2011*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.

[186] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *JELIA 2004*, volume 3229 of *LNCS*, pages 641–653. Springer, 2004.

[187] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[188] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In B. G. Zorn and A. Aiken, editors, *PLDI 2010*, pages 341–350. ACM Press, 2010.

[189] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. Submitted.

[190] D. A. Turner. A new implementation technique for applicative languages. *Softw. Pract. Exper.*, 9:31–49, 1979.

[191] J. Urban. MaLARea: A metasystem for automated reasoning in large theories. In G. Sutcliffe, J. Urban, and S. Schulz, editors, *ESARLT 2007*, volume 257 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[192] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskočil. MaLARea SG1—Machine learner for automated reasoning with semantic guidance. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *LNAI*, pages 441–456. Springer, 2008.

[193] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng. J.*, 10(2):203–232, 2003.

[194] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(3):167–187, 1996.

[195] D. Walker. Substructural type systems. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–44. MIT Press, 2005.

[196] M. Wampler-Doty. A complete proof of the Robbins conjecture. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Robbins-Conjecture.shtml`, 2010.

[197] T. Weber. A SAT-based Sudoku solver. In G. Sutcliffe and A. Voronkov, editors, *LPAR 2005 (Short Papers)*, pages 11–15, 2005.

[198] T. Weber. *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T.U. München, 2008.

[199] T. Weber. SMT solvers: New oracles for the HOL theorem prover. *J. Softw. Tools Technol. Transfer*, 13(5):419–429, 2011.

[200] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1965–2013. Elsevier, 2001.

[201] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *TPHOLs 1997*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.

[202] M. Wenzel. Isabelle/Isar—A generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007.

[203] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *PLMMS 2009*, pages 13–29. ACM Digital Library, 2009.

[204] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. Sacerdoti Coen and D. Aspinall, editors, *UITP '10*, 2010. To appear in *Electr. Notes Theor. Comput. Sci.*

[205] C. A. Wick and W. W. McCune. Automated reasoning about elementary point-set topology. *J. Autom. Reasoning*, 5(2):239–255, 1989.

[206] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS 2004*. IEEE, 2004.

[207] J. Zhang and H. Zhang. SEM: A system for enumerating models. In C. S. Mellish, editor, *IJCAI-95*, volume 1, pages 298–303. Morgan Kaufmann, 1995.

[208] J. Zimmer, A. Meier, G. Sutcliffe, and Y. Zhan. Integrated proof transformation services. In C. Benzmüller and W. Windsteiger, editors, *IJCAR WS 7*, 2004.