Automatic Proof and Disproof in Isabelle/HOL

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow

Fakultät für Informatik, Technische Universität München

Abstract. Isabelle/HOL is a popular interactive theorem prover based on higher-order logic. It owes its success to its ease of use and powerful automation. Much of the automation is performed by external tools: The metaprover Sledgehammer relies on resolution provers and SMT solvers for its proof search, the counter-example generator Quickcheck uses the ML compiler as a fast evaluator for ground formulas, and its rival Nitpick is based on the model finder Kodkod, which performs a reduction to SAT. Together with the Isar structured proof format and a new asynchronous user interface, these tools have radically transformed the Isabelle user experience. This paper provides an overview of the main automatic proof and disproof tools.

1 Introduction

In the tradition of LCF-style interactive theorem provers [21], Isabelle [35] has long emphasized tactics: functions written in ML that operate on the proof state via a trusted inference kernel. Tactics discharge a proof goal directly or, more often, break it down into one or more subgoals that must then be tackled by other tactics. In the last decade, the structured Isar language [34, 57] has displaced ML as the language of choice for Isabelle proofs, but the most important ML tactics are still available as Isar proof methods.

Much effort has been devoted to developing general-purpose proof methods (or tactics) that work equally well on all object logics supported by Isabelle, notably higher-order logic (HOL) [20] and Zermelo–Fraenkel set theory (ZF) [37, 38]. The most important methods are the simplifier, which rewrites the goal using equations as oriented rewrite rules, and the tableau prover (Section 2). These are complemented by specialized decision procedures, especially for arithmetic. For the users of an interactive theorem prover, one of the main challenges is to find out which proof methods to use and which arguments to specify.

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

The most important proof tool besides the simplifier and the tableau prover is probably Sledgehammer, which connects Isabelle with external resolution provers and SMT solvers (Section 3). It boasts a fairly high success rate on goals that cannot be discharged directly by standard proof methods: In a recent study involving older Isabelle proof scripts, Sledgehammer could prove 43% of the more difficult goals contained in those proofs [6]. The addition of SMT solvers is recent and helps solve both arithmetic and nonarithmetic problems [6]. Sledgehammer works well in combination with structured Isar proofs: The new way of teaching Isabelle is to let students think up intermediate properties and rely on automatic tools to fill in the gaps, rather than teach them low-level tactics and have them memorize lemma libraries [41, §4].

As useful as they might be, most automatic proof tools are helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy. To make proving more enjoyable and productive, Isabelle includes counterexample generators that complement the proof tools. The main ones are Quickcheck (Section 4) and Nitpick (Section 5).

Quickcheck [3] combines Isabelle's code generation infrastructure with random testing, in the style of the QuickCheck tool for Haskell [14]. It analyses the definitions of inductively defined predicates to generate values that satisfies them by construction [11] and has recently been extended with exhaustive testing and narrowing.

A radically different approach is based on systematic model enumeration using a SAT solver. This approach was pioneered by the tool Refute [54] and is now embodied by Nitpick [8]. Nitpick looks for finite fragments (substructures) of infinite countermodels, soundly approximating problematic constructs. Common Isabelle idioms, such as inductive and coinductive predicates and datatypes as well as recursive and corecursive functions, are treated specially to ensure efficient SAT solving. The actual reduction to SAT is performed by the Kodkod library [53] (the Alloy Analyzer's [25] backend).

With so many tools at their disposal, users run the risk of forgetting to invoke them at the right point; this is especially true for the counterexample generators, given that humans have a natural tendency to trust their own conjectures. For this reason, the proof and disproof tools can be set up to run automatically in parallel for a few seconds on all newly entered conjectures. They can of course also be launched at any point in a proof with a more liberal time limit. Either mode of operation exploits multiple processor cores if they are available, and Sledgehammer also sends its problems to remote servers to further distribute the load.

2 Standard Proof Methods

AKA Tactics

Isabelle provides the user with an array of general-purpose proof methods that perform proof search. We discuss the most important ones.

2.1 Simplification

Just as in ACL2 [26], simplification is the main workhorse in Isabelle. It performs conditional, contextual rewriting with a number of hooks for customizations:

- *Pattern-driven simplification procedures* that derive and apply rewrite rules dynamically. Many such procedures are preinstalled, notably arithmetic simplification procedures for numerals and symbolic terms.
- *Special solvers for conditional rewrite rules*. Typical examples are fragments of linear arithmetic and a transitive closure prover for arbitrary transitive relations.

