Technische Universität München
Institut für Informatik

Proving Theorems of Higher-Order Logic with SMT Solvers

Sascha Böhme

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. Felix Brandt

Prüfer der Dissertation:

1. Univ.-Prof. Tobias Nipkow, Ph.D.
2. Univ.-Prof. Dr. Andrey Rybalchenko

Die Dissertation wurde am 24. 10. 2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05. 03. 2012 angenommen.

Zusammenfassung

Der Nachfrage nach erhöhter Beweisautomatisierung für den interaktiven Theorembeweiser Isabelle folgend zeigt diese Arbeit, wie SMT-Löser den Stand der Technik verbessern können. SMT-Löser sind automatische Theorembeweiser mit Entscheidungsverfahren für verschiedene Theorien wie zum Beispiel Gleichheit und lineare Arithmetik. Unsere Hauptbeiträge sind eine korrekte Übersetzung von Isabelles höherstufiger Logik in die erststufige Logik von SMT-Lösern sowie die effiziente Rekonstruktion von Beweisen, die vom SMT-Löser Z3 gefunden wurden. Mittels einer umfangreichen Evaluierung belegen wir, dass viele Theoreme nun automatisch und beinahe augenblicklich bewiesen werden können, wofür früher zeitraubendes Nachdenken seitens des Benutzers nötig war. Das Überprüfen von Z3-Beweisen ist dank unserer aufwendigen Optimierungen schnell. Weitere Beiträge sind ein neues Werkzeug und eine neue Methode, um funktionale Korrektheit von C-Code im Zusammenspiel mit dem automatischen Programmbeweiser VCC zu beweisen. Wir demonstrieren ihre Eignung für Implementierungen aus der Praxis anhand von Baum- und Graphalgorithmen.

Abstract

Following the demand for increased proof automation in the interactive theorem prover Isabelle, this thesis describes how satisfiability modulo theories (SMT) solvers improve on the state of the art. SMT solvers are automatic theorem provers with decision procedures for several theories such as equality and linear arithmetic. Our main contributions are a sound translation from Isabelle's higher-order logic to the first-order logic of SMT solvers, and efficient checking of proofs found by the solver Z3. Based on a large evaluation, we find that many theorems can now be proved automatically and almost instantaneously for which time-consuming user ingenuity was previously required. Checking Z3's proofs is fast thanks to our extensive optimizations. Further contributions are a new tool and methodology to verify the functional correctness of C code in conjunction with the automatic program verifier VCC. We demonstrate their suitability on real-world implementations of tree and graph algorithms.

Acknowledgments

First and foremost, I am deeply indepted to Tobias Nipkow who introduced me to the great world of theorem proving and who invited me to join his group. I appreciate his guidance and his questions as well as the possibility to always ask him for advice that together broadened my knowledge beyond the topics of this thesis.

Many thanks go to all members of the Isabelle group at TUM: Clemens Ballarin, Stefan Berghofer, Jasmin Christian Blanchette, Lukas Bulwahn, Amine Chaieb, Johannes Hölzl, Dongchen Jiang, Cezary Kaliszyk, Alexander Krauss, Ondřej Kunčar, Peter Lammich, Lars Noschinski, Steven Obua, Andrei Popescu, Thomas Türk, Christian Urban and Markus Wenzel. They all contributed to the pleasant and productive atmosphere of the group and also to great after-work activities. To some of my colleagues I owe special thanks. Stefan Berghofer patiently spent hours in answering all my questions related to Isabelle, Makarius tought me to appreciate good software engineering, Jasmin motivated me to consider practical solutions instead of overly general and artificial ones, and Alex made me understand many things that I had not even known before. I thank Alex, Jasmin and Andrei for reading and commenting earlier versions of this thesis. In addition, Jasmin deserves special thanks for his advices on language and written style that contributed to the readability of this thesis.

Several parts of this thesis are the product of joint work. I have collaborated with Eyad Alkassar, Rustan Leino, Kurt Mehlhorn, Michał Moskal, Larry Paulson, Christine Rizkallah, Wolfram Schulte, Thomas Sewell, Tjark Weber and Burkhart Wolff. In addition, Nikolaj Bjørner and Leonardo de Moura provided many insights into Z3. Mark Hillebrand and Dirk Leinenbach helped me understand VCC. I thank them all. Nikolaj invited me to a short research visit at MSR, and Michał invited me to a research internship at MSR, and I am grateful to both of them for these opportunities.

Finally, my parents, my brother and Jana deserve special thanks for supporting me, especially pushing me to finish this thesis, and accepting that I spent only little time with them during the last months.

This thesis was partly supported by BMBF under grant 01IS07008.

Contents

Chapter 1.

Introduction

Contents

1.1. Motivation

Interactive theorem provers such as Isabelle/HOL [130], HOL4 [78] and Coq [20] rely on automatic methods to discharge simple proof steps without much user guidance. More precisely, showing that a theorem holds with such a prover requires the user to find a proof sketch whose intermediate steps can be established by the prover's automatic methods. These methods, too, must be chosen and, in most cases, also configured by the user. With more powerful automation at hand, proof sketches can be more coarse-grain and hence users need to consider fewer invocations of proof methods resulting in higher productivity. Although in Isabelle, for instance, there are already quite a few automatic methods, there is a consistent demand for even more powerful automation.

A common approach is to pass proof obligations to external theorem provers. Satisfiability modulo theories (SMT) solvers are a new class of automatic theorem provers that combine decision procedures for diverse theories such as equality and arithmetic with provers for propositional satisfiability (SAT). Thanks to recent improvements in the design of SAT solvers that yielded dramatic performance gains, SMT solvers attracted much attention. Current major SMT solvers are highly efficient and solve typical problems within a few seconds only. They even partly outperform saturation-based automatic first-order provers, which have been integrated into Isabelle for years. This makes SMT solvers promising candidates to increase the proof automation in interactive theorem provers such as Isabelle.

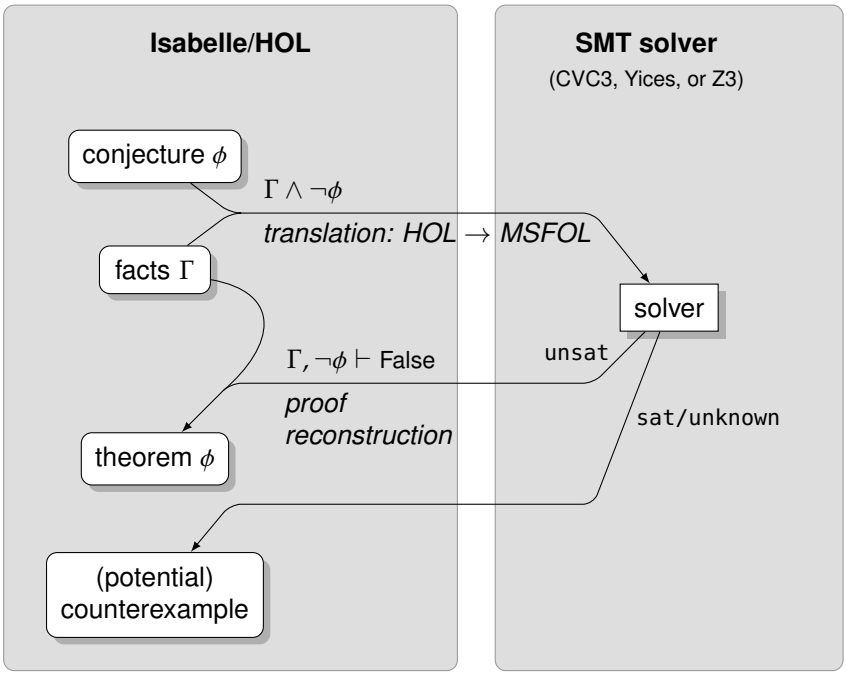


Figure 1.1. The SMT solver integration in Isabelle/HOL

1.2. Contributions

We developed a generic interface to SMT solvers in Isabelle/HOL with the expectation that they can efficiently prove many typical theorems in Isabelle/HOL and thus contribute to better proof automation. Figure 1.1 gives an overview of our integration that consists of a translation from Isabelle's higher-order logic (HOL, Section 1.3.1) to the SMT solvers' first-order logic (MSFOL, Section 1.4.1) and the reconstruction of proofs found by the SMT solver Z3 with inference rules of Isabelle/HOL (Section 1.3.2). The translation from HOL to MSFOL builds on existing work, but our combination of these techniques is genuine, especially with the support of different decision procedures that are provided by SMT solvers (Section 1.4.2). Proof checking for Z3 has not been attempted before, largely because the proof format of Z3 is vaguely specified. We provide in-depth documentation for this format as well as an efficient implementation for its reconstruction in Isabelle/HOL. Counterexamples produced by SMT solvers are beyond the scope of this thesis.

Developing a proof method that invokes external solvers requires to solve several technicalities, and without empirical tests it not clear whether our cooperation with SMT solvers works well in practice. To this end, we contribute an extensive evaluation on a large number of test cases comprising typical applications of Isabelle on which we also demonstrate the efficiency of proof reconstruction for Z3. We found that our integration is fast and robust and can automatically prove many theorems beyond the reach of methods provided by Isabelle so far.

While developing and testing our proof reconstruction for Z3, we have found several soundness bugs in Z3. We reported these to the developers of Z3 and they have been corrected in current versions of the solver. This confirms the persisting suspicion, if not fear, of bugs in automated provers, and motivates the checking of proofs.

We underpin the usefulness of our SMT solver integration in Isabelle by three examples, two of which are derived from user applications.

First example The integer recurrence relation $x_{i+2} = |x_{i+1}| - x_i$ has period 9. This property can be stated in Isabelle/HOL as follows:

$$\begin{aligned} &x_3 = |x_2| - x_1 \wedge x_4 = |x_3| - x_2 \wedge x_5 = |x_4| - x_3 \wedge x_6 = |x_5| - x_4 \wedge \\ &x_7 = |x_6| - x_5 \wedge x_8 = |x_7| - x_6 \wedge x_9 = |x_8| - x_7 \wedge x_{10} = |x_9| - x_8 \wedge \\ &x_{11} = |x_{10}| - x_9 \longrightarrow x_1 = x_{10} \wedge x_2 = x_{11} \end{aligned}$$

When passing this property from Isabelle/HOL through our binding to an SMT solver, the solver almost instantaneously finds a proof, and proof reconstruction for Z3 takes only a few seconds. In contrast, Isabelle's arithmetic decision procedure requires several minutes to prove the same result. Hence, our integration of SMT solvers is considerably faster than existing proof automation.

Second example With our integration of SMT solvers, also higher-order theorems (Section 1.3.1) can be proved. For example, consider the map function for lists over arbitrarily typed elements which is characterized as follows:

$$\begin{aligned} &\forall f.\ \mathsf{map}\, f\, \mathsf{Nil} = \mathsf{Nil} \\ &\forall f, x, xs.\ \mathsf{map}\, f\, (\mathsf{Cons}\, x\, xs) = \mathsf{Cons}\, (f\, x)\, (\mathsf{map}\, f\, xs) \end{aligned}$$

With these two theorems, our integration of SMT solvers in Isabelle is able to prove the following higher-order theorem instantaneously:

$$\mathsf{map}\, (\lambda x^{\mathsf{int}}.\ x + 2)\, (\mathsf{Cons}\, y\, (\mathsf{Cons}\, 3\, \mathsf{Nil})) = \mathsf{Cons}\, (y + 2)\, (\mathsf{Cons}\, 5\, \mathsf{Nil})$$

Hence, our work is applicable to a wide range of conjectures in Isabelle/HOL and succeeds for conjectures that are syntactically outside the variant of first-order logic which is understood by the target SMT solvers.

Third example A user new to Isabelle defined a simple recursive datatype and an inductive predicate over pairs of this type [129]. He failed to find a proof for a simple property of this predicate. Tobias Nipkow provided a structured eight-line Isabelle proof that invokes three different proof methods. This proof is straightforward, but it required a few minutes to put all pieces together. In contrast, it takes only a few seconds for an SMT solver to automatically find a (one-line) Isabelle proof when invoked with our integration through *sledgehammer* (Section 1.3.3). Hence, our integration of SMT solvers can find proofs where other automatic proof methods fail.

In addition to the SMT solver interface in Isabelle/HOL, we developed a tool, HOL-Boogie, and a methodology, abstract cooperation, for program verification of C code. Both build on the VCC code verifier [44] and our integration of SMT solvers in Isabelle. HOL-Boogie is intended to interactively debug and prove verification conditions for which fully automatic attempts with VCC fail. With abstract cooperation, high-level properties of programs can be proved for which VCC has only little support.

We demonstrate both HOL-Boogie and abstract cooperation on a case study each. For the former, we study functional verification of binary search trees, and with the latter, we describe the verification of a checker for a graph algorithm where we uncovered an implementation bug.

1.3. Isabelle/HOL

Isabelle/HOL [130] is an interactive theorem prover based on higher-order logic (Section 1.3.1). It builds on an LCF-style kernel (Section 1.3.2) and has hence only a small trusted code base and high correctness guarantees.

While most users perceive Isabelle/HOL as an interactive theorem prover, we deviate from this view and instead focus on the automation of proofs. To understand the impact of our work, it is nevertheless helpful to get a glimpse on the typical interaction with Isabelle. When faced with a conjecture, a user outlines a proof by breaking the reasoning into smaller "simple" or "obvious" steps and selects proof methods provided by Isabelle/HOL to automatically discharge these steps. Such a proof outline is then checked by Isabelle by running the specified proof methods and reporting unprovable steps back to the user, who in turn likely modifies the proof and repeats this cycle. Hence, the burden of proof is put on the shoulders of the user. Besides the ingenuity required to find a suitable proof outline, a user also needs to build up experience in deciding which automatic method of Isabelle/HOL is appropriate for a particular part of a proof. Consequently, the support for automated proof finding is commonly perceived as too weak to allow a wide adoption of Isabelle. In recent years, more and more help has been developed to assist the user in interacting with the system, and one particular tool for automated proof finding is *sledgehammer* (Section 1.3.3).

1.3.1. Higher-Order Logic

The logic underlying Isabelle/HOL is *higher-order logic* (HOL) [3, 77] that is based on Church's simply-typed λ-calculus [42]. The syntax of HOL consists of the following two entities:

- *Types* τ are either type variables α or applications of type constructors κ^n with fixed arity n:

  $$\tau \ ::= \ \alpha \mid \kappa^n \, \tau_1 \ldots \tau_n$$

  Among the infinite set of type constructors, we identify the binary function type constructor $\rightarrow^2$, typically written infix, and the types of Boolean values bool^0, nat-

ural numbers nat^0 and integers int^0. In what follows, we usually omit the superscripts if they are clear from the context. We refer to a type constructor with arity $n > 0$ applied to n types as a *compound* type. A type without type variables is called *monomorphic* type, and a type with type variables is called *schematic* type.

- *Terms* t are either typed variables x^τ, typed constructors c^τ, application or typed λ-abstraction:

$$t \ ::= \ x^\tau \ \mid \ c^\tau \ \mid \ t_1\, t_2 \ \mid \ \lambda x^\tau.\, t$$

  Special constants are the usual logical connectives and quantifiers, e.g. $\mathsf{False}^{\mathsf{bool}}$ (falsehood), $\neg^{\mathsf{bool}\to\mathsf{bool}}$ (negation), $\wedge^{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$ (conjunction), $\longrightarrow^{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$ (implication), $\mathsf{If}^{\mathsf{bool}\to\alpha\to\alpha\to\alpha}$ (condition) and $\forall^{(\alpha\to\mathsf{bool})\to\mathsf{bool}}$ (universal quantifier) as well as the polymorphic equality $=^{\alpha\to\alpha\to\mathsf{bool}}$. In what follows, we omit the type superscripts if they are clear from the context. We refer to a term of type bool as a *proposition*. A term that contains a variable or constant of schematic type is called schematic term. Quantifying over variables is expressed as application of a binder to an abstraction. For instance, the proposition $(\forall x^\tau.\, t)$ is syntactic sugar for the term $\forall\ (\lambda x^\tau.\, x)$. A sequence of quantifiers is combined into a *cluster*, e.g., instead of $(\forall x.\ (\forall y.\ t))$ we simply write $(\forall x,\, y.\, t)$. This clustering syntax applies in the same manner to λ-abstractions. The term $(\mathsf{If}\ t_1\ t_2\ t_3)$ is usually written as $(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3)$. We abbreviate $\neg\,(t = u)$ by $t \neq u$. To save parentheses, we apply the typical precedence rules where quantifiers extend to the right as much as possible, where infix operators bind weaker than any other application and where conjunction binds stronger than implication.

We refrain from presenting the semantics of HOL. See [77] instead where HOL's semantics is given in terms of set theory. We also refer to transition relations of the λ-calculus [8], especially β-reduction and η-expansion, and to notions such as free or bound variables of a term without specifying them here.

From now on, we assume that all terms are well-typed according to standard typing rules. Especially, if t is of type $\tau' \to \tau$ and t' is of type τ' then $t\ t'$ has type τ, and if t is of type τ' then $(\lambda x^\tau.\, t)$ has type $\tau \to \tau'$.

Type variables and schematic types give rise to the following notions. A *type substitution* σ is a finite mapping from type variables $\alpha_1, \ldots, \alpha_n$ to types $\tau_1, \ldots \tau_n$. It can be lifted to a mapping from types to types and to a mapping from terms to terms by recursion over the structure of types and terms. If the substitution σ is not the identity mapping and $\sigma(t)$ differs from the term t, then we call t a *(type) generalization* of the term $\sigma(t)$. Now, let τ be a schematic type and let τ' be another type. If there exists a substitution σ with $\sigma(\tau) = \tau'$, then we say that τ' matches τ, and we call σ the *substitution induced by* τ' *w.r.t.* τ. Let c^τ and $c^{\tau'}$ be two variants of a constant with types as before. If τ' matches τ with substitution σ, then we say that $c^{\tau'}$ matches c^τ, and if τ' is monomorphic we call $c^{\tau'}$ a *monomorphic variant* of c and c^τ a *schematic variant* of c. Similar as before, we call σ the *substitution induced by* $c^{\tau'}$ *w.r.t.* c^τ. We say that two substitutions σ_1 and σ_2 agree with each other if they map all type variables that are in the domain of both σ_1 and σ_2 to the same type. If σ_1 and σ_2 agree with each other, we define the *combination*

of substitutions σ_1 and σ_2 as the substitution σ that maps every type variable α in the domain of either σ_1 and σ_2 to either $\sigma_1(\alpha)$ or $\sigma_2(\alpha)$.

We fix the following further notations. A possible occurrence of the term u in t is denoted by $t[u]$. Its generalization, the occurrence of a list of terms $u_1, \ldots, u_n$, abbreviated by $\overline{u}$, in a term t is denoted by $t[\overline{u}]$. A replacement of every occurrence of u_1 with u_2 in t is denoted by $t[u_1 \mapsto u_2]$. If the term $c\ t_1\ \ldots\ t_n$ has type $\tau_1 \to \tau_2$, then c is called *partially applied*. Otherwise, if the type of $c\ t_1\ \ldots\ t_n$ is not a function type, then c is said to be *fully applied*. We call a HOL proposition a *problem* if it takes the form of an implication from a conjunction of HOL propositions, called *constituents* of the problem, to False. Since the conclusion of a problem is fixed, we typically drop it in examples.

Besides bound variables, Isabelle supports *free variables* that are arbitrary but fixed. Free variables resemble constants in most cases as they cannot be instantiated, but in contrast to constants, free variables can be quantified over under certain restrictions.[1]

1.3.2. LCF-Style Kernel and Proof Automation

The term LCF-style [75, 76] describes theorem provers that are based on a small inference kernel. Theorems are implemented as an abstract datatype, and the only way to construct new theorems is through a fixed set of functions, that correspond to the underlying logic's axiom schemata and inference rules, provided by this datatype. This design greatly reduces the trusted code base. Proof procedures based on an LCF-style kernel cannot produce unsound theorems, as long as the implementation of the theorem datatype is correct.

Isabelle, just like most LCF-style systems, implements a natural deduction calculus. Theorems are *sequents* $\Gamma \vdash t$, where Γ is a finite set of *hypotheses*, and t is the sequent's *conclusion*. Instead of $\emptyset \vdash t$, we simply write $\vdash t$. For its LCF-style kernel [136], Isabelle uses the special implication constant $\Longrightarrow$ and the special universal quantifier $\bigwedge$. Some of the inference rules of this kernel are assumption (asm), implication introduction ($\text{intro}_{\Longrightarrow}$) and introduction and elimination of the universal quantifier ($\text{intro}_{\bigwedge}$ and $\text{elim}_{\bigwedge}$):

$$\frac{}{\{t\} \vdash t}\ \text{asm} \qquad \frac{\Gamma \vdash t}{\Gamma \setminus \{u\} \vdash u \Longrightarrow t}\ \text{intro}_{\Longrightarrow} \qquad \frac{\Gamma \vdash t[x]}{\Gamma \vdash \bigwedge x.\ t[x]}\ \text{intro}_{\bigwedge} \qquad \frac{\Gamma \vdash \bigwedge x.\ t[x]}{\Gamma \vdash t[x \mapsto u]}\ \text{elim}_{\bigwedge}$$

Introduction of the universal quantifier is restricted to variables x that do not occur free in the hypotheses Γ. The inference rule used most in Isabelle is higher-order resolution res that is an extension of first-order resolution based on higher-order unification. We refer to [135] for a description of this inference rule.

The special implication constant and the special universal quantifier allow to express inference rules of the logic as theorems. Symmetry of equality, for instance, is

[1] Isabelle also knows of schematic term variables that are roughly comparable to outermost universally bound variables. Throughout this thesis, we refrain from using the former in favor of the latter. Similarly to term variables, Isabelle also distinguishes between schematic and free type variables. In this thesis, we treat free type variables interchangable to type constructors most of the time.

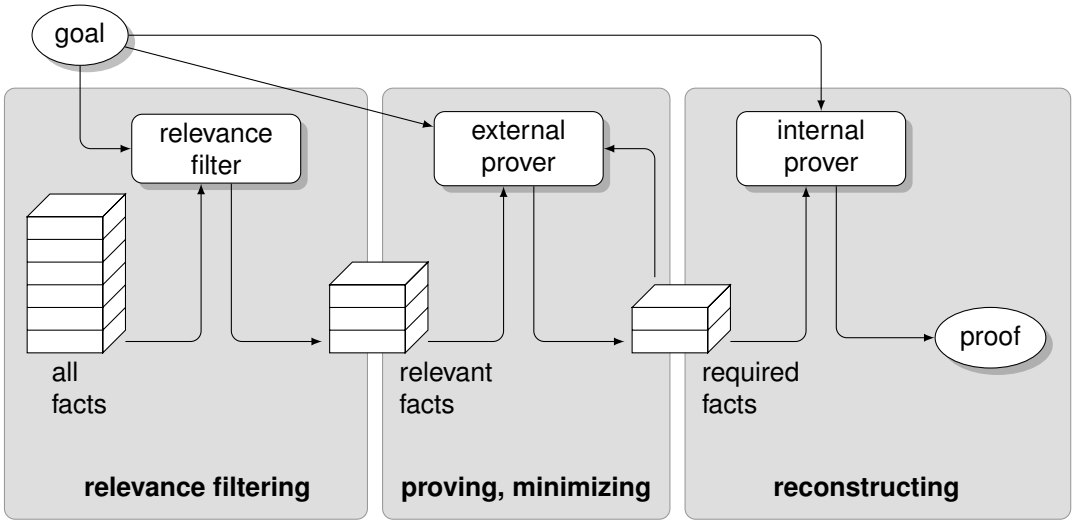


Figure 1.2. The *sledgehammer* architecture

typically written in textbooks as the following inference schema where t and u stand for arbitrary terms:

$$\frac{\Gamma \vdash t = u}{\Gamma \vdash u = t}$$

In Isabelle, symmetry is instead expressed by the theorem $\bigwedge x, y.\ x = y \Longrightarrow y = x$. We call such theorems *metatheorems* to emphasize their role as inference rules. Since elimination of implication and universal quantifiers are primitive inferences of the LCF-style kernel, reasoning with metatheorems is typically much faster than proving specific instances of such theorems anew.

On top of its LCF-style kernel, Isabelle offers various automated *proof methods*, notably a simplifier, which performs term rewriting, a decision procedure for propositional logic, tableau- and resolution-based first-order provers, and decision procedures for arithmetic on integers and reals. Since they ultimately rely on the inference rules of the LCF-style kernel, none of these proof methods extends the trusted code base. Any attempt to perform an unsound inference will be caught by Isabelle's kernel. Yet, there is a special inference rule in the kernel to turn arbitrary propositions into theorems, but Isabelle keeps track of the theorems whose proofs are tainted (directly or indirectly) by such an inference. Proof methods that use this technique are called *proof oracles* and are typically mistrusted for good reasons.

1.3.3. Sledgehammer

Originally developed to interface Isabelle/HOL with resolution-based automatic theorem provers (ATPs) such as E, SPASS and Vampire, *sledgehammer* [118] has eventually evolved into a more general prover framework [23] as depicted in Figure 1.2. It integrates several different external provers with Isabelle/HOL and delivers both auto-

matic proofs of internal provers such as the first-order resolution prover *metis* [86, 139] as well as Isar [165] proofs. The central component of *sledgehammer* is a relevance filter [119] that heuristically selects for a given proof goal (or conjecture) from all facts of Isabelle/HOL a few hundred relevant facts potentially required to prove that goal. These facts together with the goal are given in parallel to external provers. In case one of them finds a proof *sledgehammer* extracts and optionally minimizes the set of facts required by the proof and passes these facts together with the goal to an internal prover such as *metis* to find an LCF-style proof. Alternatively *sledgehammer* can construct a structured Isar proof from the proof of one of the external provers [139].

1.4. Satisfiability Modulo Theories Solvers

Satisfiability modulo theories (SMT) solvers are automatic theorem provers for first-order logic with a combination of theories such as equality, linear arithmetic and fixed-size bitvectors [37, 83, 97]. The research about SMT solvers receives active interest, and many solvers have emerged in recent years. Some major SMT solvers are CVC3 [17], Yices [64], and Z3 [59]. We give insights into the architecture of SMT solvers (Section 1.4.2) after introducing the logic understood by these solvers (Section 1.4.1).

1.4.1. Many-Sorted First-Order Logic

Many-sorted first-order logic (MSFOL) [111, chapter 6] is an extension of standard first-order logic [3] with sorts. The language of MSFOL comprises the following three syntactic categories:

- *Sorts* σ are atomic entities. The function space from sorts $\sigma_1, \ldots, \sigma_n$ to a sort σ is written as $(\sigma_1, \ldots, \sigma_n) \to \sigma$. The domain of a relation over sorts $\sigma_1, \ldots, \sigma_m$ is denoted as $(\sigma_1, \ldots, \sigma_m)$. Both function space and domain of a relation (for $m > 1$) are themselves no sorts.

- *Terms* t are either sorted variables x^σ or functions $f^{(\sigma_1,\ldots,\sigma_n)\to\sigma}$ applied to terms:

$$t \;::=\; x^\sigma \;\mid\; f^{(\sigma_1,\ldots,\sigma_n)\to\sigma}\,(t_1, \ldots, t_n)$$

- *Formulas* φ are composed of logical constants (e.g., falsehood $\bot$), logical connectives (e.g., negation $\neg$ and conjunction $\wedge$), quantifiers (e.g., the universal quantifier $\forall$), polymorphic equality $=$ on terms and predicates $P^{(\sigma_1,\ldots,\sigma_n)}$ applied to terms:

$$\varphi \;::=\; \bot \;\mid\; \neg\varphi \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; \forall x^\sigma.\, \varphi \;\mid\; t_1 = t_2 \;\mid\; P^{(\sigma_1,\ldots,\sigma_n)}\,(t_1, \ldots, t_n)$$

  Moreover, we allow the logical constant $\top$ (truth), the logical connectives $\vee$ (disjunction), $\longrightarrow$ (implication), $\longleftrightarrow$ (equivalence) and the existential quantifier $\exists$. Formulas that are either equalities between two terms or predicates applied to terms are called *atoms*. Similarly to HOL, we abbreviate $\neg\,(t_1 = t_2)$ by $t_1 \neq t_2$, and we group quantifiers in clusters, i.e., we write $(\forall x, y.\, \varphi)$ instead of $(\forall x.\ (\forall y.\, \varphi))$.

In addition, we allow if-then-else-expressions in terms and formulas. More precisely, if t_1 and t_2 are terms and φ, ψ_1 and ψ_2 are formulas then $(\mathsf{if}\ \varphi\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$ is a term and $(\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2)$ is a formula.

The semantics of MSFOL is given in [111] as an extension of the semantics of standard first-order logic. Note that MSFOL can be reduced to unsorted first-order logic.

Similarly to HOL (Section 1.3.1), we assume that all terms and formulas are well-sorted. Especially if all t_i are of sort σ_i, then $f^{(\sigma_1,\ldots,\sigma_n)\to\sigma}(t_1,\ldots,t_n)$ has sort σ and $P^{(\sigma_1,\ldots,\sigma_n)}(t_1,\ldots,t_n)$ is well-sorted. Moreover, if t_1 and t_2 are both of sort σ, then $t_1 = t_2$ is well-sorted, i.e., any two terms of the same sort can be compared for equality. Subsequently, we omit superscripts where the sorts are clear from the context.

For presentation purposes, we use l to denote a formula for which negation never creates a doubly-negated formula, and we call such a formula a *literal*. That is, if l stands for the formula $\neg\varphi$, then $\neg l$ denotes φ, and if l stands for an unnegated formula ψ, than $\neg l$ denotes $\neg\psi$. A disjunction of literals is henceforth called *clause*.

A language element commonly supported by SMT solvers is the polyadic distinct predicate, which abbreviates a quadratic number of inequations. For example, the formula $\mathsf{distinct}(t_1, t_2, t_3)$ is short for $t_1 \neq t_2 \wedge t_1 \neq t_3 \wedge t_2 \neq t_3$. Note that $\mathsf{distinct}(t)$ for any term t is equivalent to $\top$.

Two MSFOL formulas are *equisatisfiable*, denoted by $\sim$, if the first formula is satisfiable whenever the second one is, i.e., each formula has a model or none has one. Since both formulas can have different models, equisatisfiability is a generalization of equivalence. Yet in some cases, the existential closures of two equisatisfiable formulas can result in equivalent formulas, e.g., $(P(x) \vee \bot)$ is equisatisfiable to $P(y)$ and their existential closures $(\exists x.\ P(x) \vee \bot)$ and $(\exists y.\ P(y))$ are equivalent. There are also cases for which the existential closures of two equisatisfiable formulas do not yield equivalent formulas. For instance, the existential closure for Q is Q, and likewise the existential closure for $Q \vee R$ is $Q \vee R$. The two formulas Q and $Q \vee R$ are equisatisfiable but not equivalent.

Finally, we fix the following notations. The symbol $\rightsquigarrow$ is a placeholder for either implication ($\longrightarrow$), equivalence ($\longleftrightarrow$) or equisatisfiability ($\sim$). The symbols $\simeq$ and $\approx$ stand for either equality ($=$) between two terms or equivalence ($\longleftrightarrow$) or equisatisfiability ($\sim$) between two formulas unless the resulting formulas would not be well-sorted. Moreover, we use Q to denote one of the quantifiers $\forall$ or $\exists$.

1.4.2. SMT Solver Architecture

An SMT solver consists of many "little engines of proof" [147]. Among them, a decision procedure for propositional logic, typically an efficient SAT solver [168], and a decision procedure for the ground theory of equality, typically based on congruence closure [7, 63, 126], are the driving forces. Figure 1.3 gives an overview of the common architecture of SMT solvers. We briefly highlight the depicted components and their interplay.

Preprocessing Any input to an SMT solver is preprocessed first. This includes simplifications of terms and formulas, but also equivalence-preserving transformations

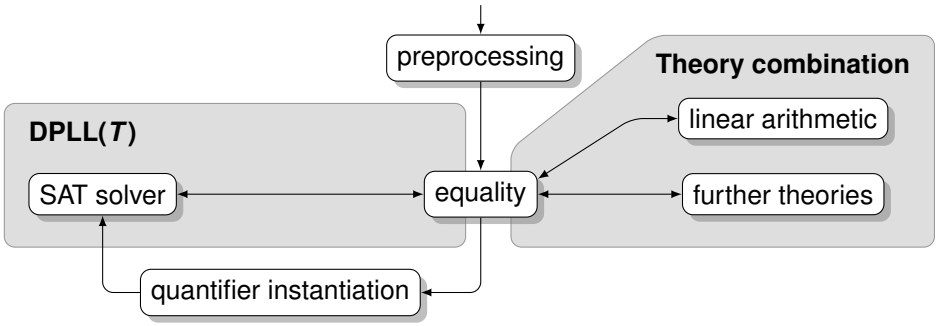


Figure 1.3. High-level SMT solver architecture

that establish certain canonical forms which are expected by the other components, notably the decision procedures.

SAT solver SAT solvers decide satisfiability of a propositional formula in *conjunctive normal form* (CNF), i.e., a conjunction of clauses, by trying to find a model for the formula. Underlying most modern solvers is the classical DPLL solver scheme [54, 55] (named after its inventors Davis, Putnam, Logemann and Loveland) that performs a search based on propagating implications of single-literal clauses and case splitting on literals. Despite the fact that deciding satisfiability of CNF formulas is NP-complete, SAT solvers developed over the last decade perform astonishingly well in practice thanks to several optimizations [123, 149] that are subsumed under the name conflict-driven clause learning (CDCL). Solving formulas with up to tens of thousands of variables and millions of clauses is still feasible in certain cases.

DPLL(*T*) The DPLL(*T*) framework [72, 127] handles the integration of a decision procedure for a ground first-order theory with the search performed by a DPLL-based SAT solver. To this end, each formula is abstracted into a propositional skeleton where atoms over functions and predicates interpreted by the theory are replaced by propositional variables. Whenever the SAT solver is unable to refine a partial model for the propositional skeleton by implications and before it would perform costly case splits, the theory decision procedure is invoked. If that finds the current (partial) model contradictory, it produces a clause, called *theory lemma*, that is conjoined to the propositional skeleton, and the SAT solver continues its search.

Theories Several first-order theories have decision procedures for conjunctions of theory atoms. Among those are equality, decided by congruence closure [7, 63, 126], linear arithmetic over integers, decided by the Omega test [140], and linear arithmetic over reals, decided by the Fourier–Motzkin [52] or the Simplex algorithm [43]. Conjunctions of bitvector atoms where all involved bitvectors are of fixed size can be flattened to propositional formulas, called bitblasting [97], and hence be decided by a SAT solver. Algebraic datatypes have a (potentially infinite) first-order axiomatization using equal-

ity [133]. They are typically handled by a special hard-coded setup of the quantifier instantiation heuristics.

Theory combination Most SMT solvers implement a theory combination suggested by Nelson and Oppen [125]. Given decision procedures for two theories that share only equality as interpreted symbol, the combined decision procedure works as follows. Any conjunction of theory atoms is first purified by introducing equalities between fresh variables and subterms such that each atom of the resulting conjunction contains only interpreted functions and predicates of a single theory. The theory decision procedures are invoked on their respective atoms and communicate with each other by exchanging implied equalities (over variables) only.

Quantifier instantiation Except for preprocessing, all SMT solver components described so far operate only on unquantified formulas. Instead of taking decision procedures for quantified formulas, e.g., linear arithmetic admits quantifier elimination [49], SMT solvers rely mostly on heuristics to instantiate quantifiers [57, 61, 122]. These heuristics are based on *patterns*. A pattern for a quantified formula $\forall \overline{x}.\ \varphi$ is a set of terms $\{t_1, \ldots, t_n\}$ where each term t_i is typically a subterm of φ. Every bound variable in the list $\overline{x}$ must occur in at least one of the terms $t_1, \ldots, t_n$, yet none of these terms must be syntactically equal to one the bound variables. The SMT solver will regularly, mostly before case splitting in the SAT solver, check if a substitution σ exists such that $\sigma(t_1), \ldots, \sigma(t_n)$ have an interpretation in the currently considered partial ground model. If so, the solver conjoins the instance $\sigma(\varphi)$, that is implied by the quantified formula $\forall \overline{x}.\ \varphi$, to the logical context and continues its search. Substitutions are found by matching patterns against the graph data structure produced by the congruence closure algorithm. Hence, a pattern's terms may involve uninterpreted functions only. Functions and predicates interpreted by other theories are excluded from pattern-based quantifier instantiations, because matching patterns against interpreted terms would require theory-specific reasoning which is typically too expensive.

There are cases in which specifying more than one pattern is convenient. We call a set of patterns for a quantified formula a *trigger*. The SMT solvers relevant to this thesis have automatic trigger inference algorithms, but users may override these algorithms, except for that in Yices, by providing hand-selected triggers. This gives users the ability to directly control how the solvers perform quantifier instantiations. In addition, Z3 supports user-configurable *quantifier weights* that influence at which stage of the search a quantified formula is participating. The larger the weight, the later a formula is considered for instantiation. Inappropriate triggers or too large weights may inhibit the corresponding formulas to participate in the solver's search and consequently render a problem unprovable for the SMT solver.

1.5. Structure of This Thesis

The largest part of this thesis is devoted to a sound and efficient integration of SMT solvers with Isabelle/HOL, as depicted in Figure 1.1. We describe how to translate con-

jectures formulated in higher-order logic to the variant of first-order logic understood by SMT solvers, in a way that the translated formula implies the original conjecture, and evaluate this integration on a representative set of examples (Chapter 2). We then present how to certify this integration, or, more specifically, how to efficiently reconstruct proofs produced by the SMT solver Z3 (Chapter 3).

Among the many applications of SMT solvers, program verification is one the most prominent and challenging driving forces [84]. We devote the second part of this thesis to this subject (Chapter 4). We describe the new tool HOL-Boogie that is intended to debug and prove verification conditions for which the automatic C code verifier VCC fails. HOL-Boogie relies on our integration of SMT solvers in Isabelle for doing most of its work. We apply HOL-Boogie to the verification of binary search trees. Moreover, we describe a new methodology called abstract cooperation to prove high-level properties of C programs and demonstrate it on the verification of a C implementation of a checker for a graph algorithm.

Finally, we conclude and give an outlook for future directions (Chapter 5).

Chapter 2.

Integrating SMT Solvers as Oracles

Contents

2.1. Introduction

Our integration of SMT solvers as proof oracles in Isabelle/HOL essentially consists of a translation (Section 2.2) from higher-order logic (HOL) to many-sorted first-order logic (MSFOL), where we deliberately ignore technical intricacies involved in combining heterogeneous applications. Our translation is generic and sound, but not complete. Generality comes from targeting the standardized exchange format SMT-LIB [141] understood by all major SMT solvers. With soundness, we mean that if a translated problem is valid, so is the original problem, whereas completeness means that a translated problem is valid if the original one is. Our translation encodes every HOL proposition into MSFOL formulas as a shallow embedding, i.e., by faithfully representing quantifiers, connectives and other symbols of HOL with respective entities of MSFOL. We use the overloaded translation function $\langle\!\langle \cdot \rangle\!\rangle$ that maps HOL types to MSFOL sorts and HOL terms t to MSFOL terms or MSFOL formulas depending on the type of t. In a less formal way, we complement our description of the translation from HOL to MSFOL by detailing how we make use of the decision procedures available in SMT solvers (Section 2.3) and how to guide the search strategy of SMT solvers (Section 2.4).

An integration of SMT solvers with Isabelle/HOL such as the one described in this chapter will necessarily stay prototypically unless validated on a large benchmark set. We do so by providing results of extensive experiments and thus demonstrating the usefulness of SMT solvers for typical proofs in Isabelle/HOL (Section 2.5). Our evaluation also confirms our initial expectation about the increased proof automation gained from SMT solvers.

$$
\begin{aligned}
\text{Sorts:}\quad & \langle\!\langle \kappa^0 \rangle\!\rangle \cong \sigma \\
\text{Terms:}\quad & \langle\!\langle x^\tau \rangle\!\rangle \cong x^{\langle\!\langle \tau \rangle\!\rangle} \\
& \langle\!\langle c^{\tau_1 \to \ldots \to \tau_n \to \tau}\ t_1\ \ldots\ t_n \rangle\!\rangle \cong c^{(\langle\!\langle \tau_1 \rangle\!\rangle, \ldots, \langle\!\langle \tau_n \rangle\!\rangle) \to \langle\!\langle \tau \rangle\!\rangle}\ (\langle\!\langle t_1 \rangle\!\rangle, \ldots, \langle\!\langle t_n \rangle\!\rangle) \\
\text{Formulas:}\quad & \langle\!\langle \mathsf{False} \rangle\!\rangle \cong \bot \qquad\qquad \langle\!\langle t = u \rangle\!\rangle \cong (\langle\!\langle t \rangle\!\rangle = \langle\!\langle u \rangle\!\rangle) \\
& \langle\!\langle \neg t \rangle\!\rangle \cong \neg \langle\!\langle t \rangle\!\rangle \qquad\qquad \langle\!\langle \forall x^\tau.\ t \rangle\!\rangle \cong \left(\forall x^{\langle\!\langle \tau \rangle\!\rangle}.\ \langle\!\langle t \rangle\!\rangle\right) \\
& \langle\!\langle t \wedge u \rangle\!\rangle \cong \langle\!\langle t_1 \rangle\!\rangle \wedge \langle\!\langle u \rangle\!\rangle \\
& \langle\!\langle c^{\tau_1 \to \ldots \to \tau_n \to \mathsf{bool}}\ t_1\ \ldots\ t_n \rangle\!\rangle \cong c^{(\langle\!\langle \tau_1 \rangle\!\rangle, \ldots, \langle\!\langle \tau_n \rangle\!\rangle)}\ (\langle\!\langle t_1 \rangle\!\rangle, \ldots, \langle\!\langle t_n \rangle\!\rangle)
\end{aligned}
$$

Figure 2.1. Translation of essentially first-order HOL entities to MSFOL equivalents following the description of MSFOL (Section 1.4.1)

Translations from HOL to (variants of) first-order logic have previously been studied. In fact, most of the work presented (Section 2.2) is not new, yet has not been combined in the presented way and accompanied with extensive tests so far. We conclude this chapter by reviewing related work and highlighting differences to our solutions (Section 2.6).

2.2. Translation

We can represent any MSFOL formula directly as a semantically equivalent proposition in HOL, and we call the latter *essentially first-order* to distinguish it from HOL propositions for which no direct MSFOL representations exist. To clarify their correspondence, Figure 2.1 shows a mapping from essentially first-order HOL entities to MSFOL equivalents. Clearly, most HOL propositions are not essentially first-order, especially if they contain type variables or λ-abstractions (except at quantifier constants). For those propositions, we describe a set of syntax-directed translation steps that map them into essentially first-order form. With the correspondence of essentially first-order propositions and MSFOL formulas, this results in a sound translation from HOL to MSFOL.

Which translations are necessary to convert an arbitrary HOL proposition into a corresponding essentially first-order one? We approach an answer to this question by studying the syntactical entities of HOL that have no counterpart in MSFOL.

- The type system of MSFOL lacks both type variables and compound types. For example, the HOL type $\kappa\ \alpha$ has no direct representation as a sort in MSFOL.
- MSFOL lacks λ-abstractions, and hence a HOL term such as $(\lambda x.\ t)$ cannot be directly represented in MSFOL.
- The single incarnation of application in HOL contrasts with several different incarnations of application in MSFOL, i.e., applications of function symbols or predicate symbols (including equality) to terms and applications of logical connectives to formulas. To highlight this issue, let us consider the possible cases of term t in

$$\Big(\forall f^{\alpha\to\beta},\ x^{\alpha},\ xs^{\mathsf{list}\ \alpha}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = f\ x\Big)\ \wedge$$
$$\mathsf{apphd}\ \big(\lambda x^{\mathsf{bool}}.\ \mathsf{if}\ x\ \mathsf{then\ a\ else\ b}\big)\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}^{\kappa}$$

Figure 2.2. A contrived HOL problem as a running example for our translations. All symbols not bound by a quantifier are constants.

a (β-reduced) HOL application $t\ t_1\ \ldots\ t_n$ with $n \geq 1$, where t is neither an application nor an abstraction:

- If t is a variable, it must be be of some function type. Clearly, this is not directly expressible in MSFOL where variables are of atomic sort.
- If t is a constant, then it might either be partially applied or fully applied. The former case has no direct representation in MSFOL, where logical connectives, predicates and functions are always applied to as many arguments as their arity demands. The latter case is possibly directly expressible in MSFOL, but certain restrictions might apply.

MSFOL further imposes a strict distinction between formulas and terms: Variables and functions may only occur in terms; logical connectives, quantifiers and predicates may only occur in formulas; quantifying over propositions (i.e., Boolean variables) is not possible; and especially function and predicate symbols are distinct entities, even if they share the same name—contrast this with HOL constants, whose semantics is independent from their position in a term. Similarly, MSFOL's equality and equivalence are both subsumed by HOL's equality.

The remainder of this section presents our translation steps to convert a HOL problem into essentially first-order form and into MSFOL. Each step concentrates on one of the mentioned points and establishes a specific normal form by erasing some HOL concepts or enforcing some MSFOL restrictions, and these normal forms are maintained by later translations. Showing that the presented translations are sufficient is omitted. We illustrate all translation steps on an (admittedly contrived) example (Figure 2.2).

2.2.1. Monomorphization

Two major approaches are known for translating problems with schematic constituents into problems without type variables. The first approach is to encode the type system into an untyped logic. The second approach, commonly known as monomorphization, is to heuristically instantiate the schematic constituents with monomorphic types and thereby keeping problems typed. Since we intend to benefit from the support of sorts in SMT solvers, we chose the second approach for our translation.

Monomorphization is the repetition of a step that generates all instances of a set of schematic terms based on a set of monomorphic terms until a fixed point is reached. Instances are generated by matching monomorphic constants with schematic constants

Original HOL problem:

$$\Big(\forall f^{\alpha\to\beta},\ x^{\alpha},\ xs^{\mathsf{list}\ \alpha}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = f\ x\Big)\ \wedge$$
$$\mathsf{apphd}\ \big(\lambda x^{\mathsf{bool}}.\ \mathsf{if}\ x\ \mathsf{then\ a\ else\ b}\big)\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}$$

After monomorphization:

$$\Big(\forall f^{\ \mathsf{bool}\ \to\ \kappa},\ x^{\ \mathsf{bool}},\ xs^{\mathsf{list}\ \mathsf{bool}}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = f\ x\Big)\ \wedge$$
$$\mathsf{apphd}\ \big(\lambda x^{\mathsf{bool}}.\ \mathsf{if}\ x\ \mathsf{then\ a\ else\ b}\big)\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}$$

Figure 2.3. The running example after monomorphization

and applying the resulting substitutions to the schematic terms. The set of instances generated by one step may contain new monomorphic terms that induce further instances in the next step. Formally, monomorphization can be defined as follows. Let $c^{\tau'}$ be a constant with monomorphic type τ' and let t be a schematic term. If t contains the constant c^{τ} with schematic type τ such that τ' matches τ, then we call $\sigma(t)$ an *instance of t w.r.t. $c^{\tau'}$* where σ is the substitution induced by c^{τ}. Furthermore, if t' is a monomorphic term, then $\sigma(t)$ is an *instance of t w.r.t. t'* if the combination σ of all substitutions induced by the monomorphic constants in t' is defined. Note that such an instance can still be schematic. Given a set of schematic terms S and a set of monomorphic terms M, we define the *instances of S w.r.t. M* as the set I of terms such that each term in I is an instance of some term from S w.r.t. to some term from M. With $(I_\mathrm{m}, I_\mathrm{s})$ being the partition of I into monomorphic and schematic terms, a *monomorphization step* for S w.r.t. M maps the pair (M, S) to the pair $(M \cup I_\mathrm{m}, S \cup I_\mathrm{s})$. Only taking these unions makes every monomorphization step monotonic, because both M and S might contain terms that are missing in I_m and I_s. For observe that every term that is an instance of a schematic term has strictly less type variables than that schematic term. Hence, there might be terms in S that cannot be obtained as an instance from S w.r.t. M. Observe further that M might contain monomorphic terms for which there is no generalization in S. That is, there might be terms in M that are different from any instance in I_m. The (complete) *monomorphization* of a set S of schematic terms w.r.t. a set M of monomorphic terms is the computation of a least fixed point of monomorphization steps of S w.r.t. M. For a problem t, let (M, S) be the partition of its constituents into monomorphic and schematic terms. If monomorphization of S w.r.t. M yields the pair (M', S'), then we call the conjunction of all terms in M' the *monomorphized problem* of t. Figure 2.3 illustrates the effect of monomorphization on the running example.

Monomorphization can yield a pair whose first component, the set of monomorphic terms, is infinite. Hence, computing that set by repeated monomorphization steps can be nonterminating. For example, consider the sets $S = \{c^{\alpha} \wedge c^{\kappa\ \alpha}\}$ and $M = \{c^{\kappa_0}\}$. The first monomorphization step yields the monomorphic term $c^{\kappa_0} \wedge c^{\kappa\ \kappa_0}$ that triggers

another step resulting in the new monomorphic term $c^{\kappa\,\kappa_0} \wedge c^{\kappa\,\kappa\,\kappa_0}$, and every further step yields a new monomorphic term with a further application of κ to the types of c. We frequently observed similar situations for more realistic problems occurring in Isabelle/HOL. Another example where monomorphization results in an infinite set of monomorphic terms is given in [51, page 265].

Clearly, most elements of the computed set of monomorphic terms are irrelevant if a proof for the monomorphized problem exists, because every proof is finite and can hence refer only to a finite number of monomorphic terms. Finding the finite subset of necessary monomorphic terms is undecidable [27], but we can devise heuristics to obtain an overapproximation of that set, and, in our experience, giving a few hundred unnecessary facts to an SMT solver has little impact on the solver's performance. Our solution is to perform a fixed number of monomorphization steps, because monomorphic terms that contribute to proofs are typically those generated by the first few steps. Since each step can lead to exponentially many new instances, we also limit the total number of computed monomorphic terms. Both numbers are user-configurable. By default, our implementation performs 5 steps and generates up to 300 new monomorphic terms. For efficiency reasons, we operate only on substitutions and postpone their application until after their computation stops. In our implementation, each monomorphization step is a refinement of substitutions starting from the singleton set containing only the empty substitution, by matching monomorphic constants against polymorphic constants. As long as new monomorphic constants emerge, our implementation collects them and performs another step.

2.2.2. Lambda-Lifting

We remove λ-abstractions by *λ-lifting* [92], a technique which introduces new constants for λ-abstractions and adds definitions to relate these new constants to the λ-abstractions they are replacing. More formally, we perform the following translation where c is a fresh constant:

$$\langle\!\langle t\,[\lambda x^{\tau}.\ u] \rangle\!\rangle \ \cong\ (t\,[(\lambda x^{\tau}.\ u) \mapsto c] \wedge (\forall x^{\tau}.\ c\ x = u))$$

Figure 2.4 exemplifies this translation. If the λ-abstraction contains free variables, they are turned into arguments to c.

We apply λ-lifting from inside out, i.e., first replacing inner λ-abstractions before replacing outer ones. For keeping the number of quantified formulas low, which reduces the overhead in the target SMT solver, it is important to minimize the number of freshly introduced constants and corresponding definitions. To this end, we treat a cluster of λ-abstractions such as $(\lambda x,\ y.\ t)$ at once, i.e., by introducing just one constant instead of two in this case. Moreover, we replace repeated occurrences of the same λ-abstraction by the same constant instead of inventing fresh constants for every occurrence again. The number of λ-liftings is reduced further by applying β-reduction to HOL terms before this step.

Result of the previous translation step:

$$\begin{array}{l}(\forall f^{\mathsf{bool}\to\kappa},\ x^{\mathsf{bool}},\ xs^{\mathsf{list\ bool}}.\ \mathsf{apphd}\, f\ (\mathsf{Cons}\ x\ xs) = f\, x)\ \wedge \\ \mathsf{apphd}\ (\lambda x^{\mathsf{bool}}.\ \mathsf{if}\ x\ \mathsf{then\ a\ else\ b})\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\end{array}$$

After λ-lifting:

$$\begin{array}{l}(\forall f^{\mathsf{bool}\to\kappa},\ x^{\mathsf{bool}},\ xs^{\mathsf{list\ bool}}.\ \mathsf{apphd}\, f\ (\mathsf{Cons}\ x\ xs) = f\, x)\ \wedge \\ \mathsf{apphd\ c}\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\ \wedge\ (\forall x^{\mathsf{bool}}.\ \mathsf{c}\ x = \mathsf{if}\ x\ \mathsf{then\ a\ else\ b})\end{array}$$

Figure 2.4. The running example after λ-lifting

2.2.3. Explicit Applications

For constants occurring with a varying number of arguments, e.g., both partially- and fully-applied, we treat the minimal number of arguments as proper arguments and make application to additional arguments explicit. We achieve this with the help of the constant $\mathsf{app}^{(\alpha_1\to\alpha_2)\to\alpha_1\to\alpha_2}$ that has the following defining equation:

$$\mathsf{app}\ t_1\ t_2 = t_1\ t_2$$

Making applications explicit works as follows, where c is a constant that occurs at least with n arguments and where all t_i with $1 \leq i \leq n$ as well as all u_j with $1 \leq j \leq m$ are HOL terms:

$$\langle\!\langle c\ t_1\ \ldots\ t_n\ u_1\ \ldots\ u_m \rangle\!\rangle\ \cong\ \mathsf{app}\ (\ldots(\mathsf{app}\ (t\ t_1\ \ldots\ t_n)\ u_1)\ \ldots)\ u_m$$

The same technique is also applied to higher-order bound variables whose minimal number of arguments is by default taken to be zero. Decorating HOL terms with app by rewriting with the definition of app from right to left is equivalence-preserving. Figure 2.5 demonstrates this translation on our running example.

Constants that are interpreted in MSFOL, e.g., logical connectives, may also occur partially applied in HOL, although this typically happens infrequently. Decorating these constants with app would result in terms which are not well-typed in MSFOL. Instead, we η-expand them before λ-lifting such that the partially applied occurrence is turned into a fresh constant.

Due to our policy of choosing the minimal number of arguments as the arity of constants, introducing explicit applications may result in different representations of HOL problems when only one further constituent is added. In the worst case, an unsolvable problem can turn into a solvable problem, although the added constituent does not directly participate in the proof. This situation is best illustrated by an example. Consider the following HOL problem, where $\mathsf{zero}^{\mathsf{N}}$, $\mathsf{one}^{\mathsf{N}}$ and $\mathsf{succ}^{\mathsf{N}\to\mathsf{N}}$ are constants:

$$\mathsf{succ\ zero} = \mathsf{one}\ \wedge\ \neg\,(\exists f.\ f\,\mathsf{zero} = \mathsf{one})$$

Result of the previous translation step:

$$\begin{array}{l}(\forall f^{\mathsf{bool}\to\kappa},\ x^{\mathsf{bool}},\ xs^{\mathsf{list\ bool}}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = f\ x)\ \wedge \\ \mathsf{apphd\ c}\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\ \wedge\ (\forall x^{\mathsf{bool}}.\ \mathsf{c}\ x = \mathsf{if}\ x\ \mathsf{then\ a\ else\ b})\end{array}$$

After introducing explicit applications:

$$\begin{array}{l}\Big(\forall f^{\mathsf{bool}\to\kappa},\ x^{\mathsf{bool}},\ xs^{\mathsf{list\ bool}}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{app}\ f\ x\Big)\ \wedge \\ \mathsf{apphd\ c}\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\ \wedge\ \Big(\forall x^{\mathsf{bool}}.\ \mathsf{app}\ \mathsf{c}\ x = \mathsf{if}\ x\ \mathsf{then\ a\ else\ b}\Big)\end{array}$$

Figure 2.5. The running example after introducing of explicit applications

This problem is clearly provable in HOL as succ is a witness for f. Following our description above, we make the application of the higher-order variable f explicit and thus obtain this corresponding problem, which is refutable:

$$\mathsf{succ\ zero} = \mathsf{one}\ \wedge\ \neg\,(\exists f.\ \mathsf{app}\ f\ \mathsf{zero} = \mathsf{one})$$

By adding the constituent P succ, which does not contribute to the proof since nothing is known about P, we force succ to be treated as nullary constant according to our translation policy:

$$\mathsf{P\ succ}\ \wedge\ \mathsf{app\ succ\ zero} = \mathsf{one}\ \wedge\ \neg\,(\exists f.\ \mathsf{app}\ f\ \mathsf{zero} = \mathsf{one})$$

This problem is again provable—especially by SMT solvers. To avoid such weaknesses as exposed by this example, we adapted our approach to detecting minimal arities of constants in the following way (thanks to a suggestion by Jasmin Christian Blanchette). If a variable x^τ occurs in a problem, then all constants of type $\tau_1 \to \ldots \tau_n \to \tau$ are considered to have at most arity n unless constrained otherwise as they might be possible witnesses (if x is existentially quantified) or instances (if x is universally quantified).

2.2.4. Erasure of Compound Types

Observe that a HOL type constructor κ^n with $n > 0$ applied to n monomorphic HOL types τ_i has a set interpretation constructed recursively from the set interpretations of the types τ_i. Hence, we can consider $\kappa^n\ \tau_1\ \ldots \tau_n$ as a HOL type κ_0^n with the same set interpretation. This observation allows us to translate compound types to fresh nullary type constructors, which in turn correspond directly to MSFOL sorts:

$$\langle\!\langle \kappa^n\ \tau_1\ \ldots \tau_n \rangle\!\rangle\ \cong\ \kappa_0^n$$

Note that due to monomorphization we can assume that all types of a problem are monomorphic at this step. Further note that this translation also applies to monomorphic function types. Figure 2.6 demonstrates on the running example how compound types are erased.

Result of the previous translation step:

$$(\forall f^{\mathsf{bool}\to\kappa},\ x^{\mathsf{bool}},\ xs^{\mathsf{list\ bool}}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{app}\ f\ x)\ \wedge$$
$$\mathsf{apphd\ c}\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\ \wedge\ (\forall x^{\mathsf{bool}}.\ \mathsf{app\ c}\ x = \mathsf{if}\ x\ \mathsf{then\ a\ else\ b})$$

After erasing compound types:

$$\left(\forall f^{\ \kappa_1},\ x^{\mathsf{bool}},\ xs^{\ \kappa_2}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{app}\ f\ x\right)\ \wedge$$
$$\mathsf{apphd\ c}\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\ \wedge\ (\forall x^{\mathsf{bool}}.\ \mathsf{app\ c}\ x = \mathsf{if}\ x\ \mathsf{then\ a\ else\ b})$$

Figure 2.6. The running example after erasing compound types

2.2.5. Separation of Formulas from Terms

The distinction between the syntactic entity of formulas and that of terms as required by MSFOL is obtained in HOL by inserting vacuous symbols to mark the border between the two categories. To this end, we define the constants $\mathsf{TT}^{\mathsf{tbool}}$ and $\mathsf{FF}^{\mathsf{tbool}}$ equivalent to True and False, where tbool is a new type isomorphic to bool, and use them as follows. A HOL term t of type bool can either be a formula or a term in the MSFOL sense depending on its syntactic structure and its occurrence in a proposition. If t is structurally a term, but its occurrence requires a formula, then t is equated with TT. If t is structurally a formula, but its occurrence requires a term, it is wrapped in a condition $(\mathsf{if}\ t\ \mathsf{then\ TT\ else}\ FF)$. Hence, we treat all HOL constants as MSFOL function symbols and never produce any MSFOL predicate symbols. Figure 2.7 makes this informal description more precise by means of three mutually-recursive translation functions that follow the exposition of MSFOL (Section 1.4.1) extended with if-then-else-expressions. Within HOL, these decorations are equivalence-preserving. The application of this translation to the running example is given in Figure 2.8.

Besides inserting marker symbols into the problem, we conjoin only $(\mathsf{TT} \neq \mathsf{FF})$ to further specify the two new constants. To make the translation complete, we could add the exhaustion rule $(\forall t^{\mathsf{tbool}}.\ t = \mathsf{TT} \vee t = FF)$, but this rule is likely to be ignored by SMT solvers since they fail in finding a trigger for the quantifier (Section 1.4.2).

2.2.6. Translation into MSFOL

After the previous translation steps, a problem is now in essentially first-order form. HOL propositions in essentially first-order form directly correspond to MSFOL formulas. Translation from HOL to MSFOL proceeds by mapping quantifiers, logical connectives, if-then-else-expressions and equality to their MSFOL counterparts (Figure 2.1). The remaining HOL constants are treated as uninterpreted in MSFOL. We highlight this by renaming them, a process which also avoids any potential name confusion in the target SMT solver. In fact, assigning new names to every constant is a cheap imple-

Types: $\langle\!\langle \mathsf{bool} \rangle\!\rangle \cong \mathsf{tbool} \qquad \langle\!\langle \kappa^0 \rangle\!\rangle \cong \kappa^0$

Terms: $\langle\!\langle x^\tau \rangle\!\rangle_\mathrm{T} \cong x^{\langle\tau\rangle} \qquad \langle\!\langle c^\tau\ t_1\ \ldots\ t_n \rangle\!\rangle_\mathrm{T} \cong c^{\langle\tau\rangle}\ \langle\!\langle t_1 \rangle\!\rangle_\mathrm{T}\ \ldots\ \langle\!\langle t_n \rangle\!\rangle_\mathrm{T}$

$\langle\!\langle \mathsf{if}\ t\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2 \rangle\!\rangle_\mathrm{T} \cong (\mathsf{if}\ \langle\!\langle t \rangle\!\rangle_\mathrm{F}\ \mathsf{then}\ \langle\!\langle u_1 \rangle\!\rangle_\mathrm{T}\ \mathsf{else}\ \langle\!\langle u_2 \rangle\!\rangle_\mathrm{T})$

$\langle\!\langle t \rangle\!\rangle_\mathrm{T} \cong (\mathsf{if}\ \langle\!\langle t \rangle\!\rangle_\mathrm{F}\ \mathsf{then\ TT\ else\ FF}) \qquad (1)$

$\langle\!\langle t = u \rangle\!\rangle_\mathrm{T} \cong (\mathsf{if}\ (\langle\!\langle t \rangle\!\rangle_\mathrm{X} = \langle\!\langle u \rangle\!\rangle_\mathrm{X})\ \mathsf{then\ TT\ else\ FF}) \qquad (2)$

Formulas: $\langle\!\langle x^\tau \rangle\!\rangle_\mathrm{F} \cong \left(x^{\langle\tau\rangle} = TT\right) \qquad \langle\!\langle c^\tau\ t_1\ \ldots\ t_n \rangle\!\rangle_\mathrm{F} \cong \left(c^{\langle\tau\rangle}\ \langle\!\langle t_1 \rangle\!\rangle_\mathrm{T}\ \ldots\ \langle\!\langle t_n \rangle\!\rangle_\mathrm{T} = \mathsf{TT}\right)$

$\langle\!\langle \mathsf{if}\ t\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2 \rangle\!\rangle_\mathrm{F} \cong (\mathsf{if}\ \langle\!\langle t \rangle\!\rangle_\mathrm{F}\ \mathsf{then}\ \langle\!\langle u_1 \rangle\!\rangle_\mathrm{F}\ \mathsf{else}\ \langle\!\langle u_2 \rangle\!\rangle_\mathrm{F})$

$\langle\!\langle \mathsf{False} \rangle\!\rangle_\mathrm{F} \cong \mathsf{False} \qquad \langle\!\langle \neg t \rangle\!\rangle_\mathrm{F} \cong \neg\langle\!\langle t \rangle\!\rangle_\mathrm{F} \qquad \langle\!\langle t \wedge u \rangle\!\rangle_\mathrm{F} \cong \langle\!\langle t \rangle\!\rangle_\mathrm{F} \wedge \langle\!\langle u \rangle\!\rangle_\mathrm{F}$

$\langle\!\langle \forall x^\tau.\ t \rangle\!\rangle_\mathrm{F} \cong \left(\forall x^{\langle\tau\rangle}.\ \langle\!\langle t \rangle\!\rangle_\mathrm{F}\right) \qquad \langle\!\langle t = u \rangle\!\rangle_\mathrm{F} \cong (\langle\!\langle t \rangle\!\rangle_\mathrm{X} = \langle\!\langle u \rangle\!\rangle_\mathrm{X}) \qquad (2)$

Figure 2.7. Translation rules for separating formulas and terms loosely following the description of MSFOL (Section 1.4.1) and extended with if-then-else-expressions. Additional side conditions apply: (1) A term t is translated as a formula wrapped into a condition if t is of the form False, $\neg t'$, $t_1 \wedge t_2$, or $\forall x.\ t'$. (2) The variable X in $\langle\!\langle t \rangle\!\rangle_\mathrm{X}$ stands for F if t is of type bool and for T otherwise.

Result of the previous translation step:

$$(\forall f^{\kappa_1},\ x^{\mathsf{bool}},\ xs^{\kappa_2}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{app}\ f\ x)\ \wedge$$
$$\mathsf{apphd\ c}\ (\mathsf{Cons\ True\ Nil}) \neq \mathsf{a}\ \wedge\ (\forall x^{\mathsf{bool}}.\ \mathsf{app\ c}\ x = \mathsf{if}\ x\ \mathsf{then\ a\ else\ b})$$

After separating formulas and terms:

$$\left(\forall f^{\kappa_1},\ x^{\boxed{\mathsf{tbool}}},\ xs^{\kappa_2}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{app}\ f\ x\right)\ \wedge$$
$$\mathsf{apphd\ c}\ \left(\mathsf{Cons}\ \boxed{(\mathsf{if\ True\ then\ TT\ else\ FF})}\ \mathsf{Nil}\right) \neq \mathsf{a}\ \wedge$$
$$\left(\forall x^{\boxed{\mathsf{tbool}}}.\ \mathsf{app\ c}\ x = \mathsf{if}\ \boxed{(x = TT)}\ \mathsf{then\ a\ else\ b}\right)\ \wedge\ \boxed{TT \neq FF}$$

Figure 2.8. The running example after separating formulas and terms

Result of the previous translation step:

$$\begin{array}{l}\big(\forall f^{\kappa_1},\ x^{\mathsf{tbool}},\ xs^{\kappa_2}.\ \mathsf{apphd}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{app}\ f\ x\big)\ \wedge \\ \mathsf{apphd\ c}\ \big(\mathsf{Cons}\ \big(\mathsf{if\ True\ then\ TT\ else\ FF}\big)\ \mathsf{Nil}\big) \neq \mathsf{a}\ \wedge \\ \big(\forall x^{\mathsf{tbool}}.\ \mathsf{app\ c}\ x = \mathsf{if}\ (x = \mathsf{TT})\ \mathsf{then\ a\ else\ b}\big)\ \wedge\ \mathsf{TT} \neq FF\end{array}$$

After translation into MSFOL:

$$\begin{array}{l}\Big(\forall x_1^{\sigma_1},\ x_2^{\sigma_2},\ x_3^{\sigma_3}.\ \mathsf{f}_1\ x_1\ \Big(\mathsf{f}_2\ x_2\ x_3\Big) = \mathsf{f}_3\ x_1\ x_2\Big)\ \wedge \\ \mathsf{f}_1\ \ \mathsf{f}_4\ \ \Big(\mathsf{f}_2\ \ \Big(\mathsf{if}\ \top\ \mathsf{then}\ \mathsf{f}_5\ \mathsf{else}\ \mathsf{f}_6\Big)\ \ \mathsf{f}_7\Big) \neq \mathsf{f}_8\ \wedge \\ \Big(\forall x^{\sigma_2}.\ \mathsf{f}_3\ \ \mathsf{f}_4\ x = \mathsf{if}\ \Big(x = \mathsf{f}_5\Big)\ \mathsf{then}\ \mathsf{f}_8\ \mathsf{else}\ \mathsf{f}_9\Big)\ \wedge\ \mathsf{f}_5 \neq \mathsf{f}_6\end{array}$$

Figure 2.9. The running example after translation into MSFOL

mentation trick to avoid namespace handling, and it does not harm as in general the output has to be understood only by automatic tools. Renaming also gives some flexibility, as we can freely choose which further constants are to be treated as interpreted or uninterpreted. Figure 2.9 illustrates this translation on the running example.

2.3. Theories and Interpreted Constants

Besides equality, most current SMT solvers also decide other quantifier-free first-order theories, notably linear arithmetic over integers and reals, but also bitvectors and even algebraic datatypes. This opens the door to use the SMT solver's built-in interpretation for those constants from Isabelle/HOL that share the same semantics—instead of leaving them uninterpreted. In addition, we η-expand partially applied occurrences of built-in constants first, in the same way as we deal with partially applied logical connectives (Section 2.2.3).

Integer and real arithmetic Numerals, addition, negation, subtraction and linear multiplication (i.e., numerals multiplied with arbitrary terms) as well as comparisons for integers and reals have direct counterparts in SMT solvers (Section 1.4.2). Nonlinear multiplication is generally treated as uninterpreted, because SMT solvers usually lack suitable decision procedures for it, and the specification of the SMT-LIB format underlying our communication with the target SMT solvers restricts multiplication to linear cases only. The solver Yices, for instances, blindly rejects nonlinear problems, although support for nonlinear multiplication is typically not required to solve them. We found it sufficient that users manually select extra facts providing enough information for SMT solvers to handle uninterpreted nonlinear arithmetic.

For Z3, we are more liberal and interpret more arithmetic constants. In particular, our translation maps nonlinear multiplication as well as division over reals to the corresponding Z3 functions. Moreover, we exploit Z3's division and modulo over integers. Since their semantics does unfortunately not conform to that of Isabelle/HOL, we relate Isabelle/HOL's integer division and modulo with those of Z3 by means of complex case distinctions and add these formalizations to the problem given to Z3.