– *Special "loopers"* that massage the goal after each round of simplification. Case splitting methods are provided this way.

The power of the simplifier is due to these extensions to rewriting together with the vast and growing library of registered rewrite rules.

2.2 Auto & Co.

On the user level, the simplifier is eclipsed by *auto*, a proof method that interleaves simplification with a small amount of proof search. It is impossible to describe succinctly what *auto* does due to its heuristic, ad hoc nature. Its great strength is its ability to discharge the easy parts of a goal and leave the user with the more difficult ones. This helps the user to quickly focus on the core of a problem.

Strengthened versions of *auto* perform more sophisticated proof search, while still interleaving it with simplification. The search is based on tableau methods [39]. These methods are often useful, but since search is involved, not only are they slower than the simplifier and *auto*, they are endgame provers that do not provide any hints when they fail to prove the goal.

2.3 Blast and Metis

The tableau implementation mentioned above can be very slow because every inference step is performed directly on the proof state, via the Isabelle kernel. For more performance, users can choose *blast* [40], a tableau prover written directly in ML that bypasses the kernel; once a proof has been found, it is replayed in the kernel to check it. The *blast* method outperforms the kernel-based tableau implementation by a wide margin but is no match for the best automatic provers. Nor does it know about simplification, which is a great loss.

Taking this one step further, Metis is a resolution theorem prover written in ML by Hurd [24]. Metis is sufficiently capable that it is a respectable competitor at CASC [51]. It has been ported to Isabelle and follows the same philosophy as *blast*: The proof search is performed directly in ML, and any proof found is checked by the Isabelle kernel.

The *blast* method relies on an extensible lemma database that drives the search and that is preconfigured to reason about sets, functions and relations, which makes it quite user-friendly. In contrast, Isabelle's version of Metis knows only about pure logic and derives its knowledge about other operators from explicitly supplied lemmas. Although Metis can be invoked directly, in practice Metis calls are almost always generated by Sledgehammer for reconstructing external resolution proofs (Section 3.4).

3 Sledgehammer: Proof Discovery Using External Provers

Sledgehammer [31, 42] is Isabelle's subsystem for harnessing the power of first-order automatic theorem provers. Given a conjecture, it heuristically selects a few hundred relevant facts (lemmas, definitions, or axioms) from Isabelle's libraries, translates them to first-order logic along with the conjecture, and delegates the proof search to external resolution provers (E [48], SPASS [56], and Vampire [44]) and SMT solvers (CVC3 [2], Yices [16], and Z3 [33]). Sledgehammer is very effective [9] and has achieved great popularity with users, novices and experts alike.

3.1 Relevance Filtering

Most automatic provers perform poorly in the presence of thousands of axioms. Sledgehammer employs a simple relevance filter [32] to extract a few hundred facts from Isabelle's libraries that seem relevant to the problem at hand. Despite its simplicity, this filter greatly improves Sledgehammer's success rate.

The filter works iteratively. The first iteration selects facts that share all or nearly all of their constants (symbols) with the conjecture. Further iterations also include facts that share constants with previously selected facts, until the desired number of facts is reached. Observing that some provers cope better with large axiom bases than others, that number was optimized independently for each prover.

3.2 Translation to First-Order Logic

Isabelle's formalism, polymorphic higher-order logic with type classes [59], is much richer than the first-order logics supported by the automatic provers. Sledgehammer relies on different translations depending on the class of prover [6, 31].

For resolution provers, standard techniques are employed to translate HOL formulas to classical first-order logic: λ-abstractions are rewritten to combinators, and curried functions are passed varying numbers of arguments by means of an explicit apply operator. Until recently, the translation of types was unsound: It provided enough type information to enforce correct type class reasoning but not to specify the type of every term. (Because the proofs are rechecked by Isabelle's inference kernel, soundness is not crucial.) The current implementation safely erases most type information by inferring type monotonicity [7, 15], resulting in a sound and efficient encoding.

For SMT solvers, the translation maps equality and arithmetic operators to the corresponding SMT-LIB [43] concepts. The SMT-LIB logic is many-sorted, which would seem to make it more appropriate to encode HOL typing information than classical first-order logic, but it does not support polymorphism. The solution is to monomorphize the formulas: Polymorphic formulas are iteratively instantiated with relevant ground instances of their polymorphic constants. This process is iterated to obtain the monomorphized problem. Partial applications are translated using an apply operator, but in contrast with the combinator approach used when communicating with resolution provers, λ-abstractions are lifted into new rules, thereby introducing fresh constants.