The maximum and minimum of two integers or reals as well as the absolute value of an integer or real have no built-in interpretation in SMT solvers. We unfold their HOL definitions, which make use of interpreted constants, before translating propositions. Alternatively, we could have decided to conjoin their definitions to a problem, but this would add further quantifiers and might impair the SMT solver's performance unnecessarily.

Natural numbers Natural numbers are typically not supported by SMT solvers. Since natural numbers are exactly the nonnegative integers, we treat the former as an abstract type isomorphic to the latter. To this end, we introduce the two coercions nat and int, characterized by the following three rules:

$$\forall n^{\mathsf{nat}}.\ \mathsf{nat}\ (\mathsf{int}\ n) = n$$
$$\forall i^{\mathsf{int}}.\ 0 \leq i \longrightarrow \mathsf{int}\ (\mathsf{nat}\ i) = i$$
$$\forall i^{\mathsf{int}}.\ i < 0 \longrightarrow \mathsf{int}\ (\mathsf{nat}\ i) = 0$$

With those two helper functions, we can coerce terms of type nat into terms of integer type, apply interpreted functions on integers and coerce the result back to the uninterpreted natural numbers. Functions on natural numbers that have integer representations are the numerals, the successor function Suc and the usual arithmetic operations addition, subtraction and multiplication. The minimum and maximum of two natural numbers are unfolded similar to their integer counterparts. In addition, the arguments to the natural number comparison predicates $<$ and $\leq$ are coerced to integers to make use of interpreted integer comparison. We restrict the introduction of coercions to these cases. In particular, we keep bound natural number variables unchanged despite the possibility that an SMT solver could exploit its quantifier elimination on integers. Moreover, a term t of type nat is kept undecorated, i.e., we avoid to wrap it into coercions $\mathsf{nat}\ (\mathsf{int}\ t)$ to not spoil the search in the target SMT solver.

Figure 2.10 depicts the translation function $\langle\!\langle\cdot\rangle\!\rangle_{\mathrm{N}}$ and the additional translation function $\langle\!\langle\cdot\rangle\!\rangle_{\mathrm{I}}$ that together translate the mentioned natural number operations completely into integer ones. More precisely, they replace every non-negative numeral M^{nat} by a coerced integer number $\mathsf{nat}\ M^{\mathsf{int}}$. Built-in operations on natural numbers are lifted to integer operations. Uninterpreted functions and bound variables are kept unchanged except for the decoration with necessary coercions. For example, the term $f\ (3^{\mathsf{nat}} + n + m) < m - 2^{\mathsf{nat}} \cdot n$ is translated into:

$$\mathsf{int}\ \big(f\ \big(\mathsf{nat}\ \big(3^{\mathsf{int}} + \mathsf{int}\ n + \mathsf{int}\ m\big)\big)\big) < \mathsf{int}\ \big(\mathsf{nat}\ \big(\mathsf{int}\ m - 2^{\mathsf{int}} \cdot \mathsf{int}\ n\big)\big)$$

$$
\begin{array}{rcl@{\qquad}rcl}
\langle\!\langle n_1 = n_2 \rangle\!\rangle_{\mathrm{N}} & \cong & \langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} = \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}} & & & \\
\langle\!\langle n_1 < n_2 \rangle\!\rangle_{\mathrm{N}} & \cong & \langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} < \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}} & & & \\
\langle\!\langle n_1 \leq n_2 \rangle\!\rangle_{\mathrm{N}} & \cong & \langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} \leq \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}} & & & \\
\langle\!\langle M^{\mathsf{nat}} \rangle\!\rangle_{\mathrm{N}} & \cong & \mathsf{nat}\ M^{\mathsf{int}} & \langle\!\langle M^{\mathsf{int}} \rangle\!\rangle_{\mathrm{I}} & \cong & M^{\mathsf{int}} \\
\langle\!\langle \mathsf{Suc}\ n \rangle\!\rangle_{\mathrm{N}} & \cong & \mathsf{nat}\ (\langle\!\langle n \rangle\!\rangle_{\mathrm{I}} + 1^{\mathsf{int}}) & \langle\!\langle \mathsf{Suc}\ n \rangle\!\rangle_{\mathrm{I}} & \cong & \langle\!\langle n \rangle\!\rangle_{\mathrm{I}} + 1^{\mathsf{int}} \\
\langle\!\langle n_1 + n_2 \rangle\!\rangle_{\mathrm{N}} & \cong & \mathsf{nat}\ (\langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} + \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}}) & \langle\!\langle n_1 + n_2 \rangle\!\rangle_{\mathrm{I}} & \cong & \langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} + \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}} \\
\langle\!\langle n_1 - n_2 \rangle\!\rangle_{\mathrm{N}} & \cong & \mathsf{nat}\ (\langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} - \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}}) & \langle\!\langle n_1 - n_2 \rangle\!\rangle_{\mathrm{I}} & \cong & \mathsf{int}\ (\mathsf{nat}\ (\langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} - \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}})) \\
\langle\!\langle n_1 \cdot n_2 \rangle\!\rangle_{\mathrm{N}} & \cong & \mathsf{nat}\ (\langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} \cdot \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}}) & \langle\!\langle n_1 \cdot n_2 \rangle\!\rangle_{\mathrm{I}} & \cong & \langle\!\langle n_1 \rangle\!\rangle_{\mathrm{I}} \cdot \langle\!\langle n_2 \rangle\!\rangle_{\mathrm{I}} \\
\langle\!\langle x \rangle\!\rangle_{\mathrm{N}} & \cong & x & \langle\!\langle x \rangle\!\rangle_{\mathrm{I}} & \cong & \mathsf{int}\ x \\
\langle\!\langle c\ t_1\ \ldots\ t_k \rangle\!\rangle_{\mathrm{N}} & \cong & c\ \langle\!\langle t_1 \rangle\!\rangle_{\mathrm{N}}\ \ldots\ \langle\!\langle t_k \rangle\!\rangle_{\mathrm{N}} & \langle\!\langle c\ t_1\ \ldots\ t_k \rangle\!\rangle_{\mathrm{I}} & \cong & \mathsf{int}\ (c\ \langle\!\langle t_1 \rangle\!\rangle_{\mathrm{N}}\ \ldots\ \langle\!\langle t_k \rangle\!\rangle_{\mathrm{N}})
\end{array}
$$

Figure 2.10. Translation operations on natural number to integers, where M stands for a nonnegative numeral and n, n_1 as well as n_2 are short for terms of type nat. The translation functions are shown only for atoms and subterms thereof. The extension to full HOL terms is straightforward.

After the application of $\langle\!\langle \cdot \rangle\!\rangle_{\mathrm{N}}$ and $\langle\!\langle \cdot \rangle\!\rangle_{\mathrm{I}}$ to a problem, every occurrence of one of the natural number operation that has an interpretation on the integers in SMT solvers is translated to the corresponding integer operation. As a side effect, the two translation functions introduce as less coercions as possible by pushing the coercions as much down in terms as possible. Together with the characterization of the coercion functions above, this translation from natural numbers to integers is equivalence-preserving.

Bitvectors The formalization of machine words in Isabelle/HOL [56] and the fixed-size bitvector functions and predicates provided by SMT solvers [97] share the same semantics. Jointly with Thomas Sewell, we modified the SMT integration in Isabelle/HOL to make use of this automation. We directly map most relevant HOL constants to their built-in counterparts in SMT solvers. HOL functions that map words to integers and vice versa are treated as uninterpreted, since SMT solvers typically do not support cooperation between these two theories (Section 1.4.2).

Datatypes, records and type definitions Definitional principles to introduce new types in Isabelle/HOL comprise algebraic datatypes, record types, i.e., tuples with named selection and update functions for each component, and type definitions [130]. Since they, especially algebraic datatypes, are pervasive in Isabelle/HOL formalizations, we decided to support them directly. Although several SMT solvers provide suitable decision procedures, we only exploit that of Z3 as it is the only one that is accessible with our target exchange format SMT-LIB. Other SMT solvers expose their support for datatypes only via special input formats, and we chose to not sacrifice generosity of our SMT binding in exchange for extra decision procedures.

When translating datatypes, records and type definitions, we map each of them directly to Z3's notion of algebraic datatypes whose declarations are based on constructors and selectors [133]. For instance, the type of integer lists is declared as follows in Z3 using SMT-LIB syntax:

```
datatype ((IntList Nil (Cons (head Int) (tail IntList))))
```

`Nil` is the empty list and `Cons` is the constructor that adds one integer to the front of a list, and `head` and `tail` are partial selectors, i.e., applicable only to nonempty lists.

Internally, Z3 treats such datatypes as uninterpreted sorts associated with uninterpreted constructors and uninterpreted selectors, but with special hard-coded axiom schemata that express injectivity of selectors and acyclicity of constructors. Instantiating these quantified axioms is hard-wired in Z3 and hence circumvents the expensive quantifier instantiation. Note that acyclicity has no finite first-order axiomatization, and hence recursive datatypes benefit from the built-in support in Z3. In contrast, reasoning about nonrecursive datatypes, for example those to which we map Isabelle/HOL records and type definitions, gains only little. In fact, when Z3 is invoked as external prover of *sledgehammer* (Figure 1.2), there is no need for a special treatment of records and type definitions by Z3 since *sledgehammer* will in most cases automatically select the relevant facts.

Further interpreted constants SMT solvers provide further constructs also available in HOL, and translating them to MSFOL representations can help applications, e.g., HOL-Boogie (Section 4.3). Our translation supports let-expressions to share common subterms and distinct applied to concrete lists, abbreviating a quadratic number of inequations between the list elements.

2.4. Extra-Logical Information

Many SMT solvers provide means to control their reasoning algorithms. Especially the intricate and typically incomplete quantifier instantiation can be influenced. Applications that rely heavily on quantifiers such as, for example, in program verification (Chapter 4), can benefit substantially by decorating quantifiers with extra-logical information in the form of triggers and weights (Section 1.4.1) that control the instantiation heuristics.

We support such decorations by providing specific constants that are added to HOL terms while preserving their semantics. We keep these decorations in place throughout the translation steps described earlier (Section 2.2). Some examples of HOL terms decorated with triggers and weights are shown in Figure 2.11.

Triggers Triggers are lists of patterns associated with a quantifier cluster, where each pattern is itself a list of terms marked as either positive (with marker pat, enabling instantiations) or negative (with marker nopat, inhibiting instantiations). A trigger decoration for HOL terms is defined as a binary projection function trigger that ignores its

$$\forall f^{\alpha_1 \to \alpha_2},\ x_1^{\alpha},\ xs^{\mathsf{list}\ \alpha_1}.\ \mathsf{trigger}\ [[\mathsf{pat}\ (\mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs))]]$$
$$(\mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs))$$

$$\forall n^{\mathsf{int}},\ m^{\mathsf{int}}.\ \mathsf{trigger}\ [[\mathsf{pat}\ (\mathsf{even}\ n), \mathsf{pat}\ (\mathsf{even}\ m)]]\ (\mathsf{even}\ n \wedge \mathsf{even}\ m \longrightarrow \mathsf{even}\ (n + m))$$

$$\forall x^{\mathsf{real}},\ y^{\mathsf{real}}.\ \mathsf{trigger}\ [[\mathsf{pat}\ (\mathsf{times}\ x\ y)]]\ (\mathsf{weight}\ 2\ (\mathsf{times}\ x\ y = \mathsf{times}\ y\ x))$$

Figure 2.11. HOL terms decorated with triggers and weights. The second term has two marked terms—corresponding to the two premises—as pattern. The third term is decorated both with a trigger and a weight.

first argument, which carries the list of patterns. Hence, the exact definitions for the marker functions pat and nopat are irrelevant. We let them map to some arbitrary value. Although the trigger constant may occur anywhere in terms, we only handle it specially during translation to MSFOL when directly enclosed by a quantifier cluster.

Weights Weights are nonnegative integers associated with a quantifier cluster. We encode them in much the same way as triggers by defining a binary projection function weight to its second argument. Its first argument is an integer specifying the weight. The weight constant may occur anywhere in a term, but we only handle it specially when enclosed by a quantifier cluster or a trigger.

Decorating terms automatically Triggers and weights are powerful tools to guide and control the target SMT solvers, but finding the "right" triggers and weights is an art [122]. Choosing the "wrong" values might render a solvable problem unsolvable, i.e., the solver returns with an inconclusive answer, simply because it fails to find the necessary quantifier instances (Section 1.4.2). To slightly relieve users from this burden, we implemented a basic trigger inference algorithm that enforces facts which express equalities or equivalences to be applied as rewrite rules (from left to right) by choosing the left-hand side as trigger. For weights, we do not provide any automated inference. Instead, we leave it to other tools, e.g., the relevance filter of *sledgehammer* (Section 1.3.3), to find suitable weight values.

2.5. Evaluation

We integrated our binding to SMT solvers as the *smt* method in Isabelle, and it got first publicly available with Isabelle2009 in December 2009. Shortly thereafter, Geoff Sutcliffe tried out this integration as part of IsabelleP[1] on a collection of higher-order

[1] IsabelleP is Geoff Sutcliffe's private setup that lets Isabelle act as an automatic theorem prover by combining most of the automatic proof methods available.

problems of the TPTP [151, 152]. In a private conversation (from December 17, 2009), he showed his surprise about the efficiency of the *smt* method:

> *Wow, the new SMT tactic has really improved the automatic IsabelleP. I put it last in the list of tactics tried . . .*
>
> *simp, blast, auto, metis, fast, fastsimp, best, force, meson, smt*
>
> *. . . and it solved 161 of the total 1590 solved, i.e., the preceding nine tactics failed to solve those 161 problems. Of course SMT is surely able to solve many of the 1429 problems solved by one of the earlier tactics. That's great progress for automated higher-order ATP!*

In the beginning of 2011, Yuan Gao conducted similar experiments with the higher-order problems from the TPTP collection and found that, out of then 2826 problems, the above list of methods could prove 1693 when each method is given a timeout of 60 seconds. The *smt* method alone solved 1292 problems, nearly 30% more than the next best method, and thus contributed 178 genuinely proved problems, i.e., about 10% of all proved problems. Its median runtime was 109 milliseconds, slightly longer than the median runtimes of the other nine Isabelle methods due to the overhead of invoking an external SMT solver.

Since these figures give only a rough, although promising estimate about the capabilities of SMT solvers and our integration of them with Isabelle/HOL, we conducted more experiments to validate our high expectations. Jointly with Jasmin Christian Blanchette, we measured the success rates of different SMT solvers and compared them with ATPs, i.e., resolution-based automated theorem provers (Section 2.5.2) on a representative set of developments in Isabelle/HOL (Section 2.5.1). We also evaluated to which extent particular decision procedures and extra-logical information contribute to the performance of SMT solvers in the context of Isabelle/HOL (Section 2.5.3).

2.5.1. Experimental Setup

Instead of considering problems from the TPTP collection, which come from various sources and exploit only a subset of Isabelle's language features, we chose examples representative for Isabelle/HOL and directly reflecting the benefits for users of the system, albeit with a slight bias towards arithmetic. To this end, we selected nine Isabelle theories (listed below) with altogether 1591 proof goals. Seven of these theories have already been considered in an earlier evaluation of *sledgehammer* with ATPs only [32], and the remaining two theories were chosen due to their extensive use of arithmetic on which we intend to measure the SMT solvers' abilities. The last two columns in the overview below give the percentage of the goals that come from each Isabelle theory and the features it contains, where A means arithmetic, I means induction and recursion, L means λ-abstractions, and S means sets.

Arrow	Arrow's impossibility theorem		L S	6.3%
FFT	Fast Fourier transform	A	L	9.1%
FTA	Fundamental theorem of algebra	A		26.6%
Hoare	Completeness of Hoare logic with procedures	A I	L	12.8%
Jinja	Type soundness of a subset of Java		I L	11.4%
NS	Needham–Schroeder shared-key protocol		I	6.2%
QE	DNF-based quantifier elimination	A	L S	12.0%
S2S	Sum of two squares	A		8.1%
SN	Strong normalization of the typed λ-calculus	A I		7.2%

At every individual proof goal of these nine theories, we applied the *smt* method with the solvers CVC3 (version 2.2), Yices (version 1.0.28) and Z3 (version 2.15), wrapped in *sledgehammer* calls to select relevant facts (Section 1.3.3). We fed those facts, typically a few hundred, together with the corresponding proof goal to the SMT solvers for deciding validity within a time limit of 30 seconds. In case of success, we reduced the number of relevant facts to those necessary for the proof, typically less than ten, in two ways. For Z3 we directly extracted the necessary facts from the proof certificates produced, i.e., we distilled these certificates into the corresponding unsatisfiable cores (Section 3.2). For CVC3 and Yices, due to the lack of proof certificates, we allotted further 30 seconds for iterative narrowing of the originally selected facts. With the thus reduced set of facts, we conducted a final checking round in which we applied first *metis* with a timeout of one second and then Z3 to find a valid proof in the sense of Isabelle/HOL (Section 3.3). Only if that final round succeeded, we considered a proof goal successfully proved.

It is worth noting that this setup measures more than just success rates of oracles based on external SMT solvers. We also evaluate and check the applied translations for their soundness, and we check reconstructability of Z3 proofs in Isabelle.

All experiments were conducted on one core of a dual core Xeon processor machine running with 3 GHz and 2 GB RAM.

2.5.2. Benefits from SMT Solvers

In a previous extensive study using a subset of the Isabelle/HOL theories above [32], *sledgehammer* based on ATPs has been found to be of great help. In fact, nearly half of all goals could be proved automatically. An obvious question is whether SMT solvers can further improve upon this proof automation. To this end, we conducted our evaluation as a comparison between the aforementioned SMT solvers and the ATPs E (version 1.2), SPASS (version 3.7), Vampire (version 1.0), and SInE (version 0.4), all of them working as backends to *sledgehammer* in much the same way as the SMT solvers. Similarly as for the SMT solvers, we chose 30 seconds as time limit for the ATPs, which is the default with *sledgehammer* and reflects the typical amount of time users of Isabelle will wait (at most) when asking external tools for proofs.

Table 2.1 shows the success rates of the SMT solvers and the cumulative results of all ATPs for the selected Isabelle/HOL theory. We see that SMT solvers perform remarkably well. More than half of all goals can be solved automatically by SMT solvers,

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All	Uniq.
CVC3	36%	18%	53%	51%	37%	29%	21%	57%	55%	41.8%	1.3%
Yices	29%	18%	51%	51%	37%	31%	22%	**59%**	59%	41.7%	.9%
Z3	**48%**	18%	**62%**	**54%**	**47%**	**42%**	**25%**	58%	**62%**	**48.5%**	**5.8%**
SMT	**50%**	**23%**	66%	**65%**	**48%**	42%	27%	**66%**	63%	**52.4%**	8.8%
ATPs	40%	21%	**68%**	55%	37%	**46%**	**31%**	55%	**70%**	49.9%	6.3%
All	55%	28%	73%	66%	48%	50%	41%	73%	72%	58.7%	–

Table 2.1. Success rates on all goals

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All	Uniq.
CVC3	23%	14%	28%	36%	31%	18%	7%	25%	27%	24.3%	2.1%
Yices	11%	14%	30%	40%	33%	20%	7%	26%	44%	25.4%	1.5%
Z3	**36%**	13%	**41%**	**46%**	**46%**	**34%**	7%	**28%**	**46%**	**33.0%**	**9.7%**
SMT	**37%**	18%	**43%**	**54%**	**46%**	34%	8%	**33%**	48%	**35.8%**	8.9%
ATPs	32%	18%	42%	42%	33%	**38%**	**19%**	26%	**59%**	33.8%	6.9%
All	41%	23%	50%	57%	46%	44%	23%	42%	61%	42.7%	–

Table 2.2. Success rates on "nontrivial" goals only

especially by Z3. CVC3 and Yices are comparable, but Z3 outperforms them in many cases, which is also indicated by the number of goals uniquely proved by Z3—5.8% of all goals solved by Z3 are neither proved by CVC3 nor Yices. We found that, in many cases, SMT solvers solve more goals than ATPs, and for some theories the best SMT solver Z3 alone solves more goals than the combination of four ATPs. Concluding that SMT solvers are in general better suited to problems stemming from Isabelle/HOL is, however, premature as *sledgehammer* invokes the ATPs with a different amount and choice of facts due to a different configuration of the selection heuristics—an aspect we ignore to evaluate here—and uses an entirely different encoding into first-order logic for them than what we described earlier (Section 2.2). When looking at the five theories that involve λ-abstractions, we notice, however, that SMT solvers seem to be better on higher-order problems because they outperform ATPs on three relevant theories by 10 percentage points and are about as good as or only slightly worse than ATPs on the remaining two theories. This seems to indicate that λ-lifting is more suitable for SMT solvers than combinators are for ATPs [118]. Nevertheless, due to their differences and differences of the applied translations, both classes, ATPs and SMT solvers, complement each other: SMT solvers fail to prove 6.3% of all goals, for which ATPs found a proof, but SMT solvers also contribute 8.8% proved goals not found by any ATP. This demonstrates the usefulness for SMT solvers as additional automatic proof support for Isabelle/HOL.

When trying to prove a goal, Isabelle users typically apply a handful of methods (e.g., *auto* and *simp*) without arguments. Goals where this approach is successful are considered "trivial", and about one third of the proof goals from the chosen nine Isabelle/HOL theories fall into this category. Although being "trivial" for users of Isabelle, it is worth noting that such goals can still pose challenges for automated reasoners, but

this will be of no concern to us here. Only if a goal is "nontrivial", more ingenuity is required from a user. Either additional arguments have to be passed to the tried methods or entirely different methods come into play. This is where our SMT integration as automatic prover behind *sledgehammer* (Figure 1.2) may show its full potential, as users tend to query existing proof automation before attempting to find the proof themselves. Table 2.2 gives corresponding figures. Finding proofs for "nontrivial" goals turns out to be harder, as expected, but SMT solvers still succeed in more than a third of all cases, again mostly due to Z3, which alone contributes 9.7% unique proofs among the SMT solvers. Moreover, SMT solvers find more proofs than ATPs on average. And again, the combination of ATPs and SMT solvers gives more results than one class of provers alone—SMT solvers add 8.9% proved goals to those found by ATPs.

We distill the following key findings from our evaluation:

- SMT solvers prove more than 50% of all goals, although success rates for individual theories vary between 23% and 66%.
- The combination of all SMT solvers prove more than a third of all "nontrivial" goals.
- SMT solvers find 8.8% unique proofs, i.e., proofs for goals on which all ATPs fail. *Hence, SMT solvers increase proof automation.*
- Z3 alone can prove about as many goals as all ATPs together, and this also holds for the "nontrivial" goals.
- CVC3 and Yices add only little to what Z3 can prove. *Z3 is already sufficient.*

Thus, we see our expectations confirmed. The integration of SMT solvers with Isabelle/HOL improves upon existing proof automation and complements what is already provided by ATPs. Moreover, we deduce from this evaluation that our work has resulted in a mature tool that is applicable to a wide range of problems in Isabelle/HOL.

2.5.3. Benefits from Decision Procedures and Extra-Logical Information

In contrast with ATPs, SMT solvers can throw in their combination of different decision procedures to increase their success rates. Naturally, we are interested in how much we gain from these extra features, or how much we would lose if SMT solvers were only able to solve first-order logic with equality like ATPs. We investigate here the effects of integer/real arithmetic and fixed-size bitvector decision procedures, of direct support for datatypes and records in Z3 and of extra-logical information. We carried out these evaluations on the aforementioned nine theories (Section 2.5.1). Since they lack bitvector problems, we hand-selected a separate set of problems for evaluating our bitvector support.

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All
metis	80%	24%	89%	77%	80%	92%	60%	29%	100%	75.9%

Table 2.3. Success rates of proof reconstruction with *metis* of proofs found by Z3

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All
CVC3	0%	+2%	−1%	+3%	+2%	+1%	−9%	+12%	+4%	+.8%
Yices	0%	+3%	+2%	+4%	+1%	0%	−6%	+11%	+1%	+1.5%
Z3	−1%	−3%	+2%	−2%	+1%	0%	−13%	+8%	+2%	−.8%
SMT	−1%	+1%	+3%	+2%	+1%	0%	−12%	+12%	+3%	+.9%

Table 2.4. Absolute success rate gains for SMT solver runs with arithmetic reasoning over runs without arithmetic reasoning on all goals

Arithmetic over integers and reals In case of the nine chosen Isabelle/HOL theories only arithmetic decision procedures are required over first-order logic with equality. Hence, to what extent do these decision procedures contribute to the overall result? As a first approximation, Table 2.3 shows how many goals, for which Z3 found a proof, could be reconstructed by the arithmetic-agnostic *metis* method within 30 seconds. Taking into account that *metis* follows a different strategy to find a proof than Z3 and typically fails when given too many facts, the overall number of 75.9% successfully reconstructed proofs indicates that arithmetic plays a minor role. Only the theories *FFT*, *QE* and *S2S* suffer considerably from the lack of arithmetic support by *metis*. Table 2.4 shows that disabling usage of arithmetic decision procedures, by treating arithmetic types and constants as uninterpreted (Section 2.2.6), but at the same time supplying appropriate facts (thereby changing the problem), has mixed effect on SMT solvers. Especially those Isabelle/HOL theories, where *metis* fails miserably, give an inconclusive picture. Arithmetic support gives clear benefits for the arithmetic-intense *S2S* theory, degrades SMT performance on *QE* by about the same amount (due to the dominance of *non*linear arithmetic that slows down the SMT solver), and does not change much on the results obtained for *FFT* (which, by the way, is also challenging for ATPs). Hence, arithmetic decision procedures partly contribute to the success of SMT solvers, but not to an overwhelming extent.

Looking at the number of unique proofs, we found that SMT solvers align more with each other when arithmetic support is disabled. Especially Z3 finds only 2.7% unique proofs in contrast to 5.8% with arithmetic (Table 2.1). This might indicate the effects of certain implementation tricks in the solvers' decision procedures. Yet, we also found that the combination of all SMT solvers still contribute 7.3% unique proofs over the ATPs which mirrors the inherent differences in solving strategies of these two classes of provers.

We also measured the effects of disabling our encoding for integer division and modulo with Z3 (Section 2.3), but found little differences (Table 2.5). In fact, this encoding only plays a role when computations involving these two operations on numerals

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All
Z3	+2%	−3%	−3%	+1%	−2%	0%	−1%	+5%	+1%	−.7%

Table 2.5. Absolute success rate gains for Z3 runs with support for division and modulo over runs without on all goals

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All
Z3	0%	0%	−1%	−1%	−1%	−2%	0%	0%	−3%	−.9%

Table 2.6. Absolute success rate gains for SMT solver runs without datatype support over runs with such datatype support on all goals

are performed. Since most applications of division and modulo in Isabelle/HOL are symbolic, treating them as uninterpreted and adding relevant facts should amount to the same, if not better, results. Hence, we currently see little use of direct support of these two operations, but future applications might benefit from it.

Datatypes and records Our encoding of datatypes, records and type definitions into features provided by Z3 (Section 2.3) is only sparsely tested. Nevertheless, we evaluated what can currently be gained from this encoding, yet our measurement results (Table 2.6) should be taken with a grain of salt. Clearly, no theory benefits, but we also see no dramatic losses. Hence, building on direct Z3 support for datatypes and records in its current state is both not harmful and not essential for typical applications in Isabelle/HOL.

Extra-logical information Enriching problems with extra-logical information, in our case with triggers inferred by a simple algorithm (Section 2.4) and with weights deduced from fact relevance [23], does not improve success rates (Table 2.7). Except for Yices, which does not support extra-logical annotations, results even deteriorate slightly. This indicates that our annotations are unsuitable or too limited or that the quantifier heuristics of the tested SMT solvers are already strong enough for our problems. This is in contrast with SMT folklore that (well-chosen) triggers almost inevitably lead to performance jumps of SMT solvers [30, 122].

Fixed-size bitvectors We tested the bitvector support on a collection of examples of the Isabelle/HOL Word library [56]. Out of 43 fixed-size bitvector goals, the SMT solvers can prove 28 which corresponds to a success rate of 65%. We admit that these figures allow no conclusion for the general case, because all goals, except for one, are "trivial" (Section 2.5.2), and those examples are meant to only showcase some functions of the Word library instead of posing a challenge to proof methods.

Gerwin Klein provided us with eight complicated theorems from the seL4 verification project [95] that are given below. We use the following notations. The symbols $\ll$ and $\gg$ denote shifting to the left and shifting to the right by a natural number. The term ucast x stands for extending the bitvector x by padding zeros to its front. The term

	Arrow	*FFT*	*FTA*	*Hoare*	*Jinja*	*NS*	*QE*	*S2S*	*SN*	All
CVC3	0%	−2%	−3%	−2%	0%	+3%	−2%	−1%	+1%	−1.4%
Yices	0%	0%	0%	0%	0%	0%	0%	0%	0%	.0%
Z3	−1%	−1%	−1%	−2%	−1%	0%	−8%	+5%	−1%	−1.5%
SMT	−1%	−3%	−1%	+1%	−1%	−2%	−6%	+1%	0%	−1.3%

Table 2.7. Absolute success rate gains for SMT solver runs with triggers and weights over runs without on all goals

mask n is defined as $(1 \ll n) - 1$ for natural numbers n. Selecting bit n from a bitvector x and interpreting it as a Boolean value is denoted by x_n. Finally, the symbols &, | and ! denote bitwise conjunction, disjunction and negation. All other symbols have the obvious semantics. We fix that u is a bitvector of size 12, that v and w are bitvectors of size 32, and that x and y are bitvectors that have arbitrary but equal size.

$$u \neq 0 \longrightarrow (1 \ll 20) - 1 < (\mathsf{ucast}\ u \ll 20)^{32\ \mathsf{word}} \tag{2.1}$$

$$(\mathsf{if}\ w = 0\ \mathsf{then}\ v \leq 0\ \mathsf{else}\ v \leq w - 1) \longrightarrow v \neq -1 \tag{2.2}$$

$$n < 32 \longrightarrow 1^{32\ \mathsf{word}} \leq (1 \ll n) \tag{2.3}$$

$$v\ \&\ \mathsf{mask}\ 14 = 0 \longrightarrow v + (w \gg 20 \ll 2)\ \&\ (!\ (\mathsf{mask}\ 14)) = v \tag{2.4}$$

$$v\ \&\ \mathsf{mask}\ n = 0 \wedge (\forall n'.\ n \leq n' \wedge n' < 32 \longrightarrow \neg w_{n'}) \longrightarrow v + w\ \&\ (!\ (\mathsf{mask}\ n)) = v \tag{2.5}$$

$$0 < y\ \wedge\ y \leq x \longrightarrow (x - y)\ \mathsf{div}\ y = (x\ \mathsf{div}\ y) - 1 \tag{2.6}$$

$$(x\ \&\ y) + (x \mid y) = x + y \tag{2.7}$$

$$x\ \&\ y = 0 \longrightarrow x + y = x \mid y \tag{2.8}$$

Since SMT solvers can only tackle bitvector problems with fixed bitvector sizes, we tested several sizes up to to 400 for the last three theorems.

Our results are as follows. The three SMT solvers agree with their results on all problems except for the last one where CVC3 takes longer than 30 seconds for bitvector sizes that are greater than 20. Theorems (2.1), (2.2) and (2.4) can be solved instantaneously by our SMT integration. Theorem (2.3) fails because the SMT solvers provide bit shifting only as a binary function of two bitvectors whereas in Isabelle the second argument is a natural number. Our translation maps natural numerals that are given as second argument to corresponding bitvector numbers, but this approach must necessarily fail for arbitrary terms. Theorem (2.5) also fails. Here the problem is related to selecting individual bits from a bitvector for which the SMT solvers provide no corresponding function, and hence our translation maps bit selection to an uninterpreted function without giving additional facts to the SMT solver. Theorem (2.6) is challenging for SMT solvers when the size of x and y exceeds a certain value. We found that Z3 takes less than 5 seconds for size 11, but already 15 seconds for size 12. With size 13 it takes longer than a minute to prove this theorem. The runtimes of the other two SMT solvers are even worse. Theorems (2.7) and (2.8) can be proved easily by both Z3 and Yices, and even with bitvector sizes of 400 they take less than three seconds.

It is hard to draw a general conclusion from testing a few hand-picked problems.

We note, though, that our SMT integration is applicable to different fixed-size bitvector problems unless they use functions uninterpreted by SMT solvers. Theorems referring to functions such as bit selection or casting of bitvectors to natural numbers and back fall outside the range of what is provable with our SMT integration.

Key findings From evaluating decision procedures and extra-logical control provided by SMT solvers we distill the following key findings:

- Arithmetic support of SMT solvers is in general not essential for typical goals in Isabelle/HOL, but certain goals can benefit from it.
- Direct support for datatypes, records and type definition by Z3 gives no clear benefits.
- Proving problems of fixed-size bitvectors works well unless functions that are unsupported by SMT solvers are used.
- Extra-logical information that is automatically inferred by our current simple algorithm slightly deteriorates the success rates of SMT solvers.

We found that SMT solvers are a good complement to ATPs, even if they would not provide specific decision procedures or extra-logical control information.

2.6. Related Work

The general idea behind our work, i.e., applying an automatic theorem prover (with a first-order logic) within an interactive theorem prover (of, for example, higher-order logic), has seen several previous attempts and successes. Consequently, many of our translations described earlier (Section 2.2) appear in similar form in earlier work, although some translations are also taken from entirely different fields of research. We detail related work in separate groups.

Integration of automatic provers Applying automatic first-order provers as backends in interactive theorem provers has a long tradition [1,22,35,85,115,148,157]. Along these lines, the integration of ATPs with Isabelle/HOL, *sledgehammer*, by Paulson et al. [118,138] is most notable for its success. It originally connected a class of automatic first-order resolution provers with Isabelle/HOL, building on previous work by Hurd to integrate the first-order prover Metis with HOL4 [86], but has since evolved into a more general framework for external provers [23]. As part of our evaluation (Section 2.5) we made use of this framework and found SMT solvers a suitable complement to ATPs.

Only vaguely related to our work and far less powerful than SMT solvers, there are integrations of SAT solvers with interactive theorem provers, e.g., with Isabelle/HOL and HOL4 [159,163], with ACL2 [142] and recently with Coq [5]. Similar work has been done for integrating solvers for quantified Boolean formulas (QBF) in HOL4 [98,162]

and HOL Light [100]. To the best of our knowledge, there are no extensive studies demonstrating the usefulness and applicability of these integrations in proof developments of interactive theorem provers. It is hence unclear whether SAT or QBF solvers can in general give further proof automation over our work.

Directly in the line of our work are several integrations of SMT solvers with interactive theorem provers. Since many years already, the SMT solver Yices and its predecessor ICS have been tightly connected with PVS [69, 145], and this setup has doubtlessly helped in formalizations. Yet, we know of no extensive study giving evidence of its general usefulness beyond typical applications of SMT solvers such as program verification. An integration of UCLID [101] with the interactive theorem prover ACL2 [110] has demonstrated that SMT solvers can be valuable for discharging certain classes of problems in ACL2. There are early attempts to integrate haRVey as well as ICS and CVC Lite with Isabelle/HOL [18, 71], but they are less more than an ad-hoc experimental study without extensive tests. Furthermore, they do not exploit the full power of SMT solvers as the authors concentrate on problems directly representable in the target solver's logic. Later work [87] extends the integration of haRVey with Isabelle/HOL from supporting only quantifier-free first-order logic with equality to quantified formulas. In much the same way, the connection between CVC Lite, later succeeded by its successor CVC3, and HOL Light [73, 116] concentrates on essentially first-order propositions, failing on all higher-order propositions, but supports arithmetic. There exists also an integration of SMT solvers including Simplify, CVC Lite and haRVey in Coq [6] based on Why [67] to automate the translation from polymorphic first-order logic with inductive datatypes to MSFOL or untyped FOL, but abstracting higher-order features. Later work [48] also integrates the Ergo theorem prover that has direct support for polymorphic first-order logic with equality and linear arithmetic. Erkök and Matthews describe a similar interface between Yices and Isabelle/HOL [66] that in addition to datatypes also supports records and higher-order features all of which are directly accepted as input of Yices, but they do not underpin their description with concrete figures drawn from applications or evaluations. Parallel to our work and in much the same way, Weber implemented a generic interface to SMT solvers for HOL4 [161]. In contrast with our work, he concentrates only on essentially first-order propositions and abstracts away all terms that cannot directly be represented in MSFOL, partly except for Yices that has built-in support for λ-abstractions. Recently, work on integrating the SMT solver veriT in Coq [94] has begun. In the light of related work, we consider our integration of SMT solvers with Isabelle/HOL superior in that it is not focused on a single SMT solver, targets most if not all supported theories of current SMT solvers and, instead of using coarse-grained abstractions, aims to encode any HOL term into MSFOL which can lead to higher success rates.

Type variables Our monomorphization approach is rooted in research on compilers for functional languages [155]. Since monomorphization leads to exponential blow-up due to duplications, it has been regarded as unsuitable in the past, except for [66]. Instead, types and hence also type variables have been encoded as terms [51, 109, 118],

thereby increasing the problem size only by a small constant factor. These techniques, however, come at a price: They tend to be harder to implement, they require a careful treatment of the order-sorted type system of Isabelle/HOL [164] by means of additional predicates along the lines of *sledgehammer* [120], which might cause extra load on the target SMT solver, and they complicate proof reconstruction. Moreover, representing types as terms results in a nearly untyped encoding, potentially destroying all benefits SMT solvers may draw from sort information. Having built-in support for type variables in SMT solvers, as suggested by Conchon et al. [26], might be a solution.

Lambda-abstraction Translating λ-abstractions into first-order representations were first studied in the context of compilers for functional languages. Lambda-lifting [92], applied by Kanig [93] and us, is closely related to defunctionalization [143]. Replacing abstractions by a small set of combinators, as done by Metis [86] and *sledgehammer* [118], has even older roots [8]. The combinator approach may have its value for untyped encodings, but we found combinators to be unsuitable for SMT solvers, possibly because they require too many quantifier instantiations that easily spoil the solvers' search space.

Partial application Introducing explicit application symbols to handle partial application is a well-known practice, going back to at least Reynolds [143]. For instance, *sledgehammer* applies this technique [118], and so does Kanig [93]. Interestingly, explicit application symbols are also introduced within SMT solvers to simplify and speed-up congruence closure algorithms [72].

Separation of formulas and terms Our approach to separate formulas from terms is largely inspired by Jackson et al. [89]. They introduce an additional Boolean type and proxy constants for True and False, and they also describe the wrapping of terms in equations (a technique also applied in other program verifiers such as ESC/Java [61], there called quasi-relations) as well as the lifting of formulas inside terms via if-then-else-expressions. In addition, they also suggest to provide proxies for logical connectives and predicates, which we consider unnecessary. Instead of equating terms with the proxy truth TT, Kanig [93] wraps terms into a special uninterpreted predicate, which should amount to similar results.

Evaluation Not surprisingly, evaluating the integration of automated provers with interactive theorem provers attracted much fewer attention than the actual integration, because evaluations tend to be hard and less rewarding work. In most cases, only few case studies were performed instead of extensive measurements on a large body of examples, or nothing about the practical usefulness of these integrations has been reported. Notable difference is our work with the *sledgehammer* tool [23, 32], but also the work by Urban [157] concentrating on ATPs. We thus claim that our evaluation (Section 2.5) is the first to reliably demonstrate the general usefulness of SMT solvers for interactive theorem provers.

Chapter 3.

Reconstructing Z3 Proofs

Contents

3.1. Introduction

Oracles are typically mistrusted in Isabelle/HOL and this applies equally to our integration of SMT solvers. There are two reasons. First, our translation from HOL to MSFOL (Chapter 2), although conceptually sound, contained several bugs in earlier versions, and we are uncertain whether our extensive evaluation (Section 2.5) left some bugs undiscovered. Second, it is well known that virtually all SMT solvers have bugs [39]. In fact, while developing and testing the work described in this chapter, we have found several soundness bugs in Z3 that have been resolved in more recent versions after we informed the developers. Thus, by blindly trusting an SMT solver and our translation we risk to consider invalid conjectures as valid ones.

In principal, there are at least two approaches to overcome this issue and yield high confidence. The first one is to formally verify the SMT solver, which is an immense amount of work [94]. Already verifying a modern SAT solver, one of the core components of an SMT solver, is highly involved [113]. The second approach is checking the correctness of each output, which is the approach that we will follow in this chapter for the SMT solver Z3. We concentrate on reconstructing proofs step-by-step for MSFOL with the theories of equality and linear arithmetic, i.e., a combination of theories that already gave good improvement over existing proof automation in Isabelle/HOL (Section 2.5.2), and leave the theories of bitvectors and datatypes as future work.

We begin by describing the proof format of Z3 (Section 3.2). Then, we detail efficient reconstruction of Z3 proofs in Isabelle/HOL (Section 3.3), focusing on specific optimizations to obtain high performance, before giving evidence of the approach's efficiency with a detailed evaluation (Section 3.4). Much of the optimization work has

been done jointly with Tjark Weber [33], although he concentrated on HOL4 instead of Isabelle/HOL where he found partly different solutions. We conclude by reviewing related work (Section 3.5).

3.2. Z3 Proof Format

The language underlying Z3's proofs is many-sorted first-order logic (MSFOL, Section 1.4.1) with one notable difference: Conjunctions and disjunctions are polyadic—a straightforward extension of the standard binary case. For example, the polyadic conjunction $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ should be understood as exactly one application of $\wedge$ to three formulas φ_1, φ_2 and φ_3 instead of a nested application of binary conjunctions such as $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$. For this entire section, we will consider conjunctions and disjunctions polyadic.

Z3's proof system is a natural deduction calculus. It uses 38 axiom schemata and inference rules (collectively referred to as proof rules) ranging from simple propositional to complex theory reasoning. We present Z3's proof system with *sequents* $\Gamma \vdash \varphi$ where Γ is a set of MSFOL formulas (*hypotheses*) and φ is an MSFOL formula (*proposition*). When the set of hypotheses is empty, we simply write $\vdash \varphi$. Inferences in natural deduction are tree-shaped where each leaf is an instance of an axiom schemata of the calculus and each inner node is a deduction from a set of sequents (*premises*) to a new sequent (*conclusion*) via an inference rule of the calculus. We use the notation $\langle S_i \rangle_{i \in I}$ to denote a set of premises or preconditions S_i, indexed by a finite set I.

The textual representation of proofs as emitted by Z3 is slightly different. Instead of the typical tree structure, Z3's proofs exploit sharing and are thus directed acyclic graphs. Each node represents a single deduction step, labeled by the name of a proof rule and the proposition to conclude. The edges of a proof graph connect premises with conclusions. Hypotheses of sequents are not given explicitly. Since every Z3 proof deduces unsatisfiability of a set of formulas, the root node of the proof concludes $\bot$. It thus corresponds to the sequent $\Pi' \vdash \bot$ where Π' is a subset of the assertions Π initially given to Z3 to be shown unsatisfiable. The set Π' is thus called an *unsatisfiable core* of Π.

The proof rules of Z3 fall into three categories. First, several proof rules are simple enough to be presented without further ado (Figure 3.1). Second, some proof rules deserve more explanation, and we will spend most of the remainder of this section on them. Since existing documentation of Z3's proof rules is sparse, our description of some of them is vague, depending on how much information was revealed by the developers of Z3 and our tested proof examples. Third, a small fraction of proof rules reflect deductions performed only rarely or when particular Z3 options are enabled. Since we never apply these options in interactions with Z3 and also rarely exercised these proof rules in any proof, we refrain from describing them in detail and give only short explanations at the end of this section. Our presentation uses slightly different, shorter names for the proof rules than Z3 (see Appendix A for the correspondence).

$$\frac{}{\vdash \top}\ \text{true} \qquad \frac{\varphi \in \Pi}{\{\varphi\} \vdash \varphi}\ \text{asserted} \qquad \frac{\Gamma_1 \vdash \varphi_1 \qquad \Gamma_2 \vdash \varphi_1 \rightsquigarrow \varphi_2}{\Gamma_1 \cup \Gamma_2 \vdash \varphi_2}\ \text{mp}_{\rightsquigarrow}$$

$$\frac{}{\{\varphi\} \vdash \varphi}\ \text{hypothesis} \qquad \frac{\Gamma \cup \{l_1, \ldots, l_n\} \vdash \bot}{\Gamma \setminus \{l_1, \ldots, l_n\} \vdash \neg l_1 \vee \ldots \vee \neg l_n}\ \text{lemma}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \longleftrightarrow \top}\ \text{iff}_{\top} \qquad \frac{\Gamma \vdash \neg\varphi}{\Gamma \vdash \varphi \longleftrightarrow \bot}\ \text{iff}_{\bot} \qquad \frac{\Gamma \vdash \varphi_1 \longleftrightarrow \varphi_2}{\Gamma \vdash \varphi_1 \sim \varphi_2}\ \text{iff}_{\sim} \qquad \frac{}{\vdash t \simeq t}\ \text{refl}_{\simeq}$$

$$\frac{\Gamma \vdash l_1 \wedge \ldots \wedge l_n}{\Gamma \vdash l_i}\ \text{elim}_{\wedge} \qquad \frac{\Gamma \vdash \neg\left(l_1 \vee \ldots \vee l_n\right)}{\Gamma \vdash \neg l_i}\ \text{elim}_{\neg\vee} \qquad \frac{\Gamma \vdash t_1 \simeq t_2}{\Gamma \vdash t_2 \simeq t_1}\ \text{symm}_{\simeq}$$

$$\frac{\Gamma_1 \vdash t_1 \simeq t_2 \qquad \Gamma_2 \vdash t_2 \simeq t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \simeq t_3}\ \text{trans}_{\simeq} \qquad \frac{\Gamma \vdash (t_1 \approx t_2) \simeq (t_1' \approx t_2')}{\Gamma \vdash (t_2 \approx t_1) \simeq (t_2' \approx t_1')}\ \text{comm}_{\simeq,\approx}$$

Figure 3.1. Simple Z3 proof rules

Unit resolution Unit resolution is the single inference performed by Z3's propositional reasoning engine following the DPLL design (Section 1.4.2). A unit resolution step expresses strengthening of a disjunction by removing literals which have been disproved. The unit-resolution rule reflects this inference step in Z3's proofs. We use I as a shorthand for the set $\{1, \ldots, n\}$ and $J \subseteq I$ for a nonempty subset of I.

$$\frac{\Gamma \vdash \bigvee_{i \in I} l_i \qquad \langle \Gamma_j \vdash \neg l_j \rangle_{j \in J}}{\Gamma \cup \bigcup_{j \in J} \Gamma_j \vdash \bigvee_{i \in I \setminus J} l_i}\ \text{unit-resolution}$$

Tseitin-like axioms Tseitin [156] proposed a conversion of propositional formulas into equisatisfiable formulas in conjunctive normal form (CNF) with polynomial complexity, and Z3 uses a variant thereof. To understand the approach of Z3, we first review Tseitin's original transformation on the example $\varphi \longrightarrow \psi$ where φ and ψ are arbitrary formulas that we treat atomically here. The basic idea of the transformation is to introduce auxiliary propositional variables and to define them to be equivalent to formulas not already in CNF. In our example, we replace the implication by the fresh propositional variable p_1 and conjoin the definition expressing that p_1 is equivalent to the implication. Thus, we obtain the following formula equisatisfiable to $\varphi \longrightarrow \psi$:

$$p_1 \wedge (p_1 \longleftrightarrow (\varphi \longrightarrow \psi))$$

With φ and ψ we can proceed similarly using fresh propositional variables p_2 and p_3 resulting in the following equisatisfiable formula:

$$p_1 \wedge (p_1 \longleftrightarrow (p_2 \longrightarrow p_3)) \wedge (p_2 \longleftrightarrow \varphi) \wedge (p_3 \longleftrightarrow \psi) \tag{3.1}$$

To the second conjunct, we can apply equivalence-preserving transformations to obtain a formula in CNF:

$$(\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_2) \wedge (p_1 \vee \neg p_3) \tag{3.2}$$

In the same way, we can find equivalent formulas for the last two conjuncts of (3.1) and finally obtain a formula ϕ in CNF that is equisatisfiable to $\varphi \longrightarrow \psi$ and whose size is only polynomially greater than that of the initial implication.

Z3 optimizes this transformation in two ways. First, Z3 applies it in a lazy manner as in Simplify [61] during the satisfiability search. That is, instead of transforming the entire input at once, it does so gradually. For note that if p_1 together with a set of clauses C is already unsatisfiable, so is ϕ and C. Otherwise, i.e., if p_1 conjoined to C is satisfiable, Z3 will additionally conjoin the clauses (3.2) and continue the search, followed by unfolding and conjoining further definitions of auxiliary propositional variables on satisfiability. Second, Z3 applies Tseitin's transformation without introducing fresh propositional variables—the subformulas themselves are used instead [58]. Recalling our above example, Z3 uses $\varphi \longrightarrow \psi$, φ and ψ instead of p_1, p_2 and p_3 everywhere. For example, the formula (3.2) is represented in Z3 as follows where boxed formulas are treated atomically by Z3:

$$\left(\neg\boxed{\varphi \longrightarrow \psi} \vee \neg\boxed{\varphi} \vee \boxed{\psi}\right) \wedge \left(\boxed{\varphi \longrightarrow \psi} \vee \boxed{\varphi}\right) \wedge \left(\boxed{\varphi \longrightarrow \psi} \vee \neg\boxed{\psi}\right)$$

Note that each of these clauses is a propositional tautology when ignoring the boxes. Z3 introduces such clauses with unboxed formulas as definitional axioms via the def-axiom rule into proofs. For example, these would be the def-axiom steps corresponding to the above clauses:

$$\frac{}{\vdash \neg(\varphi \longrightarrow \psi) \vee \neg\varphi \vee \psi} \qquad \frac{}{\vdash (\varphi \longrightarrow \psi) \vee \varphi} \qquad \frac{}{\vdash (\varphi \longrightarrow \psi) \vee \neg\psi}$$

Figure 3.2 lists the schemata that are documented for def-axiom. This list is not exhaustive for several reasons. Conjunction and disjunction are polyadic, i.e., for definitional axioms about binary conjunctions or disjunctions there exist also variants with more than two subformulas. For some of the listed schemata we have seen variants that are isomorphic under associativity and commutativity of disjunction. There are also definitional axioms for if-then-else-expressions on MSFOL terms where the branches are terms instead of formulas. Finally, we observed undocumented schemata for a ternary exclusive disjunction operator.

Local definitions Z3 can locally abbreviate terms and formulas by fresh names. The rule intro-def binds a fresh name n to a term t or formula φ in such a way that the proposition of the rule is in CNF, whereas apply-def re-establishes the defining equation by purely logical rewriting.

The schemata of intro-def depend on the structure of t or φ. If t is syntactically equivalent to $(\text{if } \psi \text{ then } t_1 \text{ else } t_2)$ or φ is syntactically equivalent to $(\text{if } \psi \text{ then } \phi_1 \text{ else } \phi_2)$, then these are the corresponding schemata of intro-def:

$$\frac{}{\vdash \neg\neg\varphi \vee \neg\varphi} \qquad \frac{}{\vdash \neg\varphi \vee \varphi}$$

$$\frac{}{\vdash \neg(\varphi \wedge \psi) \vee \varphi} \qquad \frac{}{\vdash \neg(\varphi \wedge \psi) \vee \psi} \qquad \frac{}{\vdash (\varphi \wedge \psi) \vee \neg\varphi \vee \neg\psi}$$

$$\frac{}{\vdash \neg(\varphi \vee \psi) \vee \varphi \vee \psi} \qquad \frac{}{\vdash (\varphi \vee \psi) \vee \neg\varphi} \qquad \frac{}{\vdash (\varphi \vee \psi) \vee \neg\psi}$$

$$\frac{}{\vdash \neg(\varphi \longleftrightarrow \psi) \vee \neg\varphi \vee \psi} \qquad \frac{}{\vdash \neg(\varphi \longleftrightarrow \psi) \vee \varphi \vee \neg\psi}$$

$$\frac{}{\vdash (\varphi \longleftrightarrow \psi) \vee \neg\varphi \vee \neg\psi} \qquad \frac{}{\vdash (\varphi \longleftrightarrow \psi) \vee \varphi \vee \psi}$$

$$\frac{}{\vdash \neg(\text{if } \varphi \text{ then } \psi \text{ else } \phi) \vee \neg\varphi \vee \psi} \qquad \frac{}{\vdash \neg(\text{if } \varphi \text{ then } \psi \text{ else } \phi) \vee \varphi \vee \phi}$$

$$\frac{}{\vdash (\text{if } \varphi \text{ then } \psi \text{ else } \phi) \vee \neg\varphi \vee \neg\psi} \qquad \frac{}{\vdash (\text{if } \varphi \text{ then } \psi \text{ else } \phi) \vee \varphi \vee \neg\phi}$$

Figure 3.2. Schemata for the def-axiom rule from the documentation of Z3

$$\frac{}{\vdash (\neg\psi \vee n = t_1) \wedge (\psi \vee n = t_2)} \qquad \frac{}{\vdash (\neg\psi \vee (n \longleftrightarrow \varphi_1)) \wedge (\psi \vee (n \longleftrightarrow \varphi_2))}$$

Otherwise the schemata for intro-def are as follows:

$$\frac{}{\vdash n = t} \qquad \frac{}{\vdash (\neg n \vee \varphi) \wedge (n \vee \neg\varphi)}$$

For every schema of intro-def there is a corresponding schema of apply-def:

$$\frac{\Gamma \vdash (\neg\psi \vee n = t_1) \wedge (\psi \vee n = t_2)}{\Gamma \vdash (\text{if } \psi \text{ then } t_1 \text{ else } t_2) = n} \qquad \frac{\Gamma \vdash (\neg\psi \vee (n \longleftrightarrow \varphi_1)) \wedge (\psi \vee (n \longleftrightarrow \varphi_2))}{\Gamma \vdash (\text{if } \psi \text{ then } \varphi_1 \text{ else } \varphi_2) \longleftrightarrow n}$$

$$\frac{\Gamma \vdash n = t}{\Gamma \vdash t = n} \qquad \frac{\Gamma \vdash (\neg n \vee \varphi) \wedge (n \vee \neg\varphi)}{\Gamma \vdash \varphi \longleftrightarrow n}$$

The intro-def schemata are designed to integrate well with the propositional reasoning of Z3 which is the reason for the conclusion being in CNF, whereas apply-def establishes a form suitable for equality reasoning. The symmetric nature of apply-def is an optimization to avoid a further application of the $\text{symm}_{\simeq}$ rule (Figure 3.1).

Congruence Equality ($=$), equivalence ($\longleftrightarrow$) and equisatisfiability ($\sim$) are congruence relations. We use the placeholder $\simeq$ to stand for one of these relations. Let $I = \{1, \ldots, n\}$, let $J \subseteq I$ be a nonempty subset of I, and let $\equiv$ denote syntactic equality of terms. The congruence rule for a function symbol f is as follows:

$$\frac{\left\langle \Gamma_j \vdash t_j \simeq t'_j \right\rangle_{j \in J} \qquad \left\langle t_i \equiv t'_i \right\rangle_{i \in I \setminus J}}{\bigcup_{j \in J} \Gamma_j \vdash f(t_1, \ldots, t_n) \simeq f\left(t'_1, \ldots, t'_n\right)} \text{ cong}_{\simeq}$$

Congruence for predicate symbols is similar. The $\mathsf{cong}_{\sim}$ rule deviates from the standard congruence rule (see, e.g., [83, chapter 4]) in that reflexive premises are omitted to shorten proofs as indicated by the precondition in the above schematic rule.

Quantifiers *Z3*'s quantifier proof rules are tailored to its propositional and equality reasoning and thus deviate from the standard introduction and elimination rules of natural deduction. Before presenting these rules, we fix some notations.

We use Q to denote a quantifier, i.e., either $\forall$ or $\exists$, and $\overline{x}$ and $\overline{y}$ for lists of (bound) variables $x_1, \ldots x_m$ and $y_1, \ldots y_n$ as well as $\overline{t}$ for a list of terms $t_1, \ldots t_k$. Moreover, the notations $\varphi[\overline{x}]$, $\varphi[\overline{y}, \overline{x}]$ and $\varphi[\overline{y}, \overline{t}]$ stand for formulas possibly containing variables $\overline{x}$ and $\overline{y}$ and terms $\overline{t}$, the formula $\varphi[\overline{x} \mapsto \overline{t}]$ is short for replacing in φ each x_i of $\overline{x}$ by the corresponding t_i from $\overline{t}$, and $\overline{f}(\overline{y})$ abbreviates a list of fresh function symbols each applied to variables $\overline{y}$.

Given two equisatisfiable formulas $\varphi_1[\overline{x}]$ and $\varphi_2[\overline{x}]$ depending on variables $\overline{x}$ that are not occurring in a set of formulas Γ, a Z3 proof can introduce a quantifier Q as follows:

$$\frac{\Gamma \vdash \varphi_1[\overline{x}] \sim \varphi_2[\overline{x}]}{\Gamma \vdash (Q\overline{x}.\, \varphi_1[\overline{x}]) \sim (Q\overline{x}.\, \varphi_2[\overline{x}])}\ \mathsf{intro}_Q$$

Instantiating a universally quantified formula $(\forall \overline{x}.\, \varphi[\overline{x}])$ with terms $\overline{t}$ is expressed by the following axiom schema whose proposition is an implication transformed to CNF:

$$\frac{}{\vdash \neg\, (\forall \overline{x}.\, \varphi[\overline{x}]) \vee \varphi[\overline{x} \mapsto \overline{t}]}\ \mathsf{inst}_\forall$$

Existential quantifiers are eliminated by Skolem constants as witnesses using one of the following two axiom schemata:

$$\frac{}{\vdash (\exists \overline{x}.\, \varphi[\overline{y}, \overline{x}]) \sim \varphi[\overline{y}, \overline{f}(\overline{y})]}\ \mathsf{sk}_\exists \qquad \frac{}{\vdash (\neg\, (\forall \overline{x}.\, \varphi[\overline{y}, \overline{x}])) \sim (\neg \varphi[\overline{y}, \overline{f}(\overline{y})])}\ \mathsf{sk}_\forall$$

Finally, if a formula $\varphi[\overline{x}]$ depends only on the variables $\overline{x}$, Z3 can eliminate redundantly bound variables $\overline{y}$ that are not occurring in $\varphi[\overline{x}]$ with the following axiomatic proof rule:

$$\frac{}{\vdash (Q\overline{x}\,\overline{y}.\, \varphi[\overline{x}]) \longleftrightarrow (Q\overline{x}.\, \varphi[\overline{x}])}\ \mathsf{elim}_Q$$

Negation normal form Every MSFOL formula can be transformed into an equisatisfiable (and even equivalent) formula in negation normal form (NNF) in linear time. For this transformation, Z3 provides two rules, nnf-pos and nnf-neg, depending on the polarity of the subformula transformed. Both rules express inferences that perform one step of the NNF transformation, i.e., either a step that changes the "top-level" connective or quantifiers or a step that performs the NNF transformation under some quantifiers. There appear to be schemata of nnf-pos and nnf-neg for all quantifiers and logical connectives, but we have no exhaustive list of these schemata and thus present only typical ones including those found in the documentation of Z3.

Assume that prior proof steps have performed NNF transformations or other equisatisfiability-preserving steps to formulas φ_1 and φ_2 as well as to the corresponding negated formulas. A schema for nnf-pos that subsequently changes the "top-level" equivalence in the formula $\varphi_1 \longleftrightarrow \varphi_2$ is as follows:

$$\frac{\Gamma_1 \vdash \neg\varphi_1 \sim \varphi_1' \qquad \Gamma_2 \vdash \neg\varphi_2 \sim \varphi_2' \qquad \Gamma_3 \vdash \varphi_1 \sim \varphi_1'' \qquad \Gamma_4 \vdash \varphi_2 \sim \varphi_2''}{\Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4 \vdash (\varphi_1 \longleftrightarrow \varphi_2) \sim \left((\varphi_1' \vee \varphi_2'') \wedge (\varphi_1'' \vee \varphi_2')\right)}$$

An nnf-pos schema that expresses an NNF transformation under some quantifiers is as follows where none of the variables in $\overline{x}$ occurs in Γ and $\overline{y}$ is a sublist of $\overline{x}$:

$$\frac{\Gamma \vdash \varphi_1[\overline{x}] \sim \varphi_2[\overline{y}]}{\Gamma \vdash (Q\overline{x}.\, \varphi_1[\overline{x}]) \sim (Q\overline{y}.\, \varphi_2[\overline{y}])}$$

Note the subtle difference between the intro_Q rule and this schema. Here, the premise may be the result of a step that eliminated some variables. Thus, the right-hand side of the conclusion possibly quantifies over less variables than the left-hand side in contrast to the intro_Q rule.

Each nnf-neg schema expresses one of the standard NNF transformation steps. Here are schemata for conjunction, disjunction and equivalence:

$$\frac{\Gamma_1 \vdash \neg\varphi_1 \sim \varphi_1' \qquad \cdots \qquad \Gamma_n \vdash \neg\varphi_n \sim \varphi_n'}{\Gamma_1 \cup \ldots \cup \Gamma_n \vdash \neg(\varphi_1 \wedge \ldots \wedge \varphi_n) \sim (\varphi_1' \vee \ldots \vee \varphi_n')}$$

$$\frac{\Gamma_1 \vdash \neg\varphi_1 \sim \varphi_1' \qquad \cdots \qquad \Gamma_n \vdash \neg\varphi_n \sim \varphi_n'}{\Gamma_1 \cup \ldots \cup \Gamma_n \vdash \neg(\varphi_1 \vee \ldots \vee \varphi_n) \sim (\varphi_1' \wedge \ldots \wedge \varphi_n')}$$

$$\frac{\Gamma_1 \vdash \neg\varphi_1 \sim \varphi_1' \qquad \Gamma_2 \vdash \neg\varphi_2 \sim \varphi_2' \qquad \Gamma_3 \vdash \varphi_1 \sim \varphi_1'' \qquad \Gamma_4 \vdash \varphi_2 \sim \varphi_2''}{\Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4 \vdash \neg(\varphi_1 \longleftrightarrow \varphi_2) \sim \left((\varphi_1' \vee \varphi_2') \wedge (\varphi_1'' \vee \varphi_2'')\right)}$$

The nnf-neg schemata for quantifiers are similar to those of nnf-pos except that here the "top-level" quantifiers get changed:

$$\frac{\Gamma \vdash \neg\varphi_1[\overline{x}] \sim \varphi_2[\overline{y}]}{\Gamma \vdash \neg(\forall\overline{x}.\, \varphi_1[\overline{x}]) \sim (\exists\overline{y}.\, \varphi_2[\overline{y}])} \qquad \frac{\Gamma \vdash \neg\varphi_1[\overline{x}] \sim \varphi_2[\overline{y}]}{\Gamma \vdash \neg(\exists\overline{x}.\, \varphi_1[\overline{x}]) \sim (\forall\overline{y}.\, \varphi_2[\overline{y}])}$$

Theory reasoning Explanations from Z3's theory solvers (Section 1.4) are captured by the th-lemma rule. Such inferences conclude disjunctions of theory literals from premises that are literals of the same theory. Each such step is either a lemma learned from the theory solver or a lemma expressing unsatisfiability of a set of literals within a particular theory. Each instance of the th-lemma rule carries a tag of the corresponding theory and, in case of linear arithmetic, provides an additional tag specifying the exact decision step behind the inference as well as coefficients for the participating arithmetic (in-)equalities. There is no comprehensive list of theory tags, yet we know that arith specifies the theory of arithmetic, and the tags farkas (for a refutation according to Farkas' lemma), triangle-eq (a split of an equality into two inequalities) and gcd-test (unsatisfiability of integer inequations due to divisibility arguments) are tags for linear arithmetic.

Rewriting Any preprocessing (Section 1.4), be it simplification or canonicalization, of the initially given assertions or parts thereof is reflected by a single proof rule called rewrite. Many different kinds of inferences can be declared as rewriting, e.g., simple propositional or much more complicated theory-specific reasoning (including quantifier elimination), but unlike the th-lemma rule they are not distinguished by any tags. A rewriting step takes the form of an axiom, and its conclusion is either an equivalence or an equation expressing exactly one simplification or canonicalization step. Beyond this schematic form, the variations of the rewrite rule are not a priori restricted.

Other proof rules There are further Z3 proof rules, which we discuss only briefly. The distributivity rule distributes disjunctions over conjunctions in case Z3 is configured to perform eager conversion of formulas to conjunctive normal form. The rules pull-quant and push-quant move quantifiers in a formula, especially for finding better triggers (Section 1.4.2). The der rule performs destructive equality resolution [74]. Z3 knows of five further rules to shorten proofs, i.e., each of them represents an inference that, when Z3 is instructed not to compress proofs, is equivalently composed of some of the other rules described so far.

3.3. Proof Reconstruction

Replaying Z3's proofs in Isabelle/HOL requires to map the underlying language to HOL. This is straightforward as there is a natural representation of MSFOL in HOL (Section 2.2), and we apply it to represent formulas in Z3 proofs as terms in Isabelle/HOL. Z3's extensions to MSFOL, i.e., polyadic conjunctions and disjunctions, are represented as trees of binary conjunctions and disjunctions, balanced for better efficiency.

Equisatisfiability is not readily formalized in Isabelle/HOL, yet such a formalization is unnecessary: We can soundly replace equisatisfiability by equivalence everywhere with the sk_Q rules being the only exception. To prove this claim we argue inductively over the structure of Z3 proofs and restrict ourselves to those proof rules where equisatisfiability occurs explicitly. It suffices to consider the following two cases. First, the rules $\mathsf{refl}_\simeq$ and $\mathsf{iff}_\sim$ introduce equisatisfiability, but only between two equivalent formulas, i.e., their conclusion is unnecessarily general. Second, the rules $\mathsf{symm}_\simeq$, $\mathsf{trans}_\simeq$, $\mathsf{comm}_{\simeq,\approx}$, $\mathsf{cong}_\simeq$, nnf-pos, nnf-neg and intro_Q deduce equisatisfiability of two formulas from equisatisfiability premises. From our induction hypothesis we derive that the conclusions of those rules can be specialized to equivalences. For note that their premises are themselves conclusions of certain proof rules in which equisatisfiability can soundly be replaced by equivalence. Hence, only the sk_Q rules need to be treated specially. We replay them in such a way that their conclusions are equivalences (Section 3.3.2). Whenever a fresh constant, such as a Skolem constant, occurs in a Z3 proof, we represent it in Isabelle/HOL by a free variable.

Z3 proofs are reconstructed in a bottom-up traversal of the proof graph, or more precisely in depth-first postorder starting from the root node. Proof nodes, which contain the information given by Z3, i.e., rule name, references to premises, and proposi-

tion, are represented as an algebraic data type. Each node is associated with a unique identifier that Z3 has already assigned to inferences and that is present in the Z3 proof. We store the nodes of the proof graph in a balanced tree with efficient lookup (in $O(\log n)$) using those node identifiers as keys. Replayed proof nodes are replaced by Isabelle/HOL theorems encapsulating the conclusion (both hypotheses and proposition). Hence, proof nodes are replayed at most once, even if referenced several times in the proof, and proof steps that do not contribute to the proof are never replayed. After every step we compare the inferred theorem's proposition with the proposition stored in the proof node to detect bugs in either our reconstruction methods or in the Z3 proof. Hypotheses in a Z3 proof can be introduced either by asserted or hypothesis. Local assumptions from the latter rule are discharged by the lemma rule later. After having replayed the root node, we check that only hypotheses from asserted rules, i.e., the propositions in the unsat core Π' (Section 3.2), remain.

Proof reconstruction for Z3 is sound by construction, because we rely on Isabelle/HOL's LCF-style kernel (Section 1.3.2), but proof reconstruction is likely incomplete due to the sparse documentation of some of Z3's proof rules (Section 3.2). Despite extensive tests, proof reconstruction may still fail for a small fraction of Z3 proofs. Such failures are not always caused by our incomplete replay implementation, but may also be provoked by soundness bugs in Z3, particularly in its proof generation, of which we found about ten and that have subsequently been resolved by Z3's developers.

We use four techniques to reconstruct individual inference steps, each for a different set of Z3 proof rules. Table 3.1 summarizes these techniques and partitions the proof rules correspondingly. They vary in implementation effort (highest for the second and fourth technique) and performance (decreases from first to fourth technique). Specifically, applying primitive inferences or metatheorems (Section 1.3.2) is the most efficient technique in an LCF-style system. Its drawback is limited applicability: The set of primitive inferences is fixed, and metatheorems can only be applied to terms of a fixed structure. For instance, deriving t from a conjunction $t \wedge u$ is within the realm of possibility, but a derivation of some t_i from an arbitrarily nested conjunction $t_1 \wedge \ldots \wedge t_n$ already requires a combination of primitive inferences and metatheorem instantiations. In contrast, automated proof procedures work well for complex problems and require little programming effort, but their performance is hard to control. By replacing them with methods that apply specific combinations of metatheorems, we achieved speedups of up to four orders of magnitude. Designing such methods tailored for Z3 proof reconstruction was only possible because about half of Z3's proof rules merely require propositional or simple first-order reasoning. We fall back to automated proof procedures for complex theory reasoning. We also use automated proof procedures for rules that we never exercised in any example of our extensive evaluation (Section 3.4) and of which we only have a vague idea (Section 3.2). This part of proof reconstruction is largely untested and is likely to fail for complex problems.