3.3 Invocation of External Provers

Sledgehammer lets the external provers run in parallel, either locally or remotely. On a typical Isabelle installation, E, SPASS, and Z3 are run on the user's machine, whereas Vampire and the SInE metaprover [23] are provided via the remote SystemOnTPTP service [50]. Users can also enable CVC3 and Yices.

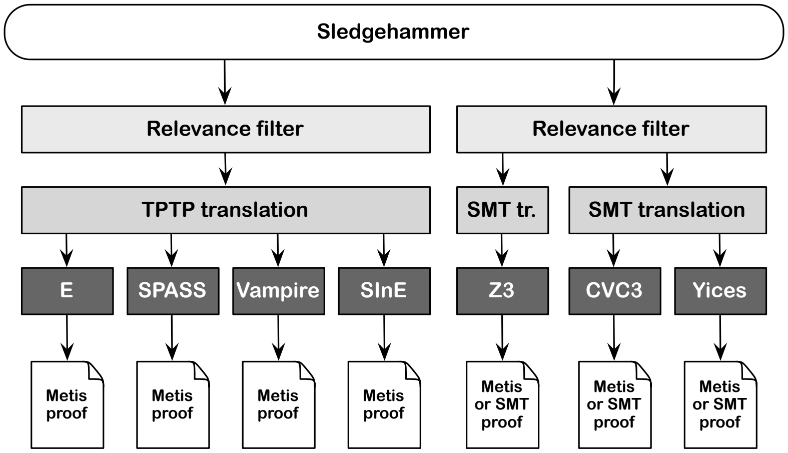

Fig. 1. Sledgehammer's architecture

Figure 1 depicts the architecture, omitting proof reconstruction and minimization. Two instances of the relevance filter are run, to account for different sets of built-in constants. The relevant facts and the conjecture are translated to the TPTP [52] or SMT version of first-order logic, and the resulting problems are passed to the provers. The translation for Z3 is done slightly differently than for CVC3 and Yices to profit from Z3's support for nonlinear arithmetic.

Third-party provers should ideally be bundled with Isabelle and ready to be used without requiring configuration. Isabelle includes CVC3, E, SPASS, and Z3 executables for the major hardware platforms; users can download Yices and Vampire, whose licenses forbid redistribution, but most simply run Vampire remotely on SystemOnTPTP. In addition, we set up a server in Munich in the style of SystemOnTPTP for running CVC3 and Z3 remotely.

Remote servers are satisfactory for proof search, at least when they are up and running and the user has Internet access. They also help distribute the load: Unless the user's machine has eight processor cores, it would be reckless to launch four resolution provers and three SMT solvers and expect the Isabelle user interface to remain responsive. The parallel invocation of provers is invaluable: Running E, SPASS, and Vampire together for five seconds solves as many problems as running a single prover for two minutes [9, §8].

3.4 Proof Reconstruction

In keeping with the LCF philosophy [21], Isabelle theorems can only be generated within a small inference kernel. It is possible to bypass this safety mechanism, if some external tool is to be trusted as an oracle, but all oracle inferences are tracked.

For resolution provers, Sledgehammer performs true proof reconstruction by running Isabelle's built-in resolution prover, Metis, supplying it with the short list of facts used in the proof found by the prover. Given only a handful of facts, Metis usually succeeds within milliseconds. Since Metis has to re-find the proof, the external provers are essentially used as very precise relevance filters.

As an example, consider the conjecture "*length* (*tl xs*) $\leq$ *length xs*", which states that the length of a list's tail (its "cdr") is less than or equal to the length of the entire list. Thanks to Vampire, Sledgehammer finds the following proof:

```
by (metis append_Nil2 append_eq_conv_conj drop_eq_Nil drop_tl tl.simps(1))
```

Proof reconstruction using Metis loses about 4% of resolution proofs because Metis times out, typically because the proof found by the external prover is too long. Sledgehammer then falls back on a detailed Isabelle proof, expressed in the structured Isar language. While the detailed output is primarily designed for replaying resolution proofs, it also has a pedagogical value. Unlike Isabelle's automatic tactics, which are black boxes, the proofs delivered by Sledgehammer can be inspected and understood, as in the example below:

```
proof –
  have "tl [] = []" by (metis tl.simps(1))
  hence "∃u. xs @ u = xs ∧ tl u = []" by (metis append_Nil2)
  hence "tl (drop (length xs) xs) = []" by (metis append_eq_conv_conj)
  hence "drop (length xs) (tl xs) = []" by (metis drop_tl)
  thus "length (tl xs) ≤ length xs" by (metis drop_eq_Nil)
qed
```

The generated proofs often require some postediting to make them syntactically correct. Efforts are underway to make the generated output both more robust and more concise.