Our translation steps from HOL to MSFOL (Section 2.2) have a specific impact on proof reconstruction. For efficiency reasons, λ-lifting (Section 2.2.2), introduction of explicit application (Section 2.2.3) and separation of formulas and terms (Section 2.2.5) are performed without accompanying proofs, and hence proof replay needs to deliver

Reconstruction technique	Proof rules
Primitive inference or metatheorem	apply-def, asserted, $\text{comm}_{\simeq,\approx}$, hypothesis, $\text{iff}_{\bot}$, $\text{iff}_{\top}$, $\text{iff}_{\sim}$, $\text{mp}_{\rightsquigarrow}$, $\text{refl}_{\simeq}$, sk_{Q}, $\text{symm}_{\simeq}$, $\text{trans}_{\simeq}$, true
Combination of primitive inferences and metatheorems	$\text{cong}_{\simeq}$, def-axiom, $\text{elim}_{\wedge}$, $\text{elim}_{\neg\vee}$, elim_{Q}, $\text{inst}_{\forall}$, intro_{Q}, intro-def, lemma, nnf-neg, nnf-pos, unit-resolution
Automatic proof methods	der, distributivity, pull-quant, push-quant
Combination of the above	rewrite, th-lemma

Table 3.1. Reconstruction techniques for Z3 proof rules

these omitted proofs to turn our oracle-based integration of Z3 into a fully trusted proof component of Isabelle. To minimize the impact and to avoid modifying the Z3 proof structure, we embed this reasoning into the overall reconstruction procedure as much as possible.

Recall that λ-lifting introduces fresh constants as replacements for λ-abstractions and adds definitions to correlate the constants with the λ-abstractions. This is conceptually similar to local definitions in Z3 proofs, and in fact, we handle both features in much the same way (Section 3.3.1) by hiding the definitions in hypotheses.

Introduction of explicit application operators as well as separation of formulas from terms merely add constants to a problem, but do not change its logical structure. Instead of completely removing these additional constants from the Z3 proof which possibly requires to modify also uninvolved steps of the proof such as $\text{cong}_{\simeq}$ steps, we keep these constants in all propositions of the proof. It suffices to adjust replaying of asserted steps as follows. Assume that t is a constituent of a problem. Before given to Z3, translation of t results in a decorated proposition t' where applications are made explicit and terms and formulas are separated. Now let us assume that Z3 finds a proof that uses t', i.e., some asserted rule introduces t' into the following shortened Z3 proof:

$$\dfrac{\dfrac{}{\{t'\} \vdash t'}\ \text{asserted}}{\begin{array}{c}\vdots\\ \Gamma \cup \{t'\} \vdash \bot\end{array}}$$

Replaying the asserted step in Isabelle establishes the equivalence between t and t' via rewriting (rewr) with suitable rules such as for instance the definition of the application operator (Section 2.2.3), and under the assumption that t holds we can conclude that also t' holds:

$$\dfrac{\dfrac{}{\vdash t = t'}\ \text{rewr} \qquad \dfrac{}{\{t\} \vdash t}\ \text{asm}}{\{t\} \vdash t'}\ \text{res}\ \ (\bigwedge x.\, x = y \Longrightarrow x \Longrightarrow y)$$

Since no Z3 inference rule ever inspects or modifies the hypotheses introduced by an asserted rule, we are free to pass on any term as hypothesis. Here, the term t is propagated to the root node of the proof instead of t' as in the original Z3 proof above. No further proof rules need to be adapted.

The remainder of this section describes in detail our optimized approaches to reconstructing Z3 rules that cannot directly be replayed by a primitive inference, a metatheorem or a straightforward combination thereof. We show how to reconstruct local definitions (Section 3.3.1) and Skolemization (Section 3.3.2) as well as propositional reasoning on conjunctions and disjunctions (Section 3.3.3). We also describe the reconstruction of the vaguely specified proof rule th-lemma (Section 3.3.4) and detail how we replay rewrite (Section 3.3.5).

3.3.1. Local Definitions

Names introduced as abbreviations for formulas and terms by intro-def are given *hypothetical definitions* in Isabelle/HOL. A hypothetical definition for some fresh name is an hypothesis that equates this name with some term and thus acts as a definition for this name. Using this concept, the second example for intro-def given earlier (Section 3.2) is replayed in Isabelle as follows, where n is represented by a free variable:

$$\dfrac{\dfrac{}{\{n = t\} \vdash n = t}\,\text{asm}}{\{n = t\} \vdash (\neg n \vee t) \wedge (n \vee \neg t)}\,\text{res}\ (\bigwedge x, y.\ x = y \Longrightarrow (\neg x \vee y) \wedge (x \vee \neg y))$$

Here, the hypothetical definition is $n = t$.

The apply-def rule, used by Z3 to recover the local definition, is faithfully replayed in Isabelle/HOL. Especially, we do not discharge the hypothetical definition at this point. Instead, we keep the hypothesis until after replaying the root node where we discharge it via generalization of the free variable, which does not occur in any hypotheses, and application of the reflexivity rule in analogy to the treatment of Skolemization in [28]. More precisely, we proceed in the following way:

$$\dfrac{\dfrac{\dfrac{\Gamma \cup \{n = t\} \vdash \mathsf{False}}{\Gamma \setminus \{n = t\} \vdash n = t \Longrightarrow \mathsf{False}}\,\text{intro}_{\Longrightarrow}}{\Gamma \setminus \{n = t\} \vdash \bigwedge n.\ n = t \Longrightarrow \mathsf{False}}\,\text{intro}_{\wedge}}{\Gamma \setminus \{n = t\} \vdash \mathsf{False}}\,\text{res}\ (\bigwedge x.\ x = x)$$

The same technique is also applied to hypothetical definitions stemming from the λ-lifting translation step (Section 2.2.2). Those definitions get assumed when replaying the asserted rule and are discharged in the above manner after concluding the root node of the Z3 proof.

3.3.2. Skolemization

Via Skolemization, new Skolem constants, represented as free variables in Isabelle/HOL, are introduced in a Z3 proof. We focus here on discussing how to reconstruct the

$\mathsf{sk}_\exists$ rule. Handling the $\mathsf{sk}_\forall$ rule is analog.

The naive approach of directly modeling the $\mathsf{sk}_\exists$ rule in Isabelle/HOL requires to construct a definition for the Skolem constants, e.g., as follows, where ε denotes Hilbert's choice operator:

$$\frac{}{\{c = (\lambda x.\ \varepsilon y.\ P\ x\ y)\} \vdash (\exists y.\ P\ x\ y) = P\ x\ (c\ x)}\ \mathsf{asm}$$

The hypothetical definition needs be constructed explicitly from the proposition as it is not given in the Z3 proof. Moreover, in case several existential quantifiers are Skolemized in one step, a hypothetical definition for every introduced Skolem constant has to be constructed which can be bulky. Instead of this intricate approach, we directly assume the proposition in much the same way as with the Z3 rules hypothesis and asserted, but with the exception that additional unbound variables (except for the Skolem constants) are explicitly universally quantified:

$$\frac{\dfrac{}{\left\{\bigwedge x.\ (\exists y.\ P\ x\ y) = P\ x\ (c\ x)\right\} \vdash \bigwedge x.\ (\exists y.\ P\ x\ y) = P\ x\ (c\ x)}\ \mathsf{asm}}{\left\{\bigwedge x.\ (\exists y.\ P\ x\ y) = P\ x\ (c\ x)\right\} \vdash (\exists y.\ P\ x\ y) = P\ x\ (c\ x)}\ \mathsf{elim}_\bigwedge$$

After having replayed all Z3 proof steps, i.e., after having concluded the root node, the extra hypothesis is discharged in a way reminiscent of the treatment of local definitions (Section 3.3.1), but here it is slightly more involved. All steps are based on the following Skolemization rule:

$$\bigwedge P, c.\ \ c = (\varepsilon x.\ P\ x) \Longrightarrow (\exists x.\ P\ x) = P\ c \tag{3.3}$$

This Skolemization rule eliminates only one existential quantifier of a premise at a time. We perform a stepwise elimination by interleaving resolutions with this rule and resolutions with the transitivity rule $(\bigwedge x, y, z.\ x = z \Longrightarrow z = y \Longrightarrow x = y)$. For example, consider that the hypothesis $(\exists x, y.\ P\ x\ y) = P\ c\ d$ remained from a Skolemization step at the end of reconstruction where $P\ x\ y$ abbreviates a proposition in which x and y may occur. After introducing it into the proposition and generalizing it, we obtain:

$$\bigwedge v, w.\ (\exists x, y.\ P\ x\ y) = P\ v\ w \Longrightarrow \mathsf{False}$$

The steps to discharge the premise are summarized in Figure 3.3. In short, we eliminate each existential quantifier with the rule (3.3) after resolving with the transitivity rule, and we discharge premises by reflexivity in the reverse order in which they emerge.

Arguably, discharging a Skolemization hypothesis can be done in a simpler way. Yet, the intricate order of resolution steps and the specific Skolemization rule lead to small terms and thus improved efficiency. In Figure 3.3, every occurrence of x in $P\ x\ y$ is replaced only with the Skolem constant c or the variable v, both of which do not increase the size of terms, but never with the much larger Skolem term $(\varepsilon x.\ \exists y.\ P\ x\ y)$. This is especially relevant if x occurs more than once in $P\ x\ y$. There is also no nesting of Skolem terms such as would be obtained by replacing c with its Skolem term in

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\bigwedge v, w.\ \boxed{(\exists x, y.\ P\ x\ y) = P\ v\ w} \Longrightarrow \mathsf{False}}{\bigwedge v, w, z.\ \boxed{(\exists x, y.\ P\ x\ y) = z} \Longrightarrow z = P\ v\ w \Longrightarrow \mathsf{False}}\ (1)}{\bigwedge v, w, c.\ c = (\varepsilon x.\ \exists y.\ P\ x\ y) \Longrightarrow \boxed{(\exists y.\ P\ c\ y) = P\ v\ w} \Longrightarrow \mathsf{False}}\ (2)}{\bigwedge v, w, c, z.\ c = (\varepsilon x.\ \exists y.\ P\ x\ y) \Longrightarrow \boxed{(\exists y.\ P\ c\ y) = z} \Longrightarrow z = P\ v\ w \Longrightarrow \mathsf{False}}\ (1)}{\bigwedge v, w, c, d.\ c = (\varepsilon x.\ \exists y.\ P\ x\ y) \Longrightarrow d = (\varepsilon y.\ P\ c\ y) \Longrightarrow \boxed{P\ c\ d = P\ v\ w} \Longrightarrow \mathsf{False}}\ (2)}{\bigwedge c, d.\ c = (\varepsilon x.\ \exists y.\ P\ x\ y) \Longrightarrow \boxed{d = (\varepsilon y.\ P\ c\ y)} \Longrightarrow \mathsf{False}}\ (3)}{\bigwedge c.\ \boxed{c = (\varepsilon x.\ \exists y.\ P\ x\ y)} \Longrightarrow \mathsf{False}}\ (3)}{\mathsf{False}}\ (3)$$

Figure 3.3. Discharging of a premise that was assumed by a $\mathsf{sk}_\exists$ rule. For the sake of clarity, we omit the set of assumptions Γ in all theorems. The inferences are as follows: (1) is a resolution with transitivity of equality, (2) is a resolution with (3.3), and (3) is a resolution with reflexivity of equality. Premises to which resolution is applied are highlighted.

$(\varepsilon y.\ P\ c\ d)$ causing an exponential blowup of the size of terms. We have seen examples where our approach dramatically improves over simpler solutions that ignore these aspects. In earlier experiments we found that especially avoiding the blowup reduces the overall replay time of the "Judgement Day" suite (Section 3.4.1) by about 10%.

3.3.3. Conjunctions and Disjunctions

Most propositional reasoning in Z3 proofs can be reduced to establishing an implication between two arbitrarily parenthesized conjunctions $p_1 \wedge \cdots \wedge p_m$, call it t, and $q_1 \wedge \cdots \wedge q_n$, call it u, where $\{q_i \mid 1 \leq i \leq n\}$ is a subset of $\{p_i \mid 1 \leq i \leq m\}$. For example, proving an implication between to disjunctions t' and u' falls into this scheme since after applying contraposition we are left to prove that the conjunction $\neg u'$ implies the conjunction $\neg t'$. Such nested conjunctions and disjunctions obviously arise from their polyadic counterparts in Z3, but also from unfolding of the distinct predicate applied to a fixed list of terms leads to conjoined inequations.

Establishing an implication between two conjuncts by rewriting with associativity, commutativity and idempotence of conjunction is far too slow due to the quadratic complexity. Instead, we perform the following, much more efficient approach:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{t\} \vdash t}\ \mathsf{asm}}{\{t\} \vdash p_1 \quad \cdots \quad \{t\} \vdash p_m}\ \text{explode } t \text{ into } p_1, \ldots, p_m}{\{t\} \vdash u}\ \text{join } u \text{ from } q_1, \ldots, q_n}{\vdash t \Longrightarrow u}\ \mathsf{intro}_\Longrightarrow$$

That is, we first derive intermediate theorems $\{t\} \vdash p_i$ for each p_i by assuming t and recursively applying conjunction elimination (a step we call *explosion* of t). These intermediate theorems are stored in a balanced tree, indexed by their conclusion and with lookup in $O(\log m)$ when assuming a fixed maximum size of the p_i. Then, we derive $\{t\} \vdash u$ by recursion over the structure of u, using conjunction introduction, from the intermediate theorems (a step we call *join* of u), and thus conclude the implication. The overall complexity of this approach is $O(m \log m)$.

We apply this technique in reconstructing $\mathsf{elim}_\wedge$, $\mathsf{elim}_{\neg\vee}$, def-axiom and lemma. Three further rules (nnf-neg, nnf-pos, and rewrite) require to establish equivalence between conjunctions or disjunctions instead of implication, and we proceed in the usual way by establishing both implications to conclude equivalence. Unit resolution (Section 3.2) deduces a disjunction from a disjunction (the major premise) and a set of disproved literals. We transform such proof steps into implications of conjunctions by contraposition, i.e., we assume the negated conclusion and show that this together with the literals implies the negated major premise.

There is one further optimization. The rules $\mathsf{elim}_\wedge$ and $\mathsf{elim}_{\neg\vee}$ deduce a literal from a polyadic conjunction or negated polyadic disjunction. Since $\mathsf{elim}_\wedge$ (and similarly $\mathsf{elim}_{\neg\vee}$) is commonly applied to the same premise t several times, deducing a different literal each time, it is more efficient to explode t once and for all instead of repeatedly extracting a single literal. The resulting balanced tree of literals is stored in the proof node that derived t and allows for efficient lookup of individual literals.

3.3.4. Theory Reasoning

We implement reconstruction of th-lemma by sequentially trying decision procedures for linear integer and real arithmetic available in Isabelle/HOL. When they fail, we simplify the goal and retry the arithmetic decision procedures again. This approach of reconstructing th-lemma is expensive as it essentially amounts to a search with more than exponential complexity. Although this task has already been performed by Z3, and Z3 even provides hints for replaying linear arithmetic steps in form of tags and coefficients (Section 3.2), we currently do not exploit this information. Despite the fact that reconstructing large proofs from big industrial benchmarks suffers considerably from our approach, we reckon that spending more effort on improving proof replay is unlikely to give dramatic speedups for typical Isabelle/HOL problems, especially because linear arithmetic plays a small role for most of them (Section 2.5.3).

Isabelle's arithmetic decision procedures will typically fail when faced with huge terms as they dive into subterms that do not contribute to the proof. In contrast, Z3 reasons only shallowly. We mimic this behavior by abstracting terms first, i.e., replacing subterms outside the fragment of arithmetic with fresh variables. We especially observed that if-then-else-expressions in terms, that are treated atomically by Z3's arithmetic decision procedures, had a deteriorating effect on Isabelle's arithmetic decision procedures, since they led to case splitting internally causing an exponential blowup. By abstracting away if-then-else-expressions we were able to also reconstruct such proofs efficiently.

3.3.5. Rewriting

Z3's simplifications and canonicalizations that are reflected in rewrite proof steps span the range from simple propositional reasoning to theory-specific normalizations. Reconstruction sequentially tries a list of techniques to replay such steps, and we detail them in their order applied.

Metatheorems Matching a theorem's conclusion against a given term and, if successful, instantiating the theorem accordingly is typically much faster than deriving the instance again. By studying the actual usage of rewrite in Z3's proofs, we identified over one hundred useful metatheorems referred to as set R. These include propositional tautologies such as $(p \longrightarrow q) = (q \vee \neg p)$, theorems about equality, for instance $(x = y) = (y = x)$, and theorems of linear integer and real arithmetic, e.g., $x + 0 = x$. Together, these theorems allow a major part of all terms given to rewrite to be proved by instantiation alone. We store all metatheorems in a term net [41, chapter 14] to speed up the search for a match.

Motivated by failed proof reconstruction attempts, we made the set R of metatheorems extensible by users of Isabelle/HOL. This is also a way to overcome the incompleteness of proof reconstruction for Z3 that is due to the vague documentation. A failed replay of a rewrite step with conclusion t can be compensated for by adding to R a specific metatheorem that is matched by t. As a consequence, subsequent reconstruction attempts of the same Z3 proof will succeed.

Since this approach turned out to be tremendously helpful, we apply the set of rules R also to other vaguely specified proof rules (def-axiom, nnf-pos, nnf-neg and th-lemma) before trying anything else. The penalty for matching against a large collection of metatheorems is outweighed by the improved coverage: Users of Isabelle/HOL are willing to wait a few more milliseconds for an automatic proof tool to finish if this spares them from the tedious search for a possibly lengthy manual proof.

Propositional reasoning Simplification and canonicalization of propositional formulas is an important prerequisite for efficient SMT solving. In rewrite, Z3 proofs express equivalences that establish (1) permutations of conjunctions and disjunctions, (2) a generalized form of the rule of the excluded middle $(l_1 \vee \ldots \vee l_n) = \mathsf{True}$, where there are i and k such that l_i is syntactically equivalent to $\neg l_k$, and the dual rule of contradiction $(l_1 \wedge \ldots \wedge l_n) = \mathsf{False}$, and (3) simplifications of propositions involving further logical connectives such as $(\mathsf{True} \longrightarrow p) = p$. We replay (1) with the explode/join technique (Section 3.3.3) by splitting the equivalence into two implications. In case (2) the generalized excluded middle is reduced via a variant of contraposition to the dual form with conjunctions which in turn is proved by considering both implications, i.e., by instantiating the rule $(\bigwedge x.\ \mathsf{False} \Longrightarrow x)$ and by reusing the conjunction explosion (Section 3.3.3) as follows:

$$\dfrac{\dfrac{\dfrac{}{\{l_1 \wedge \ldots \wedge l_n\} \vdash l_1 \wedge \ldots \wedge l_n}\,\text{asm}}{\{l_1 \wedge \ldots \wedge l_n\} \vdash l_1 \quad \cdots \quad \{l_1 \wedge \ldots \wedge l_n\} \vdash l_n}\,\text{explode}}{\{l_1 \wedge \ldots \wedge l_n\} \vdash \mathsf{False}}\,\text{contra}}{\vdash l_1 \wedge \ldots \wedge l_n \Longrightarrow \mathsf{False}}\,\text{intro}_{\Longrightarrow}$$

The contra inference steps finds the contradicting literals l and $\neg l$ from the exploded conjunction and resolves them with the contradiction rule $(\bigwedge x.\ x \Longrightarrow \neg x \Longrightarrow \mathsf{False})$. For (3), we fall back to automatic proof methods of Isabelle, although this should rarely happen since matching against the set R of metatheorems as described earlier should already cover many cases.

Linear arithmetic Instances of rewrite that express equalities of linear arithmetic relate one arithmetic term to an equal term that is in some canonical or simplified form. We sequentially try similar techniques as for replaying th-lemma steps, i.e., we abstract the equality first by replacing subterms that are neither arithmetic expressions nor logical formulas with fresh variables and then apply Isabelle's simplifier before Isabelle's arithmetic decision procedures. In the few cases where this approach fails, we perform a second round, due to a suggestion by Jasmin Christian Blanchette, where we abstract slightly less parts of the equality. That is, we replace smaller subterms by fresh variables thereby leaving applications of some nonarithmetic constants to terms unabstracted and retry simplification followed by arithmetic decision procedures. For example, consider the following proposition where x, y and z are integer variables:

$$(\mathsf{f}\ (\mathsf{if}\ \mathsf{P}\ \mathsf{then}\ \mathsf{g}\ x\ \mathsf{else}\ y) - x < z) = (\mathsf{if}\ \mathsf{P}\ \mathsf{then}\ \mathsf{f}\ (\mathsf{g}\ x) - x < z\ \mathsf{else}\ \mathsf{f}\ y - x < z)$$

After abstracting this proposition we obtain:

$$(a - x < z) = (\mathsf{if}\ \mathsf{P}\ \mathsf{then}\ b - x < z\ \mathsf{else}\ c - x < z)$$

where a, b and c are fresh integer variables that abstract over nonarithmetic subterms. Clearly, this equivalence is not valid and hence not provable by the simplifier or any arithmetic decision procedure. The problem is that the abstraction hides f applied to the if-then-else-expression. The more liberal abstraction applied in the second round leaves all applications of f unabstracted:

$$(\mathsf{f}\ (\mathsf{if}\ \mathsf{P}\ \mathsf{then}\ a\ \mathsf{else}\ y) - x < z) = (\mathsf{if}\ \mathsf{P}\ \mathsf{then}\ \mathsf{f}\ a - x < z\ \mathsf{else}\ \mathsf{f}\ y - x < z)$$

where a is a fresh integer that abstracts the subterm g x. For this equivalence, the simplifier now succeeds.

Similar as with th-lemma, this entire approach is potentially costly due to the sequential application of different techniques and since the invoked arithmetic decision procedures have exponential complexity. In contrast to th-lemma, however, Z3 provides no further hints for rewrite, and that makes it challenging to implement an efficient method tailored to reconstruct linear arithmetic equalities (and possibly rewrite instances for other theories as well).

Injective functions Z3 can discover from the constituents of a problem that a function is injective and thus there must be an inverse function for it. The hypothetical inverse function is then used in further steps of the proof. We illustrate this abstract description with an example. Let us assume that a problem given to Z3 contains the following proposition which expresses that f is injective in its first argument:

$$\forall x, y, z.\, f\, x\, z = f\, y\, z \longrightarrow x = y \tag{3.4}$$

Z3 is then able to infer that there must be an inverse function, call it g, satisfying the following proposition:

$$\forall x, z.\, g\, z\, (f\, x\, z) = x \tag{3.5}$$

This reasoning is captured in a single rewrite step that expresses the equivalence between (3.4) and (3.5) including the Skolemization that produces the fresh name g for the inverse function.

We reconstruct this inference similar to local definitions (Section 3.3.1) or Skolemization (Section 3.3.2) w.r.t. the fresh name g, albeit with a specialized procedure to establish the two implications that together form the required equivalence. For the left-to-right implication, we first construct a hypothetical definition for g using the Isabelle/HOL function inv:

$$g = (\lambda z.\ \mathsf{inv}\ \ (\lambda x.\, f\, x\, z))$$

and can then reduce the problem to the following metatheorem about inv:

$$\bigwedge f.\ (\forall x, y.\, f\, x = f\, y \longrightarrow x = y) \Longrightarrow (\forall x.\ \mathsf{inv}\, f\ (f\, x))$$

For the right-to-left implication, we first prove the following equational tautology using Isabelle's simplifier:

$$\forall x, y, z.\, f\, x\, z = f\, y\, z \longrightarrow g\, z\, (f\, x\, z) = g\, z\, (f\, y\, z)$$

From this we conclude (3.4) by rewriting with the assumption (3.5).

3.4. Evaluation

The evaluation of our translation from higher-order to many-sorted first-order logic (Section 2.5) already includes a partial evaluation of proof reconstruction: In case that reconstruction with *metis* failed, the proofs found by one of the SMT solvers could only be checked by a subsequent invocation of Z3 with proof reconstruction enabled. Hence, the contribution of SMT solvers to the proof automation of Isabelle/HOL depends to a substantial extent on the feasibility of proof replay for Z3. Yet, we have not evaluated how efficient proof reconstruction for Z3 is. Theoretically, proof checking should be faster than proof finding, but various reasons, some of which we indicate later, can turn the relation around in practice. Nevertheless, we claim that reconstruction of Z3 proofs

is efficient for typical problems in Isabelle/HOL (Section 3.4.1) and even for more involved problems (Section 3.4.2). To support the first claim, we provide runtimes of both finding and checking proofs for the "Judgement Day" problem suite (from Section 2.5.1) as well as profiling data for proof reconstruction. For the second claim, we chose industrial and artificial benchmarks from the annual SMT competition and study similar figures. Further evaluations of our proof replaying code [28,33] provide results obtained from older versions of Z3 and compare our efficient reconstruction for Z3 with proof replay for the SMT solver CVC3 in the interactive theorem prover HOL Light.

3.4.1. Judgment Day Benchmarks

We claimed earlier that checking Z3 proofs in Isabelle/HOL is efficient and does not hinder interaction of users with Isabelle. More precisely, we claimed that proof reconstruction for typical Isabelle problems takes in general only a fraction of a second and in rare cases only few seconds. We will now give evidence to this claim.

In the previous chapter, we already used the "Judgment Day" suite (Section 2.5.1) as a representative collection for typical problems in Isabelle/HOL, and we follow this approach here as well. For evaluating proof reconstruction, we selected from this suite all problems—exactly 1000—that could be proved by Z3. We kept these problems from a run of the "Judgment Day" suite and stored them in the SMT-LIB format. This way, our evaluation is focused on proof reconstruction instead of unnecessarily spending time on proof search for possibly unprovable problems. As a side effect, this modified setup avoids invocations of *metis* that alone succeeds in reconstructing 75.9% of all proofs found by Z3 (Table 2.3) thus reducing the number of proofs to exercise our proof replay. Since we intend to evaluate proof reconstruction for Z3 here, we try to replay all 1000 proofs by Z3 alone. In contrast to the code used in the previous section, we need to compensate here for the missing context as proof replay runs separately from the "Judgement Day" theories and the Z3 proof certificates do not reveal which types have been introduced by monomorphization (Section 2.2.1) and which assertions represent only hypothetical definitions (Section 2.2.2). We thus map uninterpreted sorts in the proof certificate to fresh HOL types. Discharging of hypotheses is confined to those introduced by specific inference rules (local definitions, Skolemization, and rewriting injective functions). Rewriting and checking of assumptions as detailed earlier (Section 3.3) is mostly disabled. Our changes do not impair the central functionality of proof reconstruction and should not affect the runtime much.

On the selected SMT-LIB problems, we ran Z3 to measure the CPU time required for finding a proof. We also collected both the file size of the found proof certificate as an approximation of the number of inference rules and the size of propositions forming the proof. We then applied the slightly modified version of our Isabelle/HOL reconstruction code to these proofs and measured the overall CPU time. In a second replay run, we performed detailed profiling, i.e., we measured the cumulative CPU times for each inference rule as well as for parsing and discharging hypotheses. We performed our measurements on the same machine as for evaluation or SMT integration (Section 2.5.1).

Z3 rule	r-occs	r-time	a-time
cong$_\simeq$	20.93%	.93%	.03
rewrite	20.14%	43.72%	1.68
trans$_\simeq$	11.22%	.57%	.04
mp$_{\rightsquigarrow}$	9.80%	.24%	.02
unit-resolution	7.63%	.60%	.06
intro$_Q$	7.31%	3.82%	.41
refl$_\simeq$	5.30%	.06%	.01
other 18 rules	17.67%	25.01%	.05
parsing		24.62%	21.06
discharging		.43%	.37

(a) Sorted by decreasing occurrences

Z3 rule	r-time	r-occs	a-time
rewrite	43.72%	20.14%	1.68
th-lemma	21.03%	0.89%	18.23
intro$_Q$	3.82%	7.31%	.41
asserted	1.39%	.91%	1.19
inst$_\forall$	1.00%	2.96%	.26
other 20 rules	3.99%	67.79%	.03
parsing	24.62%		21.06
discharging	.43%		.37

(b) Sorted by decreasing runtime

Table 3.2. Profiling data for Z3 proof reconstruction of the "Judgment Day" benchmarks. The column r-occs shows the relative number of occurrences of proof rules, the column r-time shows the relative relative runtime of reconstruction steps, and the column a-time shows the average runtime of reconstruction steps in milliseconds or the median thereof when aggregating several proof rules.

Our results are as follows. The median runtime for proof finding by Z3 is 136 ms, and the median proof size is 5 KB. That is, typical proof certificates can be found almost instantly and are fairly small. Especially the former figure is important for users of Isabelle/HOL: If Z3 is able to find a proof for a conjecture, it typically does so very quickly without letting the user wait. Proof reconstruction for the 1000 proofs from Z3 succeeds for all but seven cases. These failures are due to the incompleteness of proof replay for nonlinear arithmetic (in three cases) and bugs in the proof generated by Z3 (in the remaining four cases), which we reported to Z3's developers and which have been fixed in recent versions. Reconstruction takes 48 ms (median) for a single proof which translates into a ratio of .26 between proof checking and proof finding. Hence, proof checking is considerably faster than proof finding.

Profiling (Table 3.2) reveals which inferences among the 25 seen in all proofs take the largest share on the number of occurrences and which of them contribute most to the overall runtime. When optimizing proof reconstruction, we first focused on reducing the runtime for the rules that occur most often. Table 3.2a shows that, beside rewrite, the most frequent rules, i.e., those with at least 5% relative occurrences, can be reconstructed very efficiently since their relative number of occurrences is mostly at least one order of magnitude greater than their relative runtime and their average runtime is a small fraction of a millisecond. Hence, only little room remains to improve the efficiency of our proof replaying code by improving reconstruction for the rules that run longest. Table 3.2b shows the seven most costly inferences rules w.r.t. runtime, i.e., those taking at least 1% of the overall runtime, and compares them with their number of occurrences relative to all other proof rules. This table is also exhaustive w.r.t. those rules for which replaying takes longer than a millisecond on average, i.e., this table

covers also the most costly rules in absolute numbers. We see that 89.37% of the time is spent with parsing and reconstructing rewrite and th-lemma steps whereas replay time of the remaining 23 proof rules is minor (10.20%). Especially rewrite and th-lemma together take nearly two third (64.75%) of the entire time. This figure could be slightly improved by providing more metatheorems to the set R (Section 3.3.5). A much stronger impact could be gained if no search was necessary for replaying rewrite and th-lemma instances. Especially for theory lemmas, we could improve the reconstruction time by taking advantage of the hints provided by Z3. For rewriting steps, we do not see possible performance gains without further information from Z3 such as, for example, splitting one rewrite step into smaller inferences with clear semantics [34].

We summarize our key findings as follows:

- Z3 proof checking is faster than proof finding for typical Isabelle/HOL problems.
- Except for rewrite, Z3 proof replaying is highly optimized for the most important inference rules.
- The rewrite rule is the most expensive inference step w.r.t. proof reconstruction.
- Efficiency of replay for th-lemma rules can be optimized further.

3.4.2. SMT-COMP Benchmarks

We have seen that reconstructing proofs for typical Isabelle/HOL problems is highly efficient. Once found, such proofs can typically be replayed within a fraction of a second, in rare cases within a few seconds. Nevertheless, such an evaluation does provide little to no information about how reconstruction scales with the proof size. Especially bigger proofs are likely to demonstrate whether our optimizations are worthwhile, i.e., that they lead to short replay times, and bigger proofs also help to uncover remaining performance bottlenecks in our implementation. To this end, we consider problems from other domains.

The SMT-LIB [15] has collected some tens of thousands of industrial and crafted MSFOL benchmark files, which are classified into different *logics*. Each logic is a combination of theories such as equality and uninterpreted functions (*UF*), linear integer arithmetic (*LIA*), linear real arithmetic (*LRA*) and extensional arrays (*A*). Some logics are restricted to quantifier-free (*QF*) benchmarks, and some logics allow triggers to be provided for quantifiers (+*p*) whereas some forbid triggers (-*p*). The name of a logic is formed by concatenation of these abbreviations. The annual SMT competition (SMT-COMP) selects out of each logic about 200 benchmarks representing both simple and complicated problems. For our evaluation we use the selection from the SMT competition of 2009 [14]. Not all of the competition logics are covered by our proof reconstruction, for instance those involving fixed-size bitvectors. Other logics, for example those expressing problems of difference logic, seem to be too specialized for our purpose. After all, we want to evaluate a tool that is used as a general-purpose prover within Isabelle/HOL. Therefore, we concentrate here on logics that involve either quantifiers or uninterpreted functions possibly combined with linear arithmetic.

	Solved by Z3			Reconstructed in Isabelle/HOL			
Logic	#	Time	Size	Success	Timeout	Failure	Time
AUFLIA+p	192	36	5 KB	100%	0%	0%	110
AUFLIA-p	192	32	4 KB	100%	0%	0%	96
QF_AUFLIA	86	24	146 KB	84%	16%	0%	1482
QF_UF	73	844	2 MB	97%	3%	0%	5,976
QF_UFLIA	89	36	108 KB	94%	6%	0%	3,336
QF_UFLRA	100	96	700 KB	100%	0%	0%	6,220
Total	732	48	65 KB	97%	3%	0%	968
"Judgment Day"	1000	136	5 KB	99%	0%	1%	48

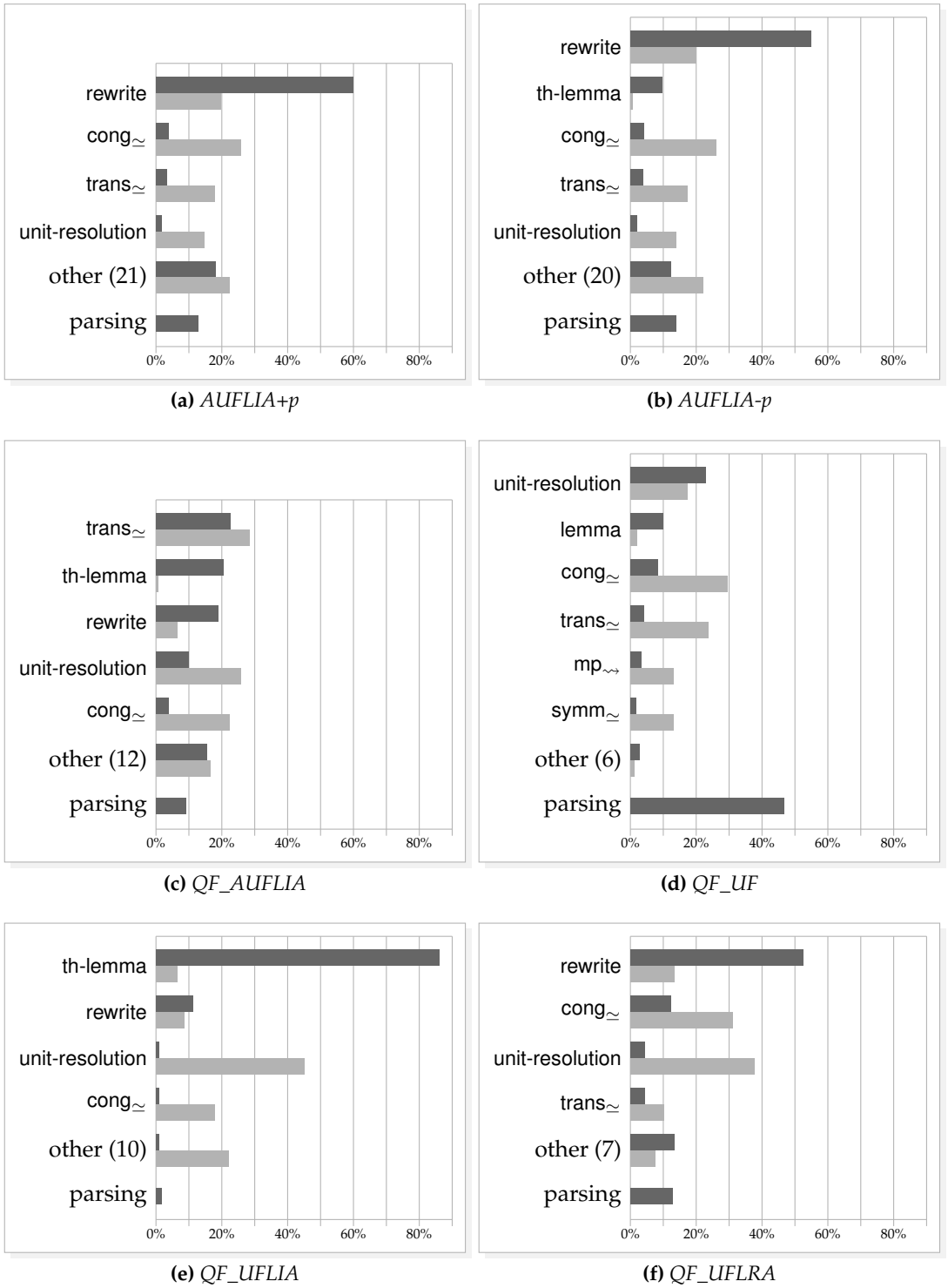
Table 3.3. Results for SMT-COMP benchmarks. All times are given in milliseconds. All times and proof sizes are median values. The reconstruction times only count successfully reconstructed benchmarks.

We turned benchmarks that make use of extensional arrays into benchmarks that treat them uninterpreted by adding suitable axiomatization and replacing the interpreted sort of arrays by an uninterpreted sort and the interpreted array functions by uninterpreted functions.

Our experimental setup is similar as before (Section 3.4.1). We especially re-use the modified proof reconstruction code and the same machine to run our experiments. In addition, we put a time limit of 60 seconds and a memory limit of 512 MB on Z3, and we also limit the runtime of each proof reconstruction attempt to five minutes.

Table 3.3 shows our results in comparison to the results of our previous "Judgment Day" evaluation. From these median values we see that Z3 is faster on the SMT-COMP benchmarks and produces much larger proofs for them. In fact, the largest proof we have seen was 124 MB in size, and still it can be successfully replayed within the time limit. Proof reconstruction does not fail for any of the considered SMT-COMP benchmarks which indicates that our implementation has reached a good level of maturity and is indeed applicable to larger proofs as well. Yet, proof reconstruction times out in 3% of all cases, and the time to reconstruct a proof is also considerably longer than the time spent for finding it. One reason to this is that these proofs contain huge propositions, and we suspect that this inevitably leads to some loss of performance in Isabelle.

Some more insights into performance bottlenecks are revealed by detailed profiling data (Figure 3.4). For each logic, we only show those rules that contribute at least 10% to either the proof reconstruction time or the number of reconstructed inference steps and group all other rules together (with the number of different rules in that group). For *AUFLIA+p*, *AUFLIA-p* and *QF_UFLRA*, the rewrite rule dominates runtime—a result that does not come as a surprise after the evaluation of the "Judgment Day" proofs. In proofs of *AUFLIA-p*, *QF_AUFLIA* and *QF_UFLIA*, the relative reconstruction time for the th-lemma rule exceeds its relative occurrences dramatically, which is also similar to our results of the previous section and indicates some potential for future improvements. Especially for *QF_UFLIA*, optimizations for replaying th-lemma steps that avoid



(a) *AUFLIA+p* **(b)** *AUFLIA-p*

(c) *QF_AUFLIA* **(d)** *QF_UF*

(e) *QF_UFLIA* **(f)** *QF_UFLRA*

Figure 3.4. Profiling data for Z3 proof reconstruction of the SMT-COMP benchmarks. Dark bars symbolize the relative reconstruction time, and light bars indicate the relative number of reconstructed inference steps.

the expensive search performed by Isabelle's arithmetic decision procedures will be tremendously beneficial as over 80% of the time is spent on the reconstruction of these steps. When no theory reasoning is involved as in *QF_UF*, we see that replaying most inferences is highly optimized since nearly every rule takes a larger share on the number of reconstructed inferences than on the reconstruction time, and parsing time for the proofs dominates. A similar observation about the efficiency of replaying proof rules besides th-lemma and rewrite also applies to the other five logics. Moreover, recall from Table 3.3 that replaying proofs from *QF_UF* is slightly faster than replaying proofs of *QF_UFLRA* although the proofs from *QF_UF* are nearly three times as big as those of *QF_UFLRA*. Hence we conclude that our optimizations (Section 3.3) have not been in vain. Performance jumps can only be gained from improving the reconstruction of theory-related inferences (th-lemma and certain instances of rewrite).

Our key findings are thus as follows:

- Reconstruction is still feasible for large proofs.
- Most proof rules can be reconstructed efficiently.
- Theory reasoning (th-lemma and rewrite) is the major bottleneck in reconstruction performance.

3.5. Related Work

Distrusting external automatic provers, and oracles in general, is common practice in the interactive theorem prover community. Since formally verifying an entire prover as was announced for an SMT solver in [48] is typically not feasible, the common solution is to request a proof certificate from the external prover which can be independently checked in the interactive theorem prover. At least three different checking approaches have been studied: *stepwise proof reconstruction* of the proof certificate in the inference system of the interactive theorem prover [1, 85, 86, 115, 137, 139], *translation* of the proof certificate into proof text of the interactive theorem prover [35, 139, 148], and applying a *verified checker* that is a program formally verified within the interactive theorem prover and obtained via reflection [4, 22, 40, 82]. It has been shown repeatedly [4, 5] that a verified checker typically outperforms stepwise proof reconstruction on the same benchmarks including our work and despite our optimizations. Nevertheless, remarkable performance has been achieved with the latter approach, too, for replaying proofs of SAT solvers [163] and QBF solvers [98, 100, 162]. From the work of Weber and Amjad [163], we adopted the beneficial principle to replay every inference step at most once, i.e., we ignore unnecessary inferences and re-use results as much as possible.

Proof checking for SMT solvers Only few current SMT solvers produce proof certificates among which are clsat [132], CVC3 [17], Fx7 [121], veriT [21, 36] and Z3 [58]. Each of them has a distinct proof format that limits portability of checker implementations—a common proof language such as TSTP [153] has yet to emerge. The format of proof

certificates produced by clsat and Fx7 were designed specifically for efficient checking, unlike those of CVC3 (with hundreds of small-step inferences rules) and Z3. Consequently, there are very efficient proof checkers for the former solvers, but none of them runs in the context of an interactive theorem prover.

Several integrations of SMT solvers with interactive theorem provers already comprise stepwise proof reconstruction. However, checking of haRVey certificates in Isabelle/HOL [71, 87] ignores linear arithmetic, checking of CVC-Lite and CVC3 proofs in HOL Light [73, 116] does not cover quantifiers, and likewise replaying of Ergo proof traces in Coq [48] is confined to the unquantified theory of equality and linear arithmetic. Hence, in the light of applying SMT solvers as strong backend provers for interactive theorem provers where especially quantifiers are encountered in almost every problem, we consider these implementations as insufficient. In comparison with the implementation for replaying CVC3 proofs in HOL Light, we found that our proof reconstruction, also due to our optimizations, outperforms the former in nearly all cases [33].

Documentation of inference systems Only few inference systems of SMT solvers are well documented [121, 132] which unnecessarily complicates implementing proof reconstruction [34]. In fact, most SMT proof checkers have been written by developers of the corresponding SMT solvers themselves or in tight connection with them. This also applies to our proof reconstruction for Z3 which required an amount of reverse engineering and communication with the developers since only sparse documentation has been provided [58].

Fast proof reconstruction techniques Using metatheorems to directly represent the SMT solver's inferences has been adopted from the proof reconstruction for CVC3 in HOL Light [73, 116].

Efficiently replaying unit resolution steps from a SAT solver in an LCF-style theorem prover has been detailed in [163]. This technique exploits a particular representation of clauses such that a long chain of unit resolution steps can be checked much faster than with our approach (Section 3.3.3). Yet, a typical Z3 proof mixes unit resolution steps with other inferences that are incompatible with the special clause representation, and translating clauses into the special form and back again outweighs potential performance gains.

Replaying Skolemization steps can also be efficiently implemented with the first approach that we have only briefly indicated (Section 3.3.2). This has been done in [28, 33] and builds on earlier work [73]. Yet, this approach requires to explicitly construct definitions for Skolem constants from the proof step's conclusion—an intricate and entirely unnecessary task as we have demonstrated (Section 3.3.2). Alternatively, one could Skolemize the assertions before giving them to SMT solver [87], but we have not investigated such a preprocessing approach.

Chapter 4.

Verifying Imperative Programs

Contents

4.1. Introduction

Automatic program verification was and still is one of the main driving forces behind the development of SMT solvers. Indeed, the need for increased proof automation in the context of program verification originally motivated our integration of SMT solvers with Isabelle. The motivating application came from combining the C code verifier VCC (Section 4.2) with Isabelle that resulted in a tool that we call HOL-Boogie (Section 4.3). We describe a complex case study for HOL-Boogie that aims at formally verifying a red-black tree implementation in C (Section 4.4). By means of this example, we evaluate HOL-Boogie and describe problems related to proving verification conditions from VCC. Some of these problems are overcome when letting VCC cooperate abstractly with Isabelle (Section 4.5). We also evaluate this approach on a complex case study taken from the domain of graph algorithms (Section 4.6). We conclude by reviewing related work (Section 4.7).

4.2. VCC: An Automatic Program Verifier for C

VCC [44] is an automatic program verifier for C [88]. It follows the philosophy that verification should be based primarily on annotations (Section 4.2.1) that are directly inserted into the code and discharged automatically. Thus, VCC stands in the tradition of tools such as ESC/Java [70] and Spec# [13], and shares with them the choice of many-sorted first-order logic (MSFOL) as annotation language.

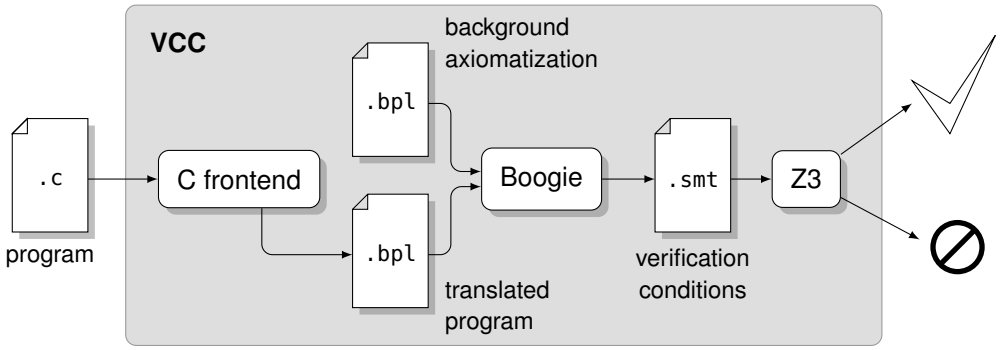


Figure 4.1. Overview of the VCC architecture. We use the typical file endings to distinguish files participating in the process: `.c` stands for a C file, `.bpl` stands for a Boogie file, and `.smt` stands for an SMT file.

VCC comprises the following three tools (Figure 4.1): The *C frontend* translates from C annotated with specifications in MSFOL (Section 4.2.1) to the intermediate imperative *Boogie language* [60], *Boogie* [10] translates from the Boogie language to MSFOL formulas, and Z3 automatically proves the resulting formulas. The translation from annotated C to the Boogie language makes explicit the memory model (Section 4.2.2) and inserts additional checks for, e.g., avoiding range overflow or null-pointer dereference, but keeps the control flow intact by mapping branches and loops of C to Boogie counterparts. Boogie is a verification condition generator that turns an imperative Boogie program with first-order annotations into MSFOL formulas (Section 4.2.3) for partial program correctness. The Boogie language modularizes code into procedures that consist of assignments, control-flow constructs and assertions in first-order logic. Moreover, the language allows to specify a *background theory* by means of type and function declarations as well as first-order axioms setting up the specification against which Boogie programs are verified. In the case of VCC, this background axiomatization expresses, among other things, the heap model, typed pointers and the concept of valid pointers (Section 4.2.2). To associate errors exposed by a failed verification attempt with source code positions, VCC inserts marker symbols (Section 4.2.4) that we exploit in HOL-Boogie.

Guaranteeing soundness of VCC is difficult due to its layered architecture consisting of several individual tools. Unsoundness can creep in from bugs in the translations performed by the C frontend or Boogie, from bugs in Z3, or from inconsistencies in the background axiomatization. This risk has to be taken into account when formally verifying code with VCC, but we indicate later (Section 4.3) how to partly improve this situation.

4.2.1. The VCC Annotation Language

As an assertional code verifier, VCC provides several different annotations all of which are enclosed in `_(` and `)`. When an annotated program is compiled into executable code,

```
struct S {
  int i;
  _(ghost bool map[\integer])
  _(invariant \forall \integer j;
      0 < j && j < i ==> map[j])
};
```

```
_(ghost _(pure) bool g(int i)
    _(requires i > 0)
    _(returns i - 1))
```

```
int f(int i  _(ghost \integer x))
  _(requires i == x && x > 0)
  _(ensures \result == 0)
{
  _(ghost \integer y)
  _(assume y == x)
  while (i > 0)
    _(invariant i >= 0)
  {
    i = i - 1;
    _(ghost y = y - 1)
  }
  _(assert i == y)
  return i;
}
```

Figure 4.2. Contrived C code with annotations

all of these annotations are ignored. Figure 4.2 gives a contrived example of C code with different kinds of annotations that are explained below.

The most basic annotations are to assert a formula via **`assert`** resulting in a proof obligation, and to assume a formula via **`assume`** to be taken as an additional fact for any subsequent proof obligations. These two annotations are complemented by loop invariants, expressed by **`invariant`**, to prove properties of loops.

The specification of a function is given by a contract consisting of the following annotations. Preconditions, marked as **`requires`**, restrict invocations of a function, and postconditions, marked as **`ensures`**, express properties guaranteed to hold on return of a function. A **`returns`** annotation is a special postcondition that gives a term which is equal to any result of the function.

For the purpose of specification, VCC provides means for defining types and functions, for adding fields to structures and parameters and local variables to functions parameters as well as for writing special code. These so-called *ghost types*, *ghost functions*, *ghost fields*, *ghost parameters*, *ghost variables* and *ghost code sections* form the basis of the annotation language and establish a constructive, proof-oriented style [44]. Ghost types can either be C types or logical types such as mathematical integers, written as **`\integer`**, tuples or maps, written in the C array notation. Ghost functions such as `g` (Figure 4.2) can, like C functions, be annotated with contracts and can additionally be declared **`pure`** when they only read from memory without altering it. Ghost functions with a body are verified just like normal C functions, and without a body they serve only as abstract specification devices. Ghost fields such as `map` and ghost variables such as `y` (Figure 4.2) can be used to store additional data, e.g., to maintain inductively defined information, and ghost parameters such as `x` can pass additional information from and to functions. They together form the *ghost state* on which ghost code such as `y = y - 1` (Figure 4.2) freely operates. VCC checks that no information flows from

the ghost state to the ordinary C state and that all ghost code sections terminate. These two conditions together guarantee that the ghost code and the ghost state can be erased without affecting the behavior of the program.

4.2.2. The VCC Memory Model

C supports a very liberal way of accessing memory by pointer arithmetic. Types give merely a hint on the size of the accessed memory chunk, but addressable chunks can overlap arbitrarily in memory. This rules out the direct adoption of simpler memory models such as those for Java [99] or Spec# [11]. Moreover, C pointers can address individual fields of structures, and fields of a structure can again be of a structure type, i.e., structures can be embedded in structures. Hence, a pointer to a structure can at the same time refer to the first field of the structure, and, if that again is a structure, to the first field of that structure, and so. Although this structural hierarchy contributes to the complexity of candidate C memory models, it is the liberal pointer arithmetic that makes designing memory models for C particularly challenging.

Yet, most C programs are written in a *typesafe* way where no two pointers address overlapping chunks of memory except for data within a structure. Ignoring structures for the moment, this observation allows to view memory as a collection of nonoverlapping *objects* instead of flat, byte-wise addressable memory with arbitrary pointer arithmetic. Note that in this setting, an address uniquely identifies an object. When re-considering structures, one needs to extend this model since each structure is itself addressable as an object and so are the fields of the structure. This breaks the view of memory as a collection of nonoverlapping objects as each field overlaps with the structure. Yet, the object view can be rescued by adding the exception that fields overlap with (a part of) a structure. Now, an address can refer to both a structure and its first field, but an address and a type together again suffice as unique identifier for objects. For note that fields of a structure are always of different type than the structure. VCC is based on this typed object view of memory [30, 31, 46]. It maintains a set of all *valid* pointers that adhere to this model, and it checks that memory is accessed only with valid pointers.

VCC partitions the set of objects using an ownership model [44, 45, 47] that is modeled after that of Spec# [11]. Each object has at most one owner, and the induced ownership graph is a forest. The root of each tree is a thread of execution. The set of objects transitively owned by one object is called its ownership domain.

Also adopted from Spec# are object invariants [11], written as **`invariant`** annotations, that are associated with structures and that specify conditions on their fields (Figure 4.2). Invariants are tightly coupled to ownership, because an invariant can only depend on objects in the ownership domain. Since the ownership domain of a structure changes during the execution of a program and fields of a structure get updated, most structure invariants necessarily break at particular program points. VCC thus associates with each structure a special Boolean ghost field **`closed`** which expresses, when set, that the invariants of the structure hold. Consequently, the fields of a structure can only be altered while the structure is open. When a structure directly owned by the cur-

rent thread of execution is closed, it is called **wrapped**. If it is open, it is called **mutable**. Unwrapping a structure introduces its invariants as assumptions. Wrapping a structure is preceded by assertions that check its invariants.

4.2.3. Generating Verification Conditions

VCC translates each C function into a Boogie procedure. This translation adds assertions which, among other things, prevent integer overflows, check that arrays are accessed within their bounds and check that memory is accessed by valid pointers only. Moreover, the translation makes the program state and the heap as well as memory accesses explicit, because these have no counterparts in the Boogie language. The resulting code is mapped to Boogie that provides the same language constructs as annotated C does. There are especially all control-flow constructs from C available. In addition, Boogie's specification language comprises counterparts for annotations such as assertions, assumptions, loop invariants and function contracts as well as for defining specification types and specification functions. Hence, translating from annotated C, which has been enriched with additional assertions and an explicit program state, to Boogie is straightforward.

In the remainder of this section, we concentrate on the translation from Boogie procedures to verification conditions. The first step encodes structured control-flow constructs in a core language of Boogie. The second step applies transformations that result in a passive program. Finally, the third step constructs a first-order formula from a passive program. All presented code uses Boogie syntax.

First step Boogie code that uses structured control-flow constructs such as **while** loops is translated into a core language of Boogie with the following grammar, where identifiers and variables are arbitrary names and where expressions are first-order formulas or terms over equality and integer arithmetic:

Procedure	::=	Block$^+$
Block	::=	Identifier : Statement* Goto
Statement	::=	Variable := Expression ;
	|	**assert** Expression ;
	|	**assume** Expression ;
	|	**havoc** Variable ;
Goto	::=	**goto** Identifier* ;

Each block is labeled by an identifier and consists of a sequence of statements followed by a set of successor blocks. Semantically, each block corresponds to a transition relation between states where a state is an assignment of values to the variables of a procedure, and a **goto** statement corresponds to a composition with the intersection of the successor transition relations. Intuitively, a **goto** performs a nondeterministic jump to one of the successor blocks. If there is no successor block, the program gets stuck. An **assert** statement constraints the subsequent transition whereas an **assume** statement

weakens it. That is, each assertion produces an obligation to be proved while an assumption inserts a property to be used. Consequently, if an expression e evaluates to $\top$, then **assert** e as well as **assume** e have no effect. Otherwise, if e evaluates to $\bot$, then **assert** e is a failure, i.e., it goes wrong, and **assume** e is a terminal success eliminating effectively all subsequent proof obligations [60]. An assignment updates the program state. A **havoc** statement corresponds to an assignment of an arbitrary value.

When translating from structured Boogie procedures as constructed by the C frontend to procedures in the core language, assertions and assumptions are kept unaltered. Pre- and postconditions of a procedure are encoded as assumption at the beginning and assertions at the end of a procedure. Structured control-flow constructs are encoded by statements of the core language. For example, a **while** loop such as

while (G) **invariant** P { B }

where G is the loop condition, P is an invariant and B is the loop body, is encoded as follows [10] where x stands for the only variable modified in B:

```
Head:  assert P ; havoc x ; assume P ; goto Body, Done;
Body:  assume G ; B ; assert P ; goto;
Done:  assume ¬G ;
```

Operationally, these blocks can be understood as follows. The invariant is tested before the first loop iteration. The subsequent **havoc** statement simulates a state after arbitrary loop iterations. Then, either the loop condition does not hold and execution continues after the loop in the block labeled `Done` or the condition does still hold. In that case, the loop body is executed followed by checking that the loop invariant still holds afterwards which simulates one arbitrary iteration of the loop.

Second step Before generating a formula for a procedure in core language, Boogie applies three transformations [12]. First, the control-flow graph of the procedure is made acyclic by safely cutting loops similarly as shown above for **while** loops. Second, a single-assignment transformation is applied. Third, the result is turned into a passive procedure by changing assignment statements into **assume** statements. A machine-checked presentation of these transformations together with the final generation of a verification condition can be found in [158]. The result of these transformations is an unstructured, acyclic, passive procedure that consists of blocks each of which comprises only **assume** and **assert** statements followed by a **goto**. Note that such a procedure can be regarded as a directed acyclic graph, the *control-flow graph*, where **assume** and **assert** statements are nodes and where edges link statements to successor statements, and each **goto** statement is a notation for several successor statements.

Third step From an unstructured, acyclic, passive procedure, Boogie's final step generates a verification condition by means of weakest preconditions [62, 104]. For any statement S and predicate Q on the post-state of S, the weakest precondition of S with respect to Q, denoted by $\mathtt{wp}(S, Q)$, is a predicate that characterizes all pre-states of S

whose reachable successor states satisfy Q. The weakest precondition for Boogie statements is defined as follows:

$$\mathsf{wp}(\textbf{assert}\ P, Q) = P \wedge (P \longrightarrow Q)$$
$$\mathsf{wp}(\textbf{assume}\ P, Q) = P \longrightarrow Q$$

Note that in contrast to the standard definition, Boogie's weakest precondition for **assert** is slightly extended, but still equivalent. This gives the target prover Z3 the proposition P as a hint for proving Q independently of whether P can be shown to hold. Lifting weakest preconditions to a sequence of statements is as follows:

$$\mathsf{wp}(S_1 \ldots S_n, Q) = \mathsf{wp}(S_1, \ldots, \mathsf{wp}(S_n, Q))$$

For every Boogie block B comprising a sequence of statements S and successor blocks $B_1, \ldots, B_n$, the weakest precondition can be defined recursively as follows:

$$\mathsf{wp}(B, Q) = \mathsf{wp}(S, \mathsf{wp}(B_1, Q) \wedge \ldots \wedge \mathsf{wp}(B_n, Q))$$

Recall that the procedure's control flow is acyclic and hence the above function always terminates. The verification condition is $\mathsf{wp}(B_0, \top)$ where B_0 is the first block of the procedure.

4.2.4. Associating Errors to Source Code Locations

VCC can output source code locations of errors [106]. The underlying idea is to enrich each asserted property P with a vacuous constant L carrying the source code position for this assertion. This is done by a special labeling primitive $\mathsf{label}(L, P)$ that is logically equivalent to P. Due to the way a verification condition is generated, this decoration, or *label*, is maintained in the final formula given to Z3. The prover has special support for labels such that, when the corresponding asserted formula cannot be proved, Z3 emits the label in the countermodel produced. VCC can hence select the labels of all failing assertions and recalculate the original source code positions corresponding to them.

4.3. HOL-Boogie

HOL-Boogie [29, 31] is a bridge between VCC and Isabelle and consists therefore of two components. One component, a module of Boogie, writes verification conditions generated by Boogie into a file. The other component, a module of Isabelle, loads the verification conditions from this file, translates them from MSFOL to HOL and sets up an environment to let a user debug and prove them. Figure 4.3 gives an overview of this architecture. HOL-Boogie is a complete rewrite of an earlier prototype by Burkhart Wolff, from which we adopted some ideas, especially the described architecture. In contrast to that prototype, HOL-Boogie provides specific debugging tools and proof methods (Section 4.3.2) for Boogie-generated verification conditions.

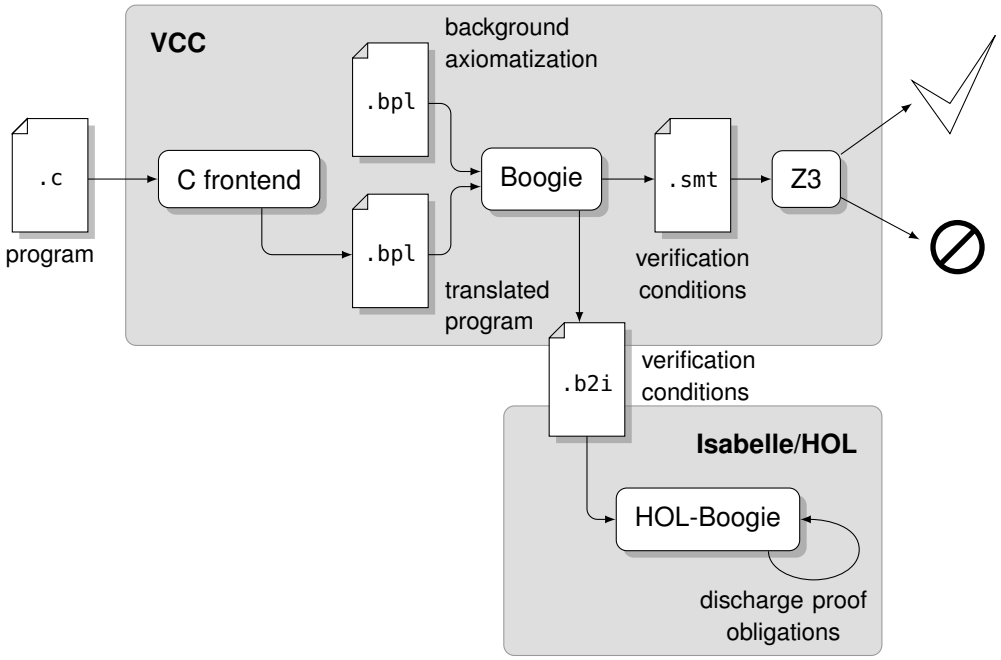


Figure 4.3. Overview of the integration of VCC with HOL-Boogie. The file ending `.b2i` is used for exchange files between Boogie and Isabelle.

There are two main incentives for a tool like HOL-Boogie. First, failing verification attempts can be debugged interactively by inspecting the background axiomatization as well as the exact parts of proof obligations that fail. This is in contrast to the terse error messages provided by VCC. Second, selected verification conditions can be proved in Isabelle/HOL using especially our integration of SMT solvers (Chapter 2) with proof reconstruction for Z3 (Chapter 3). That is, combining an automatic program verifier such as VCC with an interactive theorem prover such as Isabelle/HOL that has a rich specification language and a large number of proof tools can make infeasible verifications feasible. In addition, with our proof reconstruction (Section 3.3), soundness of the backend prover Z3 is not an issue.

Another application of HOL-Boogie is to formalize and prove correct the intricate background axiomatization of VCC consisting of about 450 axioms [29, 31]. As a consequence, verification conditions can be proved in Isabelle against this formalization instead of against VCC's axioms, thereby eliminating further soundness concerns of the VCC approach.

We continue by demonstrating how HOL-Boogie can be applied to debug annotations and to find proofs where VCC fails (Section 4.3.1). Thereafter, we explain the integration of HOL-Boogie with the VCC toolchain and Isabelle/HOL (Section 4.3.2). At the end, we indicate how verification performance can be improved substantially by changing the heap model underlying proof obligations from VCC (Section 4.3.3).

```
1  unsigned maximum(unsigned arr[], unsigned len)
2    _(requires \thread_local_array(arr, len))
3    _(requires len > 0)
4    _(ensures \result < len)
5    _(ensures \forall unsigned i;
6        i < len ==> arr[i] <= arr[\result])
7  {
8    unsigned max = 0, i;
9    for (i=0; i < len; i++)
10     _(invariant max < len)
11     _(invariant \forall unsigned j;
12         j < i ==> arr[i] <= arr[max])
13   {
14     if (arr[i] > arr[max]) max = i;
15   }
16   return max;
17 }
```

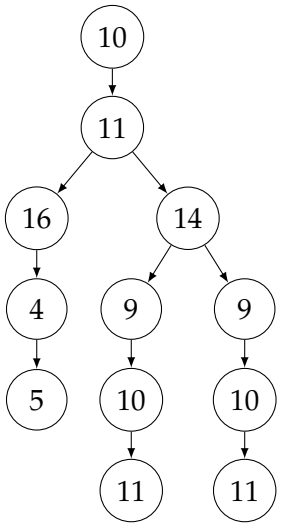


Figure 4.4. An annotated `maximum` function and the corresponding control-flow graph where assumptions are collapsed into subsequent assertions. The labels of the graph refer to line numbers in the source code.

4.3.1. Debugging and Proving Verification Conditions

While developing algorithms and their specifications such as the example given in Figure 4.4, it commonly happens that even if the program behaves as intended, its annotations are incomplete or inconsistent or they specify unintended behavior. Indeed, verifying the `maximum` function fails, and VCC reports the following error messages:

```
maximum.c(11,31): Loop body invariant '\forall unsigned j; j < i ==>
  arr[i] <= arr[max]' did not verify.
maximum.c(16,3): Postcondition '\forall unsigned i; i < len ==>
  arr[i] <= arr[\result]' did not verify.
maximum.c(5,27): (related information) Location of postcondition.
```

Without further information, it is hard to understand the reason for this error. One approach that relies on VCC only is guessing: The user gradually adds assertions with properties that are expected to hold. This way, the reason for the error is narrowed down, but finding the right assertions can be intricate. Essentially, this or similar approaches that involve fixing or tuning the specification such that they can finally be proved automatically resemble interactive proofs, but an interactive theorem prover is much more convenient.

With HOL-Boogie, the user can navigate to the cause for this error and inspect it. To this end, we exploit the labels (Section 4.2.4), which relate assertions to specific program points, and the ability to observe all assumptions of an assertion.

For navigating to errors HOL-Boogie provides two approaches. With the linear approach, HOL-Boogie sequentially applies a user-defined Isabelle method, typically our SMT integration, to each assertion in the order in which the assertions occur in the annotated C file. The first assertion that the method fails to prove is reported to the user. In our example and using Z3, we get the following message from HOL-Boogie:

```
failed on "L_11_31b"
```

That is, Z3 failed to prove the loop invariant on line 11. This coincides with one of the errors reported by VCC. Clearly, if a verification condition contains many assertions and the failing ones are among the last in the original file, the linear approach is rather inefficient. More efficient and also more comparable to the approach taken by VCC is the bisecting approach: HOL-Boogie first tries to discharge the entire verification condition. If that fails, HOL-Boogie splits the verification into smaller pieces and tries to prove the parts in parallel, followed by further splitting of failed parts until reaching individual assertions. With this approach we obtain the following result for our example within 20 seconds and after altogether 19 invocations of Z3 through our SMT integration:

```
Unsolved assertions of Boogie verification condition "maximum":
  L_11_31b
  L_11_31c
  L_5_27b
```

This list contains all assertions that also VCC considers unprovable, i.e., the loop invariant in line 11 and the postcondition in line 5. Note that the **`if`** statement in the loop body (Figure 4.4) gives rise to two possible successor states. Checking the loop invariant in line 11 for each of them is reflected by two assertions in the verification condition. VCC collapses the two induced error messages to one.

For inspecting the cause of failed proof attempts, HOL-Boogie provides means to extract assertions, referenced by label names such as `L_11_31b` above, from a verification condition. We focus here on the first failing assertion, which corresponds to checking the loop invariant of line 11 (Figure 4.4) after arbitrary iterations of the loop. In this case, Isabelle's simplifier leads us to uncover the flaw in our specification since it rewrites the assertion into the following proposition:

$$\mathsf{select}(\mathsf{memory}(s), \mathsf{idx}(arr, i+1)) = \mathsf{select}(\mathsf{memory}(s), \mathsf{idx}(arr, i))$$

This corresponds to `arr[i+1] == arr[i]` in C, and that equation is unexpected. It compares the array element visited last (index `i`) with the next and still unvisited array element (index `i+1`). In contrast, the loop invariant in line 11 (Figure 4.4) is supposed to compare only already visited elements. This extra information obtained from HOL-Boogie leads us to re-think the invariant. Indeed, the invariant uses the wrong index for the array, i.e., `i` instead of `j`. Unveiling annotation bugs such as in this case is one of the strengths of HOL-Boogie. In general, HOL-Boogie enables users to draw insufficient preconditions or invariants more systematically than by blindly testing in VCC.

That is, an interactive proof environment can be extremely helpful when debugging specifications. After HOL-Boogie has been developed, VCC was extended to produce counterexamples for failed proof attempts (Section 4.7) and hence also provides some help in debugging annotations.

After we have fixed the loop invariant by replacing `i` with `j` in line 12 (Figure 4.4), VCC succeeds in proving correct the `maximum` function. Alternatively, we can also prove correct the fixed version in HOL-Boogie to obtain the confidence of an LCF-style proof. Proofs of verification conditions in HOL-Boogie can either show an entire verification condition or discharge parts thereof in analogy to the bisecting approach for debugging verification conditions. Users of HOL-Boogie can do these proofs by applying any of the available Isabelle methods including our integration of SMT solvers or by writing structured proofs. It is worth noting that although Boogie-generated verification conditions are remarkably large, they can still be tackled with interactive theorem provers [31].