On the SMT side of things, proofs that involve no arithmetic reasoning steps can usually be replayed by Metis; otherwise, step-by-step proof replay is supported for Z3 [10], whereas CVC3 and Yices can be invoked as oracles. Z3 proof replay relies extensively on Isabelle's simplifier, tableau prover, and arithmetic decision procedures. Certificates make it possible to store Z3 proofs alongside Isabelle formalizations, allowing proof replay without Z3; only if the formalizations change must the certificates be regenerated. Using SMT solvers as oracles requires trusting both the solvers and the translation to first-order logic, so it is generally frowned upon.

3.5 Proof Minimization

The external provers frequently use many more facts than are necessary. Sledgehammer's minimization tool takes the set of used facts returned by a prover and repeatedly invokes the prover with subsets of the facts to find a minimal set. Depending on the number of initial facts, it relies either on a naive linear algorithm that attempts to remove one fact at a time or on a binary algorithm that recursively bisects the facts [9, §7].

Minimization often improves Metis's performance and success rate, while removing clutter from the Isabelle formalizations. For some provers, it is difficult or impossible to extract the list of used facts from the proof; minimization is then the only option. For example, the detailed proofs returned by CVC3 always refer to all facts, whether they are actually needed or not, and there is no easy criterion to isolate the needed facts.

4 Quickcheck: Counterexample Generation by Testing

Isabelle's proof methods and Sledgehammer are effective for proving valid conjectures, but given an invalid conjecture they normally fail to detect the invalidity, let alone produce an informative counterexample. This is where Quickcheck comes into play.

Quickcheck was originally modeled after the QuickCheck tool for Haskell [14], which tests user-supplied properties of a Haskell program for randomly generated values. We recently extended Quickcheck with exhaustive and narrowing-based testing as complements to random testing. Exhaustive testing checks the formula for every possible set of values up to a given bound, as in SmallCheck [46], and hence finds counterexamples that random testing might miss. Narrowing can be more precise and more efficient than the other two approaches because it considers the formula symbolically, instead of testing a finite set of ground values.

Thanks to a static data-flow analysis inspired by logic programming [11], Quickcheck derives test data generators that take premises into account to help avoid the vacuous test cases that plague most specification testing tools.

4.1 Random and Exhaustive Testing

Quickcheck's random testing strategy repeatedly evaluates the conjecture with pseudorandom values for its free variables. The procedure is parameterized by a size bound on the generated values and the number of tests to perform. The distribution is biased toward smaller values [3, §4].

In principle, Quickcheck could use the Isabelle simplifier to evaluate the conjecture for specific values of its free variables, but it is much more efficient to translate the conjecture and related definitions to an ML (or Haskell) program, exploiting Isabelle's code generation infrastructure [22]: The ML runtime environment can check millions of test cases within seconds, which is thousands of times faster than the simplifier.

Random testing tends to be fast and sometimes finds large counterexamples. Indeed, the QuickCheck tool for Haskell includes a minimizer to reduce overly large counterexamples, a refinement that our Quickcheck implementation currently lacks. But random testing can easily miss counterexamples, even seemingly obvious ones. It also struggles with conjectures that have hard-to-satisfy premises.

An alternative strategy is exhaustive testing, which systematically enumerates values up to a size bound (e.g., all lists of length up to 5). This ensures that all possible variable assignments up to a given size are tested. Hence, if there is a small enough counterexample, it will be found. The main drawback of this strategy is that the number of test cases quickly explodes with increasing size bounds.

Through empirical testing we found the two strategies to be roughly comparable on most types of formula, but exhaustive testing tends to be more successful on conjectures with hard-to-satisfy premises, simply because it will encounter the few small values that fulfill the conditions if such values exist, whereas random testing might miss them. The following conjecture about lists illustrates this point:

$$\mathit{nth}\ (xs \mathbin{@} ys)\ (\mathit{length}\ xs + n) = \mathit{nth}\ xs\ n$$

The nth function returns the element at a given index in a list, and @ is the append operator. The conjecture attempts to relate the elements of xs @ ys with those of ys, but a typo slipped in: The right-hand side should read $\mathit{nth}\ ys\ n$. Exhaustive testing immediately finds a counterexample with $xs = [\mathsf{a}_1]$ and $ys = [\mathsf{a}_2]$ (for $\mathsf{a}_1 \neq \mathsf{a}_2$). Random testing typically fails to find the counterexample, even with hundreds of iterations, because randomly chosen values for n are almost always out of bounds. Since such examples occur frequently in practice, we have now made exhaustive testing the default strategy.