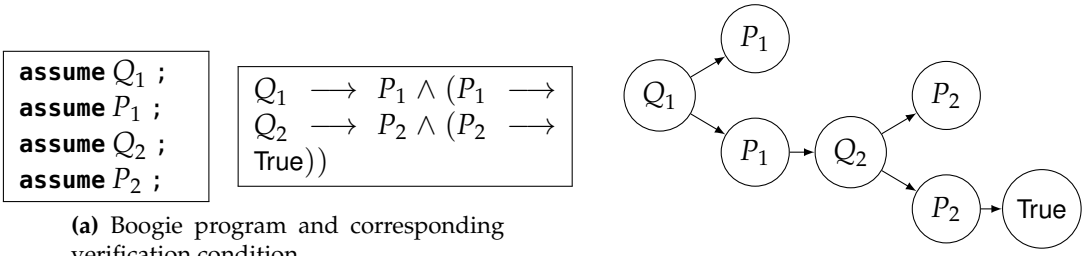
4.3.2. Integration with VCC and Isabelle/HOL

Boogie is able to pass verification conditions to different provers such as Z3, and the set of provers is extensible by a plugin mechanism. We use this mechanism for the first component of HOL-Boogie. More precisely, we implemented a pseudo-prover that, instead of proving verification conditions, writes all of them together with the entire background axiomatization unmodified into a file. Note that, since our pseudo-prover interfaces with Boogie, HOL-Boogie is not specific to C verification, but can in principle be used with any other program verifier that builds on Boogie, e.g., Dafny [105], Chalice [108] or Spec# [13].

HOL-Boogie's second component, which is a module of Isabelle/HOL, provides a special environment that comprises commands to load the files generated by our pseudo-prover as well as to inspect and prove the contained verification conditions. Central to this setup is a verification condition store that keeps track of which assertions have already been discharged. A verification condition is proved if every of its assertions is discharged. HOL-Boogie checks upon closing a verification environment that this is the case and informs the user about unfinished proof attempts. Hence, a HOL-Boogie environment can only be closed when all loaded verification conditions have been discharged. This is to ensure that everything that has been exported from Boogie is proved in Isabelle.

Note that users of HOL-Boogie prove only those verification conditions that have been imported to Isabelle. Consequently, users need to trust, in addition to VCC and Boogie, both our Boogie plugin to write Boogie-generated verification conditions to a file as well as the loader of such files in Isabelle, because these two components control the set of verification conditions that are available for proof in Isabelle.

Verification conditions are stored internally in an optimized form. Recall that unstructured, acyclic, passive Boogie procedures can be considered as directed acyclic control-flow graphs (Section 4.2.3). For example, Figure 4.4 gives a condensed version of the control-flow graph obtained from the `maximum` function. Figure 4.5 shows



(a) Boogie program and corresponding verification condition

(b) Control-flow graph

Figure 4.5. A Boogie program with corresponding verification condition and control-flow graph. Labels at assertions are omitted.

the verification condition and the complete control-flow graph for a simple Boogie program. Due to the way Boogie generates verification conditions, each control-flow graph (e.g., Figure 4.5b) is isomorphic to the corresponding procedure's verification condition (e.g., Figure 4.5a). Note that hence each graph has a designated root node corresponding to the first statement of the procedure. Internally, HOL-Boogie's Isabelle component stores verification conditions as control-flow graphs from which different kinds of proof obligations can easily be extracted. What we have called assertion so far is a proof obligation that is obtained from a path of the control-flow graph which starts at the root node and ends at a node that corresponds to a Boogie **`assert`** statement, and from which intermediate **`assert`** nodes have been removed. Removing nodes from a path is mirrored in the proof obligation by replacing corresponding formulas with True. For instance, here is the assertion P_1 for the Boogie program of Figure 4.5

$$Q_1 \longrightarrow P_1 \wedge (P_1 \longrightarrow Q_2 \longrightarrow \mathsf{True} \wedge (P_2 \longrightarrow \mathsf{True})) \tag{4.1}$$

which is equivalent to $Q_1 \longrightarrow P_1$, and this is the assertion P_2 of the same program

$$Q_1 \longrightarrow \mathsf{True} \wedge (P_1 \longrightarrow Q_2 \longrightarrow P_2 \wedge (P_2 \longrightarrow \mathsf{True})) \tag{4.2}$$

which is equivalent to $Q_1 \longrightarrow P_1 \longrightarrow Q_2 \longrightarrow P_2$. We are free to keep some of the removed **`assert`** nodes in the path and hence obtain a proof obligation composed of several assertions. Keeping all **`assert`** nodes in a path that starts at the root node and ends at a node without successors results in a proof obligation that we call *assertion path*. For example, the verification condition in Figure 4.5a is an assertion path since there are no branches in the corresponding Boogie program. In Figure 4.4, the path starting at the node labeled 10 and ending at the node labeled 5 is an assertion path. Each node labeled 10 corresponds to the invariant in line 10, and the node labeled 5 corresponds to the postcondition in line 5.

Based on this setup, we can now detail the bisecting approach to find failing assertions (Section 4.3.1). The user selects an Isabelle proof method, typically our integration of SMT solvers, that is applied to gradually shrinking parts of a verification condition with a short timeout. First, the entire verification condition is given to the selected proof

method. If it fails, HOL-Boogie splits the verification condition into its assertion paths and tries to prove them with the user-selected proof method.[1] Each failing path is split into two proof obligations by omitting the first or second half of the **`assert`** nodes. For instance, (4.1) and (4.2) are the two "halves" of an assertion path. Both parts of the assertion path are passed to the selected proof method. Splitting of proof obligations into "halves" is continued until they consist of only one **`assert`** node. This is, for example, already the case for both (4.1) and (4.2). Recall that each assertion has a label associated to it (Section 4.2.4). HOL-Boogie emits the labels of all failing assertions.

4.3.3. A Simpler Heap Model

Instead of proving the verification conditions generated by VCC, it is sometimes easier to prove variations of them in Isabelle/HOL as long as we can relate the proved propositions to the original verification conditions. We extended our verification condition store (Section 4.3.2) with a user-configurable transformation to modify HOL propositions and a set of rewrite rules to undo this transformation on HOL theorems. Whenever assertions are extracted from the store, the transformation is applied, and before a theorem resulting from a proved assertion is merged with previously proved assertions, it is rewritten by means of the rewrite rules. In the end, we compare the original verification condition with the final theorem. Thereby, we guarantee that what we proved in Isabelle/HOL is the same as what we exported from VCC.

We applied this technique to modify the heap model imposed by VCC's background axiomatization. The heap is a mapping from pointers to integer values. Pointers can either be atomic or composed of a pointer and an offset such as an array index or the field of a structure. Retrieving the base pointer or offset from a composed pointer depends on the program state in VCC since this retrieval is only defined for valid pointers, i.e., pointers to allocated memory, and VCC maintains a ghost mapping of valid pointers in the state. In a context where the set of valid pointers does not change, where only composed pointers occur and where all possible offsets are known statically, e.g., if only fields to structures are used as offsets, a much lighter heap model can be applied. This light heap model provides small heaps, or heaplets, for every field of a structure and uses the base pointer to index the stored integer values.

For example, assume that there are only two fields `f` and `g` and one heap update `p->f = v`. Translating this update to Boogie and exporting it to HOL-Boogie results in the following representation with explicit program states s and s':

$$\mathsf{heap}(s') = \mathsf{store}(\mathsf{heap}(s), \mathsf{dot}(p, f), v)$$

Under the assumption of heap extensionality, we are able to prove in Isabelle that this

[1] To reduce the runtime on multi-core machines, the assertion paths are given to several instances of the selected proof method in parallel.

is equivalent to

$$\forall q.\ \mathsf{F}(s',q) = (\mathsf{if}\ q = p\ \mathsf{then}\ v\ \mathsf{else}\ \mathsf{F}(s,q)) \tag{4.3}$$
$$\forall q.\ \mathsf{G}(s',q) = \mathsf{G}(s,q) \tag{4.4}$$
$$\forall q.\ q \neq \mathsf{dot}(p,f) \longrightarrow \mathsf{select}(\mathsf{heap}(s'),q) = \mathsf{select}(\mathsf{heap}(s),q) \tag{4.5}$$

where F and G are heaplets with the following Isabelle definitions:

$$\mathsf{F}(s,p) = \mathsf{select}(\mathsf{heap}(s),\mathsf{dot}(p,f)) \qquad \mathsf{G}(s,p) = \mathsf{select}(\mathsf{heap}(s),\mathsf{dot}(p,g))$$

That is, updating the global heap affects only the heaplet F (4.3) whereas other heaplets such as G stay unchanged. A subsequent heap lookup `r->f` for some pointer `r` needs to step only through the history of updates on F that are completely separated from those of, say, G. Compare this with a global heap where a heap lookup has to go through potentially all heap updates. Confining heap updates to smaller entities has a dramatic effect on overall verification time.

Our exposition of the simplified heap, i.e., essentially the properties (4.3), (4.4) and (4.5), is itself slightly simplified. The setup of VCC contains further technicalities related to the VCC memory model that we omitted here. Moreover, we note that property (4.5) is only required for completeness but not for reasoning about heap updates.

In HOL-Boogie, we were able to apply this technique to a larger case study (Section 4.4). That is, we were able to provide suitable transformations and according rewrite rules to manipulate verification conditions w.r.t. to heap accesses. From this example, we found that the resulting proof obligations could be proved about one order of magnitude faster than with VCC's native heap model. This is especially relevant if proving a single obligation takes more than a minute and frequent changes to related definitions require to re-prove this obligation regularly.

Following up on this experience and jointly with Michał Moskal [30], we showed that among different heap models the heaplets model described here also performs best for fully automatic program verification with VCC and Z3 as backend prover. Moreover, we found that the heap model currently underlying VCC scales particularly poorly in the number of heap updates, and much better alternatives exist. One of the more efficient alternatives models the heap as a (curried) mapping from fields and addresses to values, i.e., as a function of type $\mathit{field} \to \mathit{address} \to \mathit{integer}$. Note that such a heap fits to VCC's methodology as VCC already requires pointers, i.e., unique identifiers to objects, to be formed by an address and a field (Section 4.2.2). Note further that when all relevant fields are statically known, this model is close to the above heaplets model and thus shows similar, yet slightly worse performance. The sketched heap model will be the default one in future versions of VCC as the result of our work.

4.4. Case Study: Binary Search Trees

A *binary search tree* [50, chapter 13] is a data structure based on a binary tree to store a set of keys and associated elements in an ordered way by maintaining the following *binary-search-tree property*:

> Every node of the binary search tree carries a key, and for every node x of a binary search tree, if y is a node in the left subtree of x, then the key associated with y is less than that of x, and if y is a node in the right subtree of x, then the key of y is greater than that of x.

A binary search tree supports operations such as looking up minimal, maximal and arbitrary elements as well as inserting and removing elements. The worst-case time complexity of these operations depends on the height of the binary tree, and hence keeping the tree "balanced" yields the optimal worst-case behavior of $O(\log n)$ for all of them. Whereas looking up elements only traverses the binary tree without modifying it, naively inserting or removing elements without rebalancing the tree may impair its structure in such a way that the worst-case behavior of all search tree operations degenerates to $O(n)$.

Optimal performance of binary-search-tree operations can be guaranteed when extending the binary tree to a *red-black tree* [50, chapter 14] and maintaining the red-black property. Lookup functions for a red-black tree are the same as those for a binary search tree. Only the implementation of functions that modify the tree, i.e., insertion and removal of elements, require changes. Compared to their counterparts for general binary search trees those modification functions have to be extended by a follow-up step to rebalance the tree and hence recover the red-black property.

Implementations of red-black trees are ubiquitous. Especially their occurrences in security-sensitive applications and operating system kernels are worthwhile candidates for formal verification. There are, for instance, implementations of red-black trees in the kernels of Linux and OpenBSD. Also the Microsoft Hyper-V hypervisor [103, 122] uses red-black trees internally.

As a first step towards full functional verification of a red-black tree implementation, we focus here on binary search trees and attempt to verify two typical functions: looking up existing elements (`lookup`) and inserting new elements (`insert`). We consider C code (Figure 4.6) that is similar to the one used within Hyper-V and incidentally also to that in [50, chapter 13]. Our main interest lies in demonstrating and evaluating HOL-Boogie, and to this end we describe two attempts in verifying the binary search tree functions. Our first attempt separates low-level reasoning that is handled by VCC from proving intricate properties in Isabelle (Section 4.4.2). The way in which we formalized these properties resulted unfortunately in overly complicated Isabelle proofs. We hence report on a second attempt (Section 4.4.3) where we raise the level of abstraction and, to our surprise, are able to establish all properties in VCC. Both verification attempts have an abstract specification of binary search trees in common (Section 4.4.1) that we describe first.

4.4.1. Abstract Specification of Binary Search Trees

A binary search tree is an implementation for a partial mapping N from keys, in our case integers, to certain elements. For simplicity, we take here the addresses of the tree's

```
struct T_Node {
  int key;
  P_Node left, right;
};
```

```
struct T_Tree {
  Node nil;
  P_Node root;
};
```

```
P_Node lookup(P_Tree t, int k) {
  P_Node p = t->root;
  while (p != &t->nil) {
    switch (cmp(k, p->key)) {
      case Equal: return p;
      case Less:
        p = p->left;
        break;
      case Greater:
        p = p->right;
        break;
      default: return Null;
    }
  }
  return Null;
}
```

```
int insert(P_Tree t, int k, P_Node n) {
  Keymatch match = Equal;
  P_Node x = tree->root;
  P_Node y = &tree->nil;
  while (x != &t->nil) {
    y = x;
    match = cmp(k, x->key);
    switch (match) {
      case Less:
        x = x->left;
        break;
      case Greater:
        x = x->right;
        break;
      case Equal:
      default:
        return 0;
    }
  }
  n->key = key;
  if (y == &t->nil) t->root = n;
  else if (match == Less) y->left = n;
  else y->right = node;
  return 1;
}
```

Figure 4.6. Extract from a C implementation of binary search trees with the functions `lookup` and `insert`

nodes to be these elements.[2] Based on N, we formulate the following postconditions. Looking up a node from the tree with a key k should give the same result as N applied to k. Inserting a node n for a key k should do nothing if N maps k already to some node and otherwise should perform an update such that a subsequent lookup with k results in n and any other lookup is as before.

From VCC's ownership model we get for free the set of all nodes stored in the tree. Since every node carries its key, this set can indirectly act as the domain of the mapping N. It is straightforward to specify that the root is owned by the tree and that every child of an owned node is owned as well. VCC needs this information for establishing pointer validity when verifying our functions that traverse the tree. We guide VCC by providing loop invariants expressing that at any time all pointers under consideration address only nodes of the tree.

The mapping N and the set of owned nodes together already suffice to prove that

[2] In fact, this is a common idiom in C, where operations of a data structure never allocate memory themselves. Instead, a user of the data structure provides pointers to the required chunks of memory.

looking up the node for a key always returns a node owned by the tree if the key is in the domain of N. We are also able to verify that `insert` either increases the set of owned nodes by the added node or keeps it unmodified. Yet, we are interested in the much stronger postconditions given above for which we need a specification of the binary-search-tree property. It requires a formalization of transitive *reachability* to express that one node is in the subtree of another node. This is necessary to state the *sortedness* of subtrees, i.e., that the left subtree of a node contains only elements with smaller key and that the right subtree contains only elements with greater key. Finding a suitable specification for these two properties forms the heart of the following two sections.

4.4.2. First Attempt: Combining Automatic and Interactive Proofs

It seems reasonable to let VCC handle low-level memory-related properties and the simple abstraction of binary search trees (Section 4.4.1) and to verify functional correctness in Isabelle using HOL-Boogie. With recursive functions at hand, reachability and sortedness are simple to define in Isabelle. Likewise, inductive proofs for properties about these functions are straightforward. One would guess that full functional verification of binary search trees can be accomplished easily, but the devil is in the details.

The main obstacle is that the Isabelle definitions and related theorems have to be linked to the binary-search-tree implementation and its VCC specification. Our solution is as follows. On the VCC side, we define a ghost predicate `sub` without any contract that is meant to express the reachability between two nodes. Using this predicate, we add special invariants to the binary-search-tree structure to maintain that all nodes are transitively reachable from the tree's root if and only if they are owned by the tree and that the tree has the sortedness property. These invariants are ignored when verifying the implementation with Z3 but are passed to HOL-Boogie.[3] Similarly, we add special assertions that imply the desired postconditions or intermediate steps. These assertions are turned into assumptions when verifying with Z3 but are considered as proof obligations for HOL-Boogie. There, we define two binary relations left and right that relate each node with its left and right child. The reflexive transitive closure of their union is taken as the axiomatic definition for the sub relation in Isabelle. Based on this setup, we are able to derive facts that are required for establishing the various proof obligations of the binary-search-tree specification.

The unfinished proof development in Isabelle spans about 1000 lines of definitions and proof text. We succeeded in verifying the `lookup` function unlike for `insert`. There, we failed to show that the relevant tree invariants still hold at the end of the function. We see the following three reasons. First, our definitions in Isabelle are ultimately based on the VCC memory model that thus clutters many of our proofs. Despite several abbreviations, nearly all propositions that we stated explicitly are overly complex due to these roots in VCC. In retrospect, finding a suitably abstract representation might have reduced this complexity. Second, proof obligations tend to be huge. Several hundred assumptions to a single assertion are the regular case. Most proof methods of Isabelle

[3] We use a C preprocessor switch to make this possible.

are of little help. Only *sledgehammer* assisted in finding relevant axioms from the VCC background axiomatization. In the end, most of our proofs use the SMT integration to invoke Z3. Third, since we annotated the code gradually, our labels changed with every iteration. Since fixing them is laborious, we implemented crude code to spare us from this task. More importantly, changes to annotations also influenced our already existing and partly structured proofs. We conclude that HOL-Boogie is not apt to proving complex verification conditions that have a low level of abstraction.

4.4.3. Second Attempt: Purely Automatic Proofs

The previous attempt failed partly due to having formalized reachability and sortedness in Isabelle. To overcome this deficiency, we tried to specify these properties directly in VCC and, to our surprise, found annotations that are sufficient to verify the binary-search-tree implementation entirely in VCC (Appendix B).

Our basic ideas are as follows. To model reflexive, transitive reachability, we added to the binary-search-tree structure a ghost field that maps each key stored in the tree to the set of keys in its subtree. Observe that each key is stored at most once in the tree, and that there is a bijection between nodes and keys (Section 4.4.1). We decided to use keys instead of nodes because this prevents VCC from adding assertions to check for validity of pointers. Thus, proof obligations get lighter and more abstract. The specification for the reachability ghost field, given as invariants to the tree structure, essentially expresses that every node is in its own subtree and that every node is reachable from the tree's root node. The sortedness property is specified in a way that fits the tree traversals in `lookup` and `insert`. More precisely, for every node n_2 that is in the subtree of a node n_1 and whose key is less than that of n_1, it is implied that n_2 must be in the subtree of the left child of n_1. The other case is analog. With this specification, automatically verifying `lookup` with VCC requires only one further loop invariant in addition to the annotations induced by our abstract specification (Section 4.4.1). In contrast, more annotations are necessary for `insert`. We were able to verify this function with VCC under the assumption that reachability and sortedness can be re-established after inserting a node.

We used HOL-Boogie only as debugger, in analogy to what we demonstrated earlier (Section 4.3.1), whenever particular VCC proof attempts failed. In contrast to our first attempt (Section 4.4.2) on which we spent more than a week and where we developed a supporting Isabelle theory of abbreviations and lemmas, it took us only a day with this second attempt to reach a similar state w.r.t. the verified properties (Appendix B). Hence, choosing the right level of abstraction can make a huge difference.

4.5. Abstract Cooperation

VCC is good at discharging low-level properties w.r.t. memory, concurrency as well as pointer and linear arithmetic and also able to deal with complex specifications (Section 4.4.3). Yet, despite the specification mechanisms provided by VCC and despite the

ability of its backend prover Z3 to prove complicated problems, there are properties of C programs which are more conveniently formalized and especially more conveniently proved in an interactive, higher-order theorem prover such as Isabelle. For instance, properties from the domains of graph theory fall in this category as do properties for which Isabelle already provides a well-developed theory with many theorems.

When devising a new combination approach between VCC and Isabelle, it seems appropriate to exchange only suitably abstracted representations between VCC and Isabelle and confine reasoning related to VCC's methodology to VCC considering the lessons learned from our binary-search-tree verification (Section 4.4). Our approach, jointly developed with Eyad Alkassar, Kurt Mehlhorn and Christine Rizkallah [2], is as follows. Assume that some C code fulfills a predicate $P(x)$ which depends on the program variable x, and we want to establish that also some predicate $Q(x)$ holds. Assume further that the *concrete implication*

$$P(x) \longrightarrow Q(x) \tag{4.6}$$

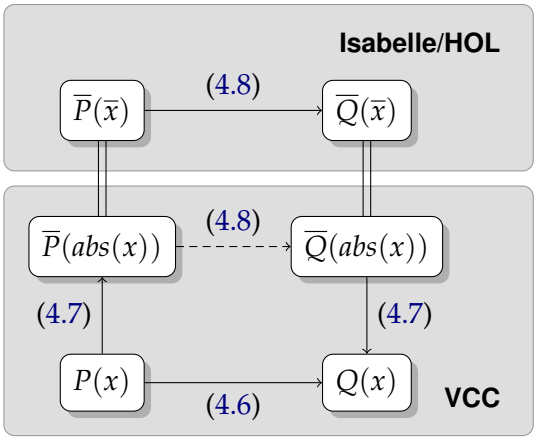
that depends on the program state is hard to prove in VCC, but proving a suitably abstracted version of it in Isabelle/HOL is possible. We thus define in VCC an *abstraction function abs* that maps from the type of the program variable x to a logical type such as the mathematical integers, a map or a tuple. We use $\overline{x}$ to denote elements of the logical type. Furthermore, we define in VCC *abstract predicates* $\overline{P}(\overline{x})$ and $\overline{Q}(\overline{x})$ that are, since they are defined over logical values $\overline{x}$ only, independent from the VCC memory model. For these predicates, we prove in VCC the *correspondence implications*

$$P(x) \longrightarrow \overline{P}(abs(x)) \qquad \text{and} \qquad \overline{Q}(abs(x)) \longrightarrow Q(x) \tag{4.7}$$

After additionally proving the *abstract implication*

$$\overline{P}(\overline{x}) \longrightarrow \overline{Q}(\overline{x}) \tag{4.8}$$

for arbitrary abstract values $\overline{x}$ in Isabelle/HOL, we finally conclude (4.6). The following commuting diagram summarizes our approach:



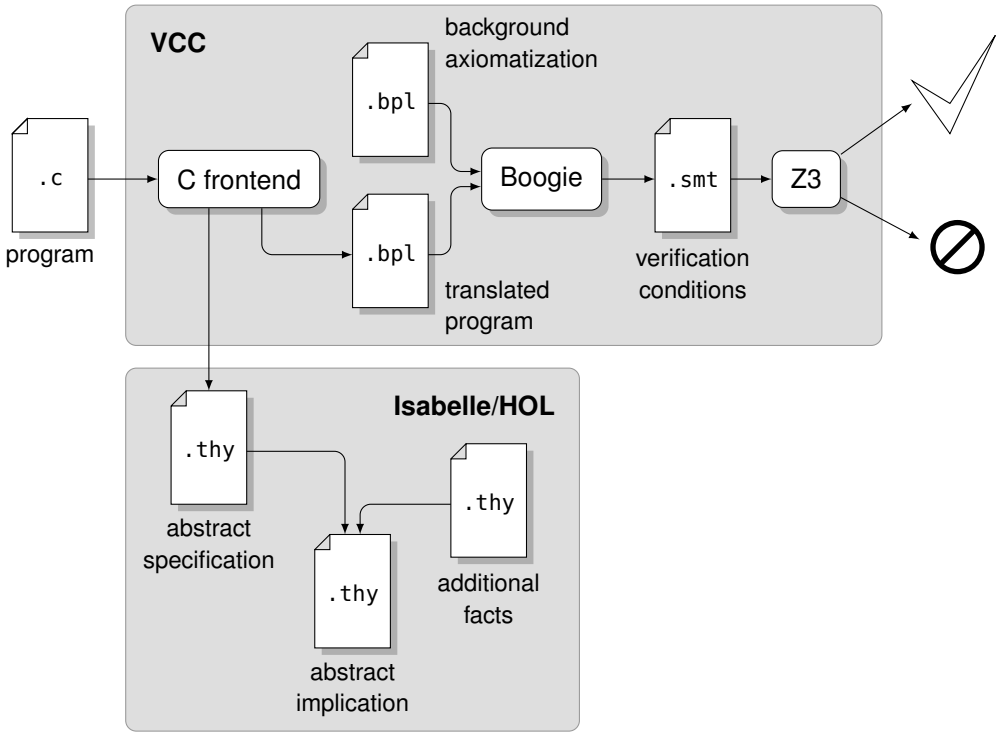


Figure 4.7. Overview of the combination between VCC and Isabelle/HOL

Solid arrows indicate implications that we prove either in VCC or in Isabelle. Since we prove implication (4.8) in Isabelle, it is sound to declare it as an axiom in VCC, denoted by a dashed arrow, assuming that the translation from VCC definitions of $\overline{P}(\overline{x})$ and $\overline{Q}(\overline{x})$ to corresponding ones in Isabelle/HOL, denoted by double lines, is equivalence-preserving. In fact, this translation is a straightforward syntactic rewriting of VCC notions into Isabelle ones. That is, the VCC types **`bool`** and **`\integer`** are mapped to the HOL types bool and int, VCC map types are translated to HOL function types, and VCC tuples are represented as Isabelle records [130]. VCC expressions, i.e., MSFOL formulas and terms, involving logical connectives, quantifiers, integer arithmetic operations and specification functions are mapped to corresponding HOL constructs. Definitions of VCC ghost functions are translated to Isabelle functions [96]. We extended the C frontend of VCC (Figure 4.7) to generate an Isabelle theory, the *abstract specification*, containing all logical declarations of an annotated C file which carry a special `isabelle` attribute. In Isabelle, we then establish the abstract implication by referring to those exported declarations and additional facts from Isabelle.

The tool support is limited to translating VCC specifications into Isabelle functions. This is the only component that needs to be trusted with this approach besides VCC, that is used for proving low-level program properties, and Isabelle, that is employed for establishing the abstract implication. Finding suitable abstract predicates and abstraction functions are in the responsibility of the user, and so is formulating and proving

```
1  int mult(int a, int b)
2    _(requires a >= 0 && b >= 0 && a * b < INT_MAX)
3    _(returns a * b)
4  {
5    int ret = 0;
6    int i;
7    for (i=0; i < b; i++)
8      _(invariant i >= 0 && i <= b)
9      _(invariant ret == a * i)
10   {
11     ret += a;
12   }
13   return ret;
14 }
```

Figure 4.8. Annotated C implementation of a multiplication function

the correspondence lemmas. It is convenient to prove these lemmas in VCC since they relate VCC's methodology to an abstract specification, but other approaches such as HOL-Boogie might be applied as well.

To illustrate the cooperation between VCC and Isabelle, we consider an implementation of a multiplication function (Figure 4.8) from the VCC discussion panel [112]. For this example, it takes VCC more than one minute to find out that the addition in line 11 might overflow. More precisely, VCC fails in proving the arithmetic implication

$$a \cdot b < c \ \wedge \ a \geq 0 \ \wedge \ i < b \ \longrightarrow \ a \cdot i + a < c \tag{4.9}$$

for arbitrary (mathematical) integers a, b, c and i. It is possible to prove this proposition as a lemma in VCC using a special option that makes VCC introduce uninterpreted proxy functions for addition and multiplication and that hence exploits the combination of decision procedures in Z3. We refrain from going into details here in favor of demonstrating our approach that proceeds with the following steps:

- Define an abstraction function for machine integers:

  ```
  _(ghost _(pure) \integer abs(int i) _(returns i))
  ```

- Define abstract predicates for the assumption and conclusion of the failing implication (4.9) and tag them as part of the abstract specification:

  ```
  _(ghost vcc_attr("isabelle", "") _(pure)
    bool abs_P(\integer x, \integer y, \integer z)
    _(returns x * y < INT_MAX && x >= 0 && z < y))

  _(ghost vcc_attr("isabelle", "") _(pure)
    bool abs_Q(\integer x, \integer z)
    _(returns x * z + x < INT_MAX))
  ```

- Define the abstract implication, tag it to be exported as part of the abstract specification and declare it as an axiom:

```
_(ghost vcc_attr("isabelle", "") _(pure)
  bool abstr_impl(\integer x, \integer y, \integer z)
  _(returns abs_P(x, y, z) ==> abs_Q(x, z)))

_(axiom \forall \integer x, y, z; abstr_impl(x, y, z))
```

- Add the following assertion between line 11 and line 12 of the multiplication function to make VCC consider the abstract implication:

```
_(assert abstr_impl(abs(a), abs(b), abs(i)))
```

VCC succeeds in verifying the modified multiplication function within less than a second. It remains to prove the correspondence lemmas and the abstract implication.

Although not strictly necessary, it is convenient to formulate the correspondence lemmas in VCC's specification language and to let VCC prove them automatically. We write each lemma as a ghost function. The lemma's assumptions are encoded as preconditions, and the conclusion is expressed by a postcondition:

```
_(ghost _(pure) void lemma_to_abs_P(int a, int b, int i)
  _(requires a * b < INT_MAX && a >= 0 && i < b)
  _(ensures abs_P(abs(a), abs(b), abs(i)))
  { })

_(ghost _(pure) void lemma_from_abs_Q(int a, int i)
  _(requires abs_Q(abs(a), abs(i)))
  _(ensures a * i + a < INT_MAX)
  { })
```

VCC verifies these lemmas within a second. Alternatively but less conveniently, we could have proved them using HOL-Boogie (Section 4.3), for instance.

Finally, we export the abstract specification to Isabelle, i.e., we let VCC translate all ghost functions with an `isabelle` attribute into Isabelle functions. The proof of the abstract implication in Isabelle is straightforward, and we only sketch the central part of the proof here. From $z < y$, which is equivalent to $z + 1 \leq y$, and $x \geq 0$ we can deduce $x \cdot (z + 1) \leq x \cdot y$ by monotonicity of multiplication, and from this together with $x \cdot y < c$ we conclude $x \cdot z + x < c$ by distributivity and transitivity.

4.6. Case Study: Maximum Cardinality Matching in Graphs

Despite recent progress, e.g., as demonstrated in the seL4 project [95], formal verification of highly complex algorithms remains challenging. Instead of verifying such algorithms, it is much simpler to verify that checking each of the algorithms' results is correct. For example, formally verifying an SMT solver such as veriT is currently out of reach, but formally verifying a checker for proof certificates produced by veriT is realistic and has been done already [4]. Indeed, it is known that several complex algorithms

are able to produce, along with the desired result, a certificate that allows for easy and efficient checking of the result's correctness [25,114,150]. Since a program that performs this checking, a *checker*, is much simpler than the complex algorithm, a checker is much better suited for current formal verification technology. Yet, the obtained property w.r.t. the complex algorithm is also much weaker: We only obtain correctness of individual results rather than correctness of the entire algorithm.

Graph algorithms such as those in LEDA [117], the library of efficient data structures and algorithms, belong into the class of complex algorithms that can produce certificates. Checkers for these algorithms are reasonable simple and hence meet our above criteria. We consider here one of the more challenging algorithms, *maximum cardinality matching* (MCM) in graphs [117, section 7.7], and describe the formal verification of the corresponding checker which is joint work with Eyad Alkassar, Kurt Mehlhorn and Christine Rizkallah [2].

A *matching* M of a graph G is a subset of the edges of G such that no two edges are incident to the same node. The cardinality $|M|$ of a matching M is the number of edges of M. A matching has maximum cardinality if its cardinality is at least as large as that of any other matching. An *odd-set cover* of a graph G is a labeling of the nodes of G such that every edge of G is either incident to a node labeled 1 or connects two nodes labeled with the same number greater than 1. For a given graph G, the MCM algorithm computes a maximum cardinality matching M and, as the certificate, an odd-set cover *osc*. The following theorem provides the foundation for the MCM checker.

Theorem 4.1 (Edmonds [65])**.** *Let M be a matching in a graph G with nodes set N and let osc be an odd-set cover for G. For any $i \geq 0$, let $n_i = |\{n \in N \mid osc(n) = i\}|$ be the number of nodes labeled i. If the equality*

$$|M| = n_1 + \sum_{i \geq 2} \left\lfloor \frac{n_i}{2} \right\rfloor \tag{4.10}$$

holds, then M is a maximum cardinality matching for G.

Hence, it is sufficient for the MCM checker to test that the cardinality of the matching M fulfills (4.10). Then, by the above theorem, which has been proved by Christine Rizkallah [2,144] in Isabelle/HOL, the desired property follows.

We continue in detailing the C data structures of the MCM checker (Section 4.6.1) and the specification for the checker (Section 4.6.2), but omit the C implementation which is straightforward. Thereafter, we show how to relate the proof of Theorem 4.1 to the checker specification based on the cooperation approach described in the previous section (Section 4.6.3). Everything has been formally verified in VCC except for the linking proofs that link Theorem 4.1 with the abstract VCC specification and that we did in Isabelle/HOL.

Instead of showing code or annotations of the checker or proof scripts of Isabelle, we present this case study on a high level. More details can be found in [2].

4.6.1. Data Structures of the MCM Checker

The checker is based on the following data structures. Assume that a graph G consists of n_G nodes and m_G edges. Hence, nodes and edges can be enumerated. We use numbers from 0 to $n_G - 1$ to identify nodes. Edges are C structures with two unsigned integer components, one for each connected node. A graph is a C structure with an array of m_G edge structures, indexed from 0 to $m_G - 1$, as well as two unsigned integer fields carrying the values of n_G and m_G. A matching M is a graph as well and thus represented by the same graph structure. Since the number of edges of M is typically less than that of G and thus the edge arrays of both graphs differ, the MCM algorithm computes, together with M, a mapping f that is represented by an array of length m_M and that relates an edge index of M to one of G. Finally, an odd-set cover *osc* is an array of n_G unsigned integers, i.e., the labels assigned to the node identifiers of G.

4.6.2. Specification for the MCM Checker

We specify with a structure invariant that a graph has no self-loops, i.e., no edge connects a node to the same node, and that edges connect only nodes of the graph, i.e., node numbers are in range. The exact values of labels that are greater than 2 are irrelevant for (4.10), but for practical purposes these values are always less than n_G. We take this restriction as an invariant on *osc*.

Given a graph G and the computed matching M, both of which are required to be wrapped, as well as the certificate consisting of the odd-set cover *osc* and the edge mapping f, the checker must return `true` if and only if all following properties Π hold:

- M is a subgraph of G w.r.t. f, i.e., for every edge index e that refers to an edge of M the index $f(e)$ refers to the same edge in G irrespective of the order in which the connected nodes are mentioned in the edges.
- M is a matching, i.e., no two edges of M have one node in common.
- *osc* is an odd-set cover for G, i.e., label values are less than n_G and for every edge of G either one of its connected nodes is labeled 1 or both nodes are labeled with the same label whose value is greater than 1.
- (4.10) holds, i.e., the number of edges in M coincides with a particular sum s of node counts n_i. Such a sum is best expressed by a recursive function, but so far VCC does not support termination proofs. Instead, we define a finite sequence, modeled by a map type, of partial sums s_i such that s_{n_G} equals s. To this end, we define $s_0 = 0$ and $s_1 = 0$ for trivial graphs and let s_2 be equal to n_1, i.e., the number of all nodes labeled 1. For every $i > 1$, we define s_{i+1} as the sum $s_i + \lfloor \frac{n_i}{2} \rfloor$. As part of the specification, we implemented ghost code to compute these sums. The node counts n_i are defined similarly by sequences of partial sums.

Verifying the MCM checker against this specification requires heavy annotations that increase the size of the code roughly by a factor of 3. Most of the effort goes into finding suitable loop invariants. There is one assumption left, though, that postulates the

existence of enough unallocated memory. Alternatively, we could have changed the postcondition of the checker such that it returns `false` also when it fails to allocate enough memory.

When verifying code it is always pleasant to discover a bug in the implementation. Indeed, we revealed that the MCM checker of LEDA neglects to test whether M is a subgraph of G.

4.6.3. Abstracting the Checker Specification and Combining the Results

The above specification only verifies that the MCM checker tests (4.10) correctly, but what we are eventually interested in is whether the computed matching M is indeed the maximum cardinality matching. This requires to link the above VCC specification with the Isabelle proof of Theorem 4.1, and we do this by instantiating the cooperation approach (Section 4.5). The relationship is as follows:

$$\begin{array}{rcl} x & \hat{=} & \text{graph } G\text{, matching } M\text{, odd-set cover } \mathit{osc} \text{ and edge mapping } f \\ P(x) & \hat{=} & \text{the properties } \Pi \text{ hold} \\ Q(x) & \hat{=} & M \text{ is a maximum cardinality matching, i.e., the number of edges} \\ & & \text{in } M \text{ is greater or equal to that of any other matching } M' \text{ for } G \end{array}$$

It thus remains for us to provide abstract representations of the MCM data structures, corresponding abstraction functions, abstract formulations for the properties Π as well as for the conclusion that M has maximal cardinality, and suitable correspondence proofs between abstract and concrete properties.

We represent abstract nodes by mathematical integers, abstract edges by pairs of abstract nodes and graphs by triples consisting of the number of nodes and edges of the graph as well as a map from mathematical integers, that abstract the edge identifiers, to abstract edges. The abstract representation is intentionally close to the concrete MCM data structures to simplify the definitions of the necessary abstraction functions and the abstract predicates and to reduce the effort for the correspondence proofs. Indeed, obtaining the abstract counterparts of concrete definitions is little more than syntactical rewriting. The main difference between the concrete and the abstract level comes from extending the range of indices from unsigned integers to mathematical integers. Simple additional guards in the definitions of the abstract predicates suffice to resolve this issue. The correspondence proofs are straightforward and require only a few hints to guide VCC's backend prover Z3.

In Isabelle/HOL, Theorem 4.1 is formalized on a higher level of abstraction than our abstract specification in VCC. In particular, nodes are arbitrary natural numbers, edges are modeled as sets of two distinct nodes, and graphs are sets of edges. Hence, a matching is simply a subset of a graph. An odd-set cover is a mapping from nodes to natural numbers, and edge mappings are unnecessary. These representations are suitably abstract to conveniently prove Theorem 4.1.

Bridging the gap between the low-level specification exported from VCC and the high-level formalization in Isabelle requires a number of mostly straightforward linking proofs. Relating the partial sums from VCC to recursive definitions in Isabelle is more

involved and requires two lengthy inductive arguments. Altogether, the linking proofs span 400 lines of Isabelle proof text. This effort is spent well in comparison to the alternative: Proving Theorem 4.1 based on the types and functions exported from VCC would be less direct since their specific representation complicates reasoning. From our experience with the binary-search-tree case study (Section 4.4.2), we estimate that without abstraction, the proof of Theorem 4.1 would require much more work than the combination of proofs that we describe above.

In contrast to HOL-Boogie and our binary-search-tree verification attempt (Section 4.4), we found that the abstract cooperation approach between VCC and Isabelle worked very well for this case study, and we expect that similar verification attempts can successfully build on this methodology.

4.7. Related Work

Our choice of VCC as verification platform is motivated by the Verisoft XT project (`http://www.verisoftxt.de`) where it was successfully used to verify tens of thousands lines of low-level C code. Other automatic program verifiers exist. C code can, for instance, be verified by Frama-C [124] and VeriFast [90]. For object-oriented languages, due to simpler language models, more code verifiers have been constructed such as Spec# [13], Chalice [108] and Dafny [105]. Especially Java has attracted research and thus Java programs can, for example, be verified by tools such as Jahob [99, 167], VeriFast [90] and ESC/Java [70]. SPARK [9], a subset of Ada specifically designed to implement high-integrity software, also includes verification tools. In addition, Jackson [89] integrated SMT solvers as backend provers for SPARK.

As an alternative to using an automatic program verifier, imperative programs can also be verified directly in interactive theorem provers such as Isabelle/HOL. This requires a formalization of C [131] or a fragment thereof [102] and a suitable verification condition generator [146]. Higher-order logic allows for clear, succinct specifications, but the low degree of automation specific to such verification tasks requires considerably more proof effort in contrast to our combinations with VCC [29, 134].

Combining automatic program verifiers with interactive theorem provers HOL-Boogie shares ideas with Why [67, 68] that is, much like Boogie, an intermediate program verifier with several frontends such as Frama-C [124] and Ada [80] and automated as well as interactive backend provers including Z3 and Isabelle. Verification conditions for which automated provers fail can be given to interactive theorem provers. In contrast to HOL-Boogie, though, sophisticated debugging features based on positional labels are not provided by the Isabelle integration of Why. Close in spirit to Why but targeting Java are Jahob [99, 167] and ESC/Java [91] both of which can pass failed proof attempts to Isabelle. Similar to Why, they give the interactive theorem prover the same information that is made available to all other automatic backend provers and thus overwhelm the user of an interactive theorem prover with a mass of details. This is in

contrast to our abstract cooperation that only exchanges suitably abstracted definitions between VCC and Isabelle.

The main incentive for HOL-Boogie, i.e., debugging failed proof attempts, is since recently also tackled by the Boogie Verification Debugger [79] and to some extent also by earlier debugging facilities provided with VCC. These tools try to reflect information obtained from Z3 such as counterexamples and partial counterexample traces and thus partly supersede HOL-Boogie. Many of the ideas behind HOL-Boogie found their way into HOL-SPARK [19], an environment for proving verification conditions from SPARK [9] programs.

Formal verification of binary search trees Moskal [107] has accomplished functional verification of a red-black-tree implementation w.r.t. the binary-search-tree property and entirely within VCC using carefully chosen triggers and related techniques to guide Z3's quantifier heuristics (Section 1.4.2). There are also Java implementations of binary search trees that have been verified against full functional specifications, for example using Jahob [167] or VeriFast [90]. Full functional verification of a red-black-tree implementation is, to the best of our knowledge, still an open problem.

Verified checkers The idea of letting algorithms produce a certificate to simplify the checking of results goes back to at least al-Khawarizmi who describes how to partially check the correctness of multiplication. In contrast, verified checkers appeared only recently. Probably one of the first formally verified checkers is for a sorting algorithm [38] that has been formalized in the Boyer-Moore theorem prover. Bulwahn et al. [40] describe a SAT checker, i.e., a checker for certificates of unsatisfiability produced by a SAT solver, verified in Isabelle/HOL. Similarly, Darbari et al. [53] as well as Armand et al. [5] formalized a SAT checker in Coq, and Armand et al. [4] have subsequently extended their work to SMT checkers. CeTA [154], a tool for certified termination analysis, is also based on formally verified checkers, again formalized in Isabelle/HOL.

Chapter 5.

Conclusion

Contents

5.1. Proof Automation

We integrated SMT solvers with the interactive theorem prover Isabelle. Our integration consists of a sound translation from Isabelle's higher-order logic to many-sorted first-order logic understood by SMT solvers (Section 2.2) and efficient proof reconstruction for proofs found by Z3 (Section 3.3). With the help of extensive evaluations (Section 2.5 and Section 3.4), we showed that SMT solvers, in particular Z3, can automatically and almost instantaneously find proofs where other proof methods of Isabelle fail. Moreover, these evaluations provide evidence that our integration is robust and applicable to a wide range of conjectures. Our contributions to the interactive theorem prover Isabelle are summarized as follows:

- A new proof method, called *smt*, that invokes one of the external SMT solvers CVC3, Yices or Z3. When Z3 is applied, the *smt* method can deliver LCF-style proofs. This new proof method is a decision procedure for unquantified first-order logic with equality and linear arithmetic. In addition, many first-order and also mildly higher-order problems can be proved by this method.

- An extension of *sledgehammer* to invoke SMT solvers for automatically finding proofs from a large collection of Isabelle facts. SMT solvers complement further external automatic theorem provers.

The *smt* method has been available in Isabelle since 2009. The integration of SMT solvers with *sledgehammer* was added one year later, which was joint work with Jasmin Christian Blanchette. Since then, SMT solvers have frequently helped in proof developments of Isabelle users. For example, a formalization of algebras [81] greatly benefited from the proof automation provided by our integration of SMT solvers in Isabelle.

5.2. Program Verification

We developed a tool, HOL-Boogie (Section 4.3), and a methodology, abstract cooperation (Section 4.5), to prove properties about C programs when automatic tools fail. Both approaches are based on extensions of VCC and rely on Isabelle, and in particular on our integration of SMT solvers, for interactive proofs.

- HOL-Boogie is a tool to debug failing proof attempts. It helps users find out *why* the automatic proof fails. HOL-Boogie can also be applied to proof verification conditions in Isabelle with the confidence provided by an LCF-style proof.
- With the abstract cooperation between VCC and Isabelle, properties that are hard to establish in VCC can be proved, on a higher level of abstraction, in Isabelle instead. VCC is restricted to reason about low-level properties of C programs, and it will typically be applied to show the correspondence between program properties and their abstract representation used in Isabelle.

We demonstrated both approaches on case studies. With HOL-Boogie, we verified functional properties of binary search trees (Section 4.4 and Appendix B). We applied abstract cooperation to verify a checker for maximum cardinality matching in graphs (Section 4.6), i.e., we established the relation between a graph algorithm written in C and a theorem from graph theory formalized in Isabelle/HOL.

5.3. Future Work

We still see possibilities to further improve our integration of SMT solvers with Isabelle. Directions for future enhancements or experiments are as follows.

Translation to MSFOL In our translation from HOL to MSFOL (Section 2.2), we made the decisions to monomorphize problems and apply lambda-lifting. Our evaluation gave no clear indication whether these choices in particular contribute to the success of our SMT integration. Experiments that evaluate encodings of types into an untyped logics and that evaluate alternatives to lambda-lifting might give more insights.

Leaving arithmetic constants that are built into SMT solvers uninterpreted occasionally leads to higher success rates (Table 2.4). The combination of treating such constants both interpreted and uninterpreted might be complementary such that even more problems can be solved by SMT solvers. This combination might, for example, use proxy constants with definitions that equate proxy constants to corresponding built-in constants.

New SMT-LIB format A new format for representing SMT problems [16] is emerging, and more and more SMT solvers are gradually adopting it. Among other things, this new format is based on a higher-order logic that identifies the syntactic entities of formulas and terms. Changing our integration to produce problems in this format is

essentially an engineering task. When accomplished, we might evaluate whether proving typical theorems in Isabelle can benefit from the extension of the SMT format. If that is the case, we might also remove from our code the incomplete step that separates formulas from terms (Section 2.2.5).

Improved proof reconstruction for Z3 Proof reconstruction for Z3 lacks support for fixed-size bitvectors and datatypes (Section 3.1). As preliminary prototypes suggest, both of them involve their share of special intricacies. Especially obtaining acceptable replay times for bitvector proofs might be challenging, because Z3 maps bitvector propositions to large propositional formulas (Section 1.4.2). The existing efficient LCF-style integration of SAT solvers in Isabelle [159, 163] might be of help in reconstructing such problems.

A further improvement of Z3 proof reconstruction might be obtained by exploiting the tags that specify the theory and reasoning expressed by th-lemma steps (Section 3.2). As a result, expensive search might potentially be reduced to more efficient specific reconstruction methods.

We observed that some Z3 proofs consist of at least two nearly independent subproofs. Parallel checking [166] of such independent branches can yield performance improvements with current multicore computers.

Countermodels When stating conjectures, users frequently make mistakes, and such mistakes can be unveiled by countermodel finders. Although our integration of SMT solvers with Isabelle concentrates on proving theorems, it might be extended to also provide countermodels for invalid conjectures as most SMT solvers are able to produce models. Although there are already three countermodel finders in Isabelle [24, 128, 160], a fourth one based on SMT solvers might complement those existing three in a similar way as SMT solvers complement ATPs in *sledgehammer*. Especially the combination of the decision procedures for integers and reals with that for equality and uninterpreted functions might lead to countermodels that the existing countermodel finders in Isabelle fail to discover. Yet, as soon as quantifiers are involved, most models found by SMT solvers are only potential ones, because the handling of quantifier instantiations in these solvers is typically incomplete.

Appendix A.

Z3 Proof Rule Names

Short name	Z3 name	Short name	Z3 name
apply-def	`apply-def`	intro-def	`intro-def`
asserted	`asserted`, `goal`	lemma	`lemma`
comm$_{\simeq,\approx}$	`commutativity`	nnf-neg	`nnf-neg`
cong$_{\simeq}$	`monotonicity`	nnf-pos	`nnf-pos`
def-axiom	`def-axiom`	mp$_{\longleftrightarrow}$, mp$_{\longrightarrow}$	`mp`
der	`der`	mp$_{\sim}$	`mp`$\sim$
distributivity	`distributivity`	pull-quant	`pull-quant`
elim$_{\wedge}$	`and-elim`	push-quant	`push-quant`
elim$_{\neg\vee}$	`not-or-elim`	refl$_{\simeq}$	`refl`
elim$_{Q}$	`elim-unused`	rewrite	`rewrite`
hypothesis	`hypothesis`	sk$_{Q}$	`sk`
iff$_{\bot}$	`iff-false`	symm$_{\simeq}$	`symm`
iff$_{\top}$	`iff-true`	th-lemma	`th-lemma`
iff$_{\sim}$	`iff`$\sim$	trans$_{\simeq}$	`trans`
inst$_{\forall}$	`quant-inst`	true	`true-axiom`
intro$_{Q}$	`quant-intro`	unit-resolution	`unit-resolution`

Table A.1. Mapping of the short proof rule names as described in Chapter 3 to the names used by Z3

Appendix B.

A Binary Search Tree Implementation in C

```
#include "vcc.h"
#define Null ((void *) 0)
#define wrapped_inside(e,o) (\wrapped(o) && e \in \domain(o))

typedef unsigned int Key;
typedef enum { Equal, Less, Greater } Keymatch;
struct T_Node; typedef struct T_Node Node, *P_Node;
struct T_Tree; typedef struct T_Tree Tree, *P_Tree;

struct T_Node
{
  Key key;
  P_Node left, right;

  // required only for initialize_node and insert:
  _(ghost P_Tree tree)
  _(ghost bool initialized)
  _(invariant initialized ==> left == &tree->nil)
  _(invariant initialized ==> right == &tree->nil)
};

_(ghost _(pure) bool not_nil(P_Tree tree, P_Node n) _(returns n != &tree->nil))
_(ghost _(pure) bool inner(P_Node n) _(returns n != Null))
_(logic bool owns(P_Tree tree, P_Node n) = (n \in tree->\owns))

#define towns(n) owns(tree, n)
#define tnnil(n) not_nil(tree, n)
#define troot (tree->root)
#define tnode(k) (tree->node[k])
#define tsub(k1,k2) (tree->sub[k1][k2])
#define tinner(k) inner(tnode(k))

_(logic bool basic(P_Tree tree) = {:split} towns(&tree->nil) && towns(troot))

_(logic bool abstraction(P_Tree tree) = {:split}
  (\forall P_Node n; {towns(n), tnnil(n)}
    towns(n) && tnnil(n) ==> tnode(n->key) == n) &&
  (\forall P_Node n; {towns(n), tinner(n->key)}
```

```
    towns(n) && tinner(n->key) ==> tnnil(n)) &&
  (!tnnil(troot) ==> \forall Key k; !tinner(k)) &&
  (\forall Key k; {tnode(k)} tnnil(tnode(k))) &&
  (\forall Key k; {tinner(k)} tinner(k) ==> towns(tnode(k))) &&
  (\forall Key k; {tinner(k)} tinner(k) ==> tnnil(tnode(k))) &&
  (\forall Key k; {tinner(k)} tinner(k) ==> tnode(k)->key == k) &&
  (\forall Key k; {tinner(k)} tinner(k) ==> towns(tnode(k)->left)) &&
  (\forall Key k; {tinner(k)} tinner(k) ==> towns(tnode(k)->right)) &&
  (\forall Key k; {tinner(k)} tinner(k) ==> !tnode(k)->initialized))

_(logic bool subtrees(P_Tree tree) = {:split}
  (\forall Key k; {tinner(k)} tinner(k) ==> tsub(tnode(k)->key, k)) &&
  (tnnil(troot) ==> \forall Key k; {tinner(k)}
    tinner(k) ==> tsub(troot->key, k)) &&
  (\forall Key k1, k2; {tsub(k1, k2)} tsub(k1, k2) ==> tinner(k1)) &&
  (\forall Key k1, k2; {tsub(k1, k2)} tsub(k1, k2) ==> tinner(k2)))

_(logic bool sorted(P_Tree tree) = {:split}
  (\forall Key k1, k2; {tsub(k1, k2)}
    tsub(k1, k2) && k1 > k2 ==> tsub(tnode(k1)->left->key, k2)) &&
  (\forall Key k1, k2; {tsub(k1, k2)}
    tsub(k1, k2) && k1 < k2 ==> tsub(tnode(k1)->right->key, k2)))

_(dynamic_owns) struct T_Tree
{
  Node nil;  // an empty node (representing leafs)
  P_Node root;  // the root of the tree
  _(ghost P_Node node[Key])  // an abstraction of this tree's proper nodes
  _(ghost bool sub[Key][Key])  // subtrees
  _(invariant {:split} basic(\this) && abstraction(\this))
  _(invariant {:split} subtrees(\this) && sorted(\this))
};


Keymatch compare_keys (Key k1, Key k2)
  _(ensures (\result == Equal) <==> (k1 == k2))
  _(ensures (\result == Less) <==> (k1 < k2))
  _(ensures (\result == Greater) <==> (k1 > k2))
{
  if (k1 == k2) return Equal;
  else if (k1 < k2) return Less;
  else return Greater;
}


void initialize (P_Tree tree)
  _(writes \extent(tree))
  _(ensures \wrapped(tree))
  _(ensures tree->node == (\lambda Key k; (P_Node) Null))
```

```
  _(ensures tree->\owns == {tree->root})
{
  tree->root = &tree->nil;
  _(ghost tree->nil.initialized = \false)
  _(wrap tree->root)
  _(ghost tree->node = (\lambda Key k; (P_Node) Null))
  _(ghost tree->sub = (\lambda Key k1; \lambda Key k2; \false))
  _(ghost tree->\owns = {tree->root})
  _(wrap tree)
}


P_Node lookup (P_Tree tree, Key key  _(ghost \object root))
  _(requires wrapped_inside(tree, root))
  _(ensures \result != Null ==> \result \in tree->\owns)
  _(ensures \result != Null ==> \result->key == key)
  _(returns tree->node[key])
{
  P_Node current;
  Keymatch match;
  current = tree->root;

  while (current != &tree->nil)
    _(invariant current \in tree->\owns)
    _(invariant inner(tree->node[key]) ==> not_nil(tree, current))
    _(invariant inner(tree->node[key]) ==> tree->sub[current->key][key])
  {
    _(assert not_nil(tree, current) && inner(tree->node[current->key]))

    match = compare_keys(key, current->key);
    switch (match)
    {
      case Equal:
        return current;
      case Less:
        current = current->left;
        break;
      case Greater:
        current = current->right;
        break;
      default:
        return Null;  // never reached
    }
  }
  return Null;
}


void initialize_node (P_Tree tree, P_Node node)
```

```
  // initialize a node prior to insertion
  _(writes \span(node))
  _(ensures \wrapped(node))
  _(ensures node->tree == tree)
  _(ensures node->initialized)
{
  node->left = &tree->nil;
  node->right = &tree->nil;
  _(ghost node->tree = tree)
  _(ghost node->initialized = \true)
  _(wrap node)
}


int insert (P_Tree tree, Key key, P_Node node)
  _(requires \wrapped(node) && !(node \in tree->\owns))
  _(requires node->initialized && node->tree == tree)
  _(writes tree, node)
  _(maintains \wrapped(tree))
  _(ensures \result ==> tree->\owns == {node} \union \old(tree->\owns))
  _(ensures \result ==> node->key == key)
  _(ensures \result ==>
    tree->node == \lambda Key k; k == key ? node : \old(tree->node[k]))
  _(ensures !\result ==> \unchanged(tree->node))
  _(ensures !\result ==> \unchanged(tree->\owns))
  _(ensures !\result ==> node->initialized)
  _(returns \old(tree->node[key]) == Null)
{
  Keymatch match;
  P_Node x, y;
  _(ghost P_Node old_node[Key])
  _(ghost \objset old_owns)

  match = Equal;
  y = &tree->nil;
  x = tree->root;
  _(ghost old_node = tree->node)
  _(ghost old_owns = tree->\owns)

  while (x != &tree->nil)
    _(invariant \wrapped(node) && \wrapped(tree))
    _(invariant node->initialized && node->tree == tree)
    _(invariant node->left == &tree->nil && node->right == &tree->nil)
    _(invariant tree->\owns == old_owns && tree->node == old_node)
    _(invariant tree \in \domain(tree))
    _(invariant x \in tree->\owns && y \in tree->\owns)
    _(invariant not_nil(tree, y) <==> x != tree->root)
    _(invariant not_nil(tree, y) <==> match == Less || match == Greater)
    _(invariant inner(tree->node[key]) ==> not_nil(tree, x))
```

```
    _(invariant inner(tree->node[key]) ==> tree->sub[x->key][key])
  {
    _(assert not_nil(tree, x) && inner(tree->node[x->key]))

    y = x;
    match = compare_keys(key, x->key);
    switch (match)
    {
      case Less:
        x = x->left;
        break;
      case Greater:
        x = x->right;
        break;
      case Equal:   // fall through to default case
      default:
        return 0;
    }
    _(assume x != tree->root)
  }
  _(assert not_nil(tree, y) ==> inner(tree->node[y->key]))

  _(unwrapping node)
  {
    node->key = key;
    _(ghost node->initialized = \false;)
  }

  _(unwrapping tree)
  {
    _(ghost tree->\owns += node);
    _(ghost tree->node[key] = node)

    if (y == &tree->nil) tree->root = node;
    else if (match == Less) {  _(unwrapping y) { y->left = node; } }
    else { _(unwrapping y) { y->right = node; } }

    _(assume subtrees(tree))
    _(assume sorted(tree))
  }
  return 1;
}
```

List of Figures

List of Tables

Bibliography

[1] 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 of *Systems and Implementation Techniques*, pages 97–116. Kluwer, 1998.

[2] E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. Verification of certifying computations. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2011.

[3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer, 2nd edition, 2002.

[4] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In G. Faure, S. Lengrand, and A. Mahboubi, editors, *Proof-Search in Axiomatic Theories and Type Theories*, 2011.

[5] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010.

[6] N. Ayache and J.-C. Filliâtre. Combining the Coq proof assistant with first-order decision procedures. `http://www.lri.fr/~filliatr/publis/coq-dp.ps`, 2006.

[7] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[8] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[9] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[10] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[11] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[12] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In M. D. Ernst and T. P. Jensen, editors, *Program Analysis for Software Tools and Engineering*, pages 82–87. ACM, 2005.

[13] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[14] C. Barrett, M. Deters, A. Oliveras, and A. Stump. 5th Annual Satisfiability Modulo Theories Competition (SMT-COMP), 2009. http://www.smtcomp.org/2009/.

[15] C. Barrett, A. Stump, and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). http://www.smt-lib.org, 2010.

[16] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Satisfiability Modulo Theories*, 2010.

[17] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[18] D. Barsotti, L. P. Nieto, and A. F. Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. *Electronic Notes in Theoretical Computer Science*, 145:63–78, 2006.

[19] S. Berghofer. Verification of dependable software using SPARK and Isabelle. In J. Brauer, M. Roveri, and H. Tews, editors, *Systems Software Verification*, 2011.

[20] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[21] F. Besson, P. Fontaine, and L. Théry. A flexible proof format for SMT: a proposal. In P. Fontaine and A. Stump, editors, *Proof eXchange for Theorem Proving*, pages 15–26, 2011.

[22] M. Bezem, D. Hendriks, and H. de Nivelle. Automatic proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3–4):253–275, 2002.

[23] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Børner and V. Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[24] 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, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[25] M. Blum and S. Kannan. Designing programs that check their work. In *Theory of Computing*, pages 86–97, 1989.

[26] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *Satisfiability Modulo Theories*, pages 1–5. ACM, 2008.

[27] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. Technical report, INRIA, 2011. http://hal.inria.fr/inria-00591414/en/.

[28] S. Böhme. Proof reconstruction for Z3 in Isabelle/HOL. In B. Dutertre and O. Strichman, editors, *Satisfiability Modulo Theories*, pages 98–108, 2009.

[29] S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie—An interactive prover for the Boogie program-verifier. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2008.

[30] S. Böhme and M. Moskal. Heaps and data structures: A challenge for automated provers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.

[31] S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie—an interactive prover-backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, 2010.

[32] S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[33] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[34] S. Böhme and T. Weber. Designing proof formats: A user's perspective. In P. Fontaine and A. Stump, editors, *Proof eXchange for Theorem Proving*, pages 27–32, 2011.

[35] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[36] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[37] A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.

[38] J. D. Bright, G. F. Sullivan, and G. M. Masson. A formally verified sorting certifier. *IEEE Transactions on Computers*, 46(12):1304–1312, 1997.

[39] R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In B. Dutertre and O. Strichman, editors, *Satisfiability Modulo Theories*, pages 1–5. ACM, 2009.

[40] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.

[41] E. Charniak, R. Christopher K, and D. V. McDermitt. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1980.

[42] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[43] V. Chvatal. *Linear Programming*. W. H. Freeman and Company, 1983.

[44] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[45] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-2019, Microsoft Research, 2009.

[46] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. In R. Huuck, G. Klein, and B. Schlich, editors, *Systems Software Verification*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Elsevier Science B.V., 2009.

[47] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. Technical Report MSR-TR-2010-9, Microsoft Research, 2010.

[48] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight integration of the Ergo theorem prover inside a proof assistant. In J. Rushby and N. Shankar, editors, *Automated Formal Methods*, pages 55–59. ACM, 2007.

[49] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, 1972.

[50] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 2nd edition, 1990.

[51] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In F. Pfenning, editor, *Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2007.

[52] G. Dantzig and B. Curtis. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory*, pages 288–297, 1973.

[53] A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In A. Cavalcanti, D. Deharbe, M.-C. Gaudel, and J. Woodcock, editors, *Theoretical Aspects of Computing*, volume 6255 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2010.

[54] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[55] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[56] J. Dawson. Isabelle theories for machine words. *Electronic Notes in Theoretical Computer Science*, 250(1):55–70, 2009.

[57] L. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In F. Pfenning, editor, *Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[58] L. M. de Moura and N. Bjørner. Proofs and refutations, and Z3. In P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, editors, *International Workshop on the Implementation of Logics*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[59] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[60] R. DeLine and K. R. M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[61] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[62] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[63] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[64] B. Dutertre and L. de Moura. The Yices SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[65] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[66] L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In J. Rushby and N. Shankar, editors, *Automated Formal Methods*, pages 3–13, 2008.

[67] J.-C. Filliâtre. Why: A multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.

[68] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[69] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonization and solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2001.

[70] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

[71] 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, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[72] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[73] Y. Ge and C. Barrett. Proof translation and SMT-LIB benchmark certification: A preliminary report. In C. Barrett and L. de Moura, editors, *Satisfiability Modulo Theories*. ACM, 2008.

[74] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In A. Bouajjani and O. Maler, editors, *Computer*

Aided Verification, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[75] M. Gordon. From LCF to HOL: a short history. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 169–185. MIT Press, 2000.

[76] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[77] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[78] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems Series*, chapter 3, pages 49–70. Elsevier, 1994.

[79] C. L. Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger. http://research.microsoft.com/en-us/um/people/moskal/pdf/bvd.pdf, 2011.

[80] J. Guitton, J. Kanig, and Y. Moy. Why hi-lite Ada? In K. R. M. Leino and M. Moskal, editors, *Boogie*, pages 27–39, 2011.

[81] W. Guttmann, G. Struth, and T. Weber. A repository for Tarski-Kleene algebras. In P. Höfner, A. McIver, and G. Struth, editors, *Automated Theory Engineering*, volume 760 of *CEUR Workshop Proceedings*, 2011.

[82] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995. http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz.

[83] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[84] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[85] J. Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, 1999.

[86] 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.

[87] C. Hurlin, A. Chaieb, P. Fontaine, S. Merz, and T. Weber. Practical proof reconstruction for first-order logic and set-theoretical constructions. In L. Dixon and M. Johansson, editors, *Isabelle Workshop*, pages 2–13, 2007.

[88] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, 1999.

[89] P. B. Jackson and G. O. Passmore. Proving SPARK verification conditions with SMT solvers. http://homepages.inf.ed.ac.uk/pbj/papers/vct-dec09-draft.pdf, 2009.

[90] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[91] P. James and P. Chalin. Faster and more complete extended static checking for the Java Modeling Language. *Journal of Automated Reasoning*, 44:145–174, 2010.

[92] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer, 1985.

[93] J. Kanig. *Specification and Proof of Higher-Order Programs*. PhD thesis, Laboratoire de Recherche en Informatique, Orsay, France, 2010.

[94] C. Keller. Cooperation between SAT, SMT provers and Coq, 2011. Presentation at the Synthesis, Verification and Analysis of Rich Models workshop.

[95] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[96] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

[97] D. Kroening and O. Strichman. *Decision Procedures – An Algorithmic Point of View*. Springer, 2008.

[98] R. Kumar and T. Weber. Validating QBF validity in HOL4. In M. V. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2011.

[99] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, 2007.

[100] O. Kunčar. Proving valid quantified boolean formulas in HOL-Light. In M. V. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2011.

[101] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[102] D. Leinenbach, W. J. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In B. K. Aichernig and B. Beckert, editors, *Software Engineering and Formal Methods*, pages 2–12. IEEE Computer Society, 2005.

[103] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In A. Cavalcanti and D. Dams, editors, *Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer, 2009.

[104] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[105] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[106] K. R. M. Leino, T. D. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1–3):209–226, 2005.

[107] K. R. M. Leino and M. Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Verified Software: Tools and Experiments*, 2010.

[108] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2009.

[109] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

[110] P. Manolios and S. K. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *International Conference on Computer Aided Design*, pages 855–862. IEEE Computer Society, 2005.

[111] M. Manzano. *Extensions of First Order Logic*. Cambridge University Press, 1996.

[112] marianrh. Help with simple algorithm needed (arithmetic overflow), 15 September 2011. Archived at `http://vcc.codeplex.com/discussions/272722`.

[113] F. Maric. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.

[114] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[115] W. McCune and O. Shumsky. System description: IVY. In D. A. McAllester, editor, *Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 401–405. Springer, 2000.

[116] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51, 2006.

[117] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[118] J. Meng and L. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40:35–60, 2008.

[119] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[120] J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[121] M. Moskal. Rocket-fast proof checking for SMT solvers. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2008.

[122] M. Moskal. Programming with triggers. In B. Dutertre and O. Strichman, editors, *Satisfiability Modulo Theories*, pages 20–29. ACM, 2009.

[123] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535. ACM, 2001.

[124] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, 2009.

[125] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[126] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[127] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[128] S. B. T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods*, pages 230–239. IEEE Computer Society, 2004.

[129] T. Nipkow. Re: [isabelle] A beginner's questionu [sic], 26 November 2010. Archived at https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2010-November/msg00097.html.

[130] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[131] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[132] D. Oe, A. Reynolds, and A. Stump. Fast and flexible proof checking for SMT. In *Satisfiability Modulo Theories*, pages 6–13. ACM, 2009.

[133] D. C. Oppen. Reasononing about recursively defined data structures. *Journal of the ACM*, 27(3):403–411, 1980.

[134] V. Ortner and N. Schirmer. Verification of BDD normalization. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 261–277, 2005.

[135] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258, 1986.

[136] L. C. Paulson. *Isabelle – A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[137] L. C. Paulson. A generic tableau prover and its integration with isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.

[138] L. C. Paulson and J. Blanchette. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In E. Ternovska, S. Schulz, and G. Sutcliffe, editors, *International Workshop on the Implementation of Logics*, 2010. Invited talk.

[139] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2007.

[140] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13. ACM, 1991.

[141] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. http://www.smt-lib.org, 2006.

[142] E. Reeber and W. A. H. Jr. A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). In U. Furbach and N. Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2006.

[143] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[144] C. Rizkallah. Maximum cardinality matching. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/Max-Card-Matching.shtml, 2011. Formal proof development.

[145] J. M. Rushby. Tutorial: Automated formal methods with PVS, SAL, and Yices. In D. V. Hung and P. Pandya, editors, *Software Engineering and Formal Methods*, page 262. IEEE, 2006.

[146] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[147] N. Shankar. Little engines of proof. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe: Formal Methods – Gettint IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2002.

[148] 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.

[149] J. P. M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[150] G. F. Sullivan and G. M. Masson. Using certification trails to achieve software fault tolerance. In B. Randell, editor, *Fault-Tolerant Computing*, pages 423–433. IEEE, 1990.

[151] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[152] G. Sutcliffe, C. Benzmüller, C. E. Brown, and F. Theiss. Progress in the development of automated theorem proving for higher-order logic. In R. A. Schmidt, editor, *Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2009.

[153] 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.

[154] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.

[155] A. P. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[156] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967 - 1970*, pages 466–483. Springer, 1983.

[157] J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *Journal of Automated Reasoning*, 37(1–2):21–43, 2006.

[158] F. Vogels, B. Jacobs, and F. Piessens. A machine-checked soundness proof for an efficient verification condition generator. In *Symposium on Applied Computing*, volume 3, pages 2517–2522. ACM, 2010.

[159] T. Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. In J. Hurd, E. Smith, and A. Darbari, editors, *Theorem Proving in Higher Order Logics*, pages 180–189. Oxford University Computing Laboratory, Programming Research Group, 2005. Research Report PRG-RR-05-02.

[160] T. Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2008.

[161] T. Weber. SMT solvers: New oracles for the HOL theorem prover. In J.-C. Filliâtre and L. Freitas, editors, *Verified Software: Theory, Tools, and Experiments*, 2009.

[162] T. Weber. Validating QBF invalidity in HOL4. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 466–480. Springer, 2010.

[163] T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, 2009.

[164] M. Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.

[165] M. Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logic*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

[166] M. Wenzel. Parallel proof checking in Isabelle/Isar. In *Programming Languages for Mechanized Mathematics Systems*, 2009.

[167] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In R. Gupta and S. P. Amarasinghe, editors, *Programming Language Design and Implementation*, pages 349–361. ACM, 2008.

[168] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In A. Voronkov, editor, *Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.