4.2 Test Data Generation

Random and exhaustive testing generate values without analyzing the conjecture. This can lead to many vacuous test cases, as in this simple example:

$$\mathit{length}\ xs = \mathit{length}\ ys \wedge \mathit{zip}\ xs\ ys = zs \Longrightarrow \mathit{map}\ \mathit{fst}\ zs = xs \wedge \mathit{map}\ \mathit{snd}\ zs = ys$$

The random and exhaustive strategies first generate values for xs, ys, and zs in an unconstrained fashion and then check the premises, namely that xs and ys are of equal length and that zs is the list obtained by zipping xs and ys together. For the vast majority of variable assignments, the premises are not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the premises into account when generating values.

We recently extended Quickcheck with test data generators that construct values in a bottom-up fashion, simultaneously testing the conjecture and generating appropriate values [11]. Briefly, we synthesize the test data generator associated with a given premise by reformulating the premise as Horn clauses and computing their data-flow dependencies; from this data-flow analysis, we synthesize generators that directly compute appropriate value.

When transforming the premises to Horn clauses, we replace n-ary functions with $(n+1)$-ary predicates; this gives more freedom to the data-flow analysis, which can then invert functions. The data-flow analysis is an extension of a classic analysis from logic programming. To execute a predicate, its arguments are classified as input or output, made explicit by means of modes. A *mode* is a data-flow assignment that annotates all arguments of a predicate as input (i) or output (o). For example, the binary predicate of type $\alpha\ \mathit{list} \to \mathit{nat} \to \mathit{bool}$ corresponding to the function length supports several modes:

- From the first argument xs, we can compute the second argument by evaluating $\mathit{length}\ xs$. This corresponds to the mode $i \to o \to \mathit{bool}$.
- Inversely, we can enumerate lists of a given length: $o \to i \to \mathit{bool}$.
- Given a list and a natural number, we can check whether the list's length equals that number: $i \to i \to \mathit{bool}$.
- Or we can simply enumerate all pairs (xs, n) such that $\mathit{length}\ xs = n$. This is the mode $o \to o \to \mathit{bool}$.

In the classic analysis, a mode is only possible if the Horn clauses allow a complete data-flow from input to output values. For Quickcheck, if the mode analysis fails to produce a complete mode assignment because the values of some variables are not constrained by the premises, we fall back on the random or exhaustive strategy to fill in the gaps in the data flow. For example, given the Horn clause $P\ x \Longrightarrow Q\ x\ y$, where P

supports the modes $i \to \mathit{bool}$ and $o \to \mathit{bool}$, the classic analysis fails to find a consistent mode assignment for Q with mode $o \to o \to \mathit{bool}$ because y is unconstrained. To generate values for x and y that fulfill Q, we can generate x values using P with $o \to \mathit{bool}$ and set y to an arbitrary value, then check $Q\ x\ y$.

If the conjecture is polymorphic, we can instantiate the type variables with any concrete type for refuting it. Older versions of Quickcheck instantiated type variables with the type of integers, but it is usually preferable to use a small finite type instead, so that existential conjectures $\exists x :: \alpha.\ P\ x$ can be refuted by a finite number of P tests.

4.3 Narrowing

The random and exhaustive strategies suffer from two important limitations: They cannot refute propositions that existentially quantify over infinite types, and they often repeatedly test formulas with values that check essentially the same execution (e.g., because of symmetries).

Both issues arise from the use of ground values and can be addressed by evaluating the formula symbolically. The technique is called narrowing and is well known from term rewriting. The main idea is to evaluate the conjecture with partially instantiated terms and to progressively refine these terms as needed. Technically, this can be achieved in at least three different ways:

1. Target a language that natively supports narrowing, such as the functional-logical language Curry [1], instead of ML.
2. Simulate narrowing by generating a functional program that includes its own refinement algorithm [46].
3. Simulate narrowing by embedding the narrowing-based execution with a library of combinators [18, 30] in a functional language.

We tried out the first two approaches and found that the second approach is faster. The third approach looks promising but would require a more involved translation.

The main benefit of narrowing is its generality: Unlike the random and exhaustive strategies, it can refute existential quantifications over infinite types. Consider the following conjecture:

$$\forall n.\ \exists m :: \mathit{nat}.\ n = \mathit{Suc}\ m$$

To disprove it, we must exhibit a natural number n such that $\forall m :: \mathit{nat}.\ n \neq \mathit{Suc}\ m$. Taking a symbolic view, if we choose $n = 0$, it is easy to see that $n \neq \mathit{Suc}\ m$ is true for every natural number m without having to instantiate m.

The above example is perhaps too simple to be convincing. A more realistic example is based on the observation that the palindrome $[a, b, b, a]$ can be split into the list $[a, b]$ and its reverse $[b, a]$. Generalizing this to arbitrary lists, we boldly conjecture that

$$\mathit{rev}\ \mathit{xs} = \mathit{xs} \Longrightarrow \exists \mathit{ys}.\ \mathit{xs} = \mathit{ys}\ @\ \mathit{rev}\ \mathit{ys}$$

The narrowing approach immediately finds the counterexample $\mathit{xs} = [\mathsf{a}_1]$, inferring that there is no witness for ys in the infinite domain of lists: If ys is empty, $\mathit{ys}\ @\ \mathit{rev}\ \mathit{ys} =$

$[] \neq [\mathsf{a}_1]$, and if *ys* is not empty, *ys* @ *rev ys* consists of at least two elements and hence cannot be equal to $[\mathsf{a}_1]$.

Narrowing tends to scale better than the random and exhaustive strategies. Consider red–black trees, a binary search data structure with two kinds of node, red and black, that must satisfy a sophisticated invariant involving node coloring. The invariant is captured by a predicate *is_rbt*. If the *delete* operation is properly implemented, the following property should hold:

$$\mathit{is\_rbt}\ t \Longrightarrow \mathit{is\_rbt}\ (\mathit{delete}\ k\ t)$$

The premise *is_rbt t* ensures that the tree *t* has a black root node, and in fact, after a few refinements, narrowing will only test symbolic values satisfying this property, already pruning away about half of the overall test cases. As expected, narrowing finds many more counterexamples than random and exhaustive testing on this kind of example. Interestingly, it even performs slightly better than a custom generator that constructs well-formed trees using a sequence of *insert* operations.

5 Nitpick: Countermodel Generation Using SAT Solvers

Irrespective of which strategy is used, Quickcheck recasts the conjecture to disprove into a functional program. An alternative is to let a SAT solver enumerate models of the negated conjecture and relevant definitions and axioms. This approach is implemented in a separate tool called Nitpick [8], which relies on the highly optimized Kodkod library [53] for the actual reduction to SAT.

Given a conjecture, Nitpick (via Kodkod and the SAT solver) searches for a standard set-theoretic model that falsifies it while satisfying any relevant axioms and definitions. Unlike Quickcheck, which performs its sophisticated code transformations using the Isabelle inference kernel, Nitpick does not certify any of its results and must be trusted.

Nitpick's design was inspired by its predecessor Refute [54], which performed a direct reduction to SAT. Nitpick works by systematically enumerating the domain cardinalities for the atomic types (type variables and other uninterpreted types) occurring in the conjecture and generates one Kodkod problem (and ultimately one SAT problem) per cardinality specification [5]. To exhaust all models up to a given cardinality bound k for a formula involving n atomic types, it must in principle iterate through k^n combinations of cardinalities, but a sophisticated monotonicity inference helps prune the search space [7]. If the conjecture has a finite countermodel, the tool eventually finds it, unless it runs out of resources.

5.1 Basic Translation to Relational Logic

Kodkod's input is expressed in first-order relational logic (FORL), an idiosyncratic formalism that combines elements from first-order logic and relational calculus, extended with a transitive closure operator. SAT solvers are particularly sensitive to the encoding of problems, so special care is needed when translating HOL formulas to FORL. Whenever practicable, HOL constants are mapped to their FORL equivalents, rather than expanded to their definitions.

As a rule, HOL scalars are mapped to FORL singletons and functions are mapped to FORL relations accompanied by a constraint. An n-ary first-order function can be coded as an $(n+1)$-ary relation accompanied by a constraint. However, if the return type is *bool*, the function is more efficiently coded as an unconstrained n-ary relation. This allows formulas such as $A^+ \cup B^+ = (A \cup B)^+$ to be translated without taking a detour through ternary relations.

Higher-order quantification and functions bring complications of their own. For example, assuming the cardinality constraints $|\alpha| = 2$ and $|\beta| = 3$, we would like to translate $\forall g :: \beta \to \alpha.\ g\,x \neq y$ into something like

$$\forall g \subseteq \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\} \times \{\mathsf{a}_1, \mathsf{a}_2\}.\ (\forall a \in \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\}.\ |g(a)| = 1) \longrightarrow g(x) \neq y$$

but since Kodkod is first-order, the $\subseteq$ symbol is not allowed at the binding site; only $\in$ is. Skolemization solves half the problem, but for the remaining quantifiers we are forced to adopt an unwieldy n-tuple singleton representation of functions, where n is the cardinality of the domain. The n-tuple simply encodes g's function table. For the formula above, this gives

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

where G is the triple corresponding to g and $\pi_i(G)$ is its ith component (i.e., the ith entry in the function table). In the body, we convert the singleton G to the relational representation, then we apply x on it. The singleton encoding is also used for passing functions to other functions; fortunately, two optimizations, function specialization and boxing [8, §5], make this rarely necessary.

5.2 Approximation of Infinite Types and Partiality

Because of the axiom of infinity, the type *nat* of natural numbers does not admit any finite models. To work around this, Nitpick considers finite subsets $\{0, 1, \ldots, K-1\}$ of *nat* and maps numbers $\geq K$ to the undefined value, denoted by $\star$ and coded as the empty set. Formulas of the form $\forall n :: nat.\ P\,n$ are treated as $(\forall n < K.\ P\,n) \wedge P\,\star$, which usually evaluates to either *False* (if $P\,i$ gives *False* for some $i < K$) or $\star$, but not to *True*, since we generally cannot determine statically whether $P\,K$, $P\,(K+1), \ldots$, collectively represented by $P\,\star$, are true. Partiality leads to a Kleene three-valued logic, which is soundly encoded in Kodkod's two-valued logic.

5.3 Encoding of (Co)inductive Predicates

Isabelle lets users specify (co)inductive predicates p by their introduction rules and synthesizes a fixed point definition $p = \mathit{lfp}\ F$ or $p = \mathit{gfp}\ F$ behind the scenes. For performance reasons, Nitpick handles (co)inductive predicates specially rather than simply expanding *lfp* and *gfp* to their definitions.

An inductive predicate p is a fixed point, so Nitpick can use the equation $p = F\,p$ as the axiomatic specification of p. In general, this is unsound since it underspecifies p, but there are two important cases for which this method is sound:

– If the recursion in F is well-founded [12], the fixed point equation $p = F\,p$ admits exactly one solution that can safely be taken as p's specification.
– If p occurs negatively in the formula, these occurrences can be soundly replaced by a fresh constant q satisfying the axiom $q = F\,q$.

For the remaining positive occurrences of p, Nitpick unrolls the predicate a given number of times, as in bounded model checking [4]. The situation is mirrored for coinductive predicates: Positive occurrences are coded using the fixed-point equation, and negative occurrences are unrolled.

5.4 Encoding of (Co)inductive Datatypes

In contrast to Isabelle's constructor-oriented treatment of inductive datatypes, Nitpick's FORL axiomatization revolves around selectors and discriminators, following a standard Alloy idiom [28]. The selector/discriminator view is usually more efficient than the constructor view because it breaks high-arity constructors into several low-arity selectors, with correspondingly smaller function tables in the SAT encoding. For example, the type α *list* generated from *Nil* :: α *list* and *Cons* :: $\alpha \to \alpha$ *list* $\to \alpha$ *list* is axiomatized in terms of the discriminators *nilp* and *consp* and the selectors *hd* and *tl*, which give access to a nonempty list's head and tail.

The FORL axiomatization specifies a subterm-closed finite substructure of lists. Examples of subterm-closed list substructures using traditional notation are $\{[], [0], [1]\}$ and $\{[], [1], [2,1], [0,2,1]\}$. On the other hand, the set $L = \{[], [1,1]\}$ is not subterm-closed, because *tl* $[1,1] = [1] \notin L$. Given cardinalities for the list type and the item type, the SAT solver enumerates all corresponding subterm-closed list substructures.

Nitpick supports coinductive datatypes, even though Isabelle does not provide a high-level mechanism for defining them. Users can define custom coinductive datatypes from first principles and tell Nitpick to substitute its efficient FORL axiomatization for their definitions.

6 Related Work

Isabelle is not the only interactive theorem prover that provides a palette of automatic proof and disproof tools. We briefly review what the other popular provers have to offer.

– HOL4 [20, 49] includes the original version of Metis [24] and an integration of SMT solvers [55] with proof reconstruction for Z3 [10].
– PVS includes a Quickcheck-like random testing tool [36] and integrates the SMT solver Yices as an oracle [47].
– For Mizar, the Miz$\mathbb{AR}$ web service [45] is a recent addition that exploits external resolution provers in the style of Sledgehammer.
– The Sedan version of ACL2 includes a counterexample generator based on random testing [13]. The tool analyses the goal to compute dependencies between free variables, similar to Quickcheck's data-flow analysis.
– Although Coq has a considerable user base, advisory tools are conspicuously missing. An SMT integration with proof certification is in the works [27].

- Earlier versions of the Agda proof assistant included a version of QuickCheck [17], but like the original QuickCheck for Haskell it required users to write dedicated data generators for custom datatypes. The Agsy tool [29, 30] implements narrowing for both counterexample generation and proof search. An integration of the equational prover Waldmeister is under development [19].

7 Conclusion

Isabelle offers a wide range of automatic tools for proving and disproving conjectures. Some of them are built into the theorem prover, but increasingly these activities are delegated to highly optimized external tools, such as resolution provers, SAT solvers, and SMT solvers. While there have been several attempts at integrating external provers and disprovers in various interactive theorem provers, Isabelle is probably the only interactive prover where external tools play such a prominent role, to the extent that they are now seen as indispensable by many if not most users.

In terms of usefulness, Sledgehammer is second only to the simplifier and tableau prover. But the counterexample generators also provide invaluable help and encourage a lightweight explorative style to formal proof development, as championed by Alloy [25]. Because it is so fast, Quickcheck is enabled by default to run on all conjectures. Users are so accustomed to its feedback that they rarely realize to what extent they benefit from it. Every now and then, Nitpick finds a counterexample beyond Quickcheck's reach. As developers of both tools, we frequently receive emails from users grateful to have been spared "several hours of hard work."

An explanation for Sledgehammer, Quickcheck, and Nitpick's success is that they are included with Isabelle and require no additional installation steps. External tools necessary to their operation are either included in the official Isabelle packages or accessible as online services. Multi-core architectures and remote servers help to bear the burden of (dis)proof, so that users can continue working on a manual proof while the tools run in the background.

Another important design goal for all three tools was one-click invocation. Users should not need to preprocess the goals, specify options, or implement custom data generators. Even better than one-click invocation is zero-click invocation, whereby the tools spontaneously run on newly entered conjectures. A more flexible user interface, such as the experimental jEdit-based PIDE [58], could help further here, by asynchronously dispatching the tools to tackle any unfinished proofs in the current proof document, irrespective of the text cursor's location.

Interactive theorem proving is still challenging, but thanks to a new generation of automatic proof and disproof tools and the wide availability of multi-core processors with spare CPU cycles, it is much easier and more enjoyable now than it was only a few years ago.

Acknowledgment. We thank Alexander Krauss, Mark Summerfield, and Thomas Türk for suggesting several textual improvements.

References

1. Antoy, S., Hanus, M.: Functional logic programming. Commun. ACM 53, 74–85 (2010)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer (2007)
3. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) SEFM 2004. pp. 230–239. IEEE C.S. (2004)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) TACAS '99. LNCS, vol. 1579, pp. 193–207. Springer (1999)
5. Blanchette, J.C.: Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. Softw. Qual. J. To appear
6. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, Springer, to appear
7. Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. J. Auto. Reas. To appear
8. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer (2010)
9. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNAI, vol. 6173, pp. 107–121. Springer (2010)
10. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer (2010)
11. Bulwahn, L.: Smart test data generators via logic programming. In: Gallagher, J.P., Gelfond, M. (eds.) ICLP '11 (Technical Communications). Leibniz International Proceedings in Informatics, vol. 11, pp. 139–150. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2011)
12. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer (2007)
13. Chamarthi, H.R., Dillinger, P., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving (2011), available at http://arxiv.org/pdf/1105.4394
14. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP '00. pp. 268–279. ACM (2000)
15. Claessen, K., Lillieström, A., Smallbone, N.: Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, Springer, to appear
16. Dutertre, B., de Moura, L.: The Yices SMT solver (2006), available at http://yices.csl.sri.com/tool-paper.pdf
17. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: Basin, D.A., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 188–203. Springer (2003)
18. Fischer, S., Kiselyov, O., Shan, C.: Purely functional lazy non-deterministic programming. In: ICFP '09. pp. 11–22. ACM (2009)
19. Foster, S., Struth, G.: Integrating an automated theorem prover into Agda. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 116–130 (2011)
20. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
21. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer (1979)

22. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010)
23. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, Springer, to appear
24. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Di Vito, B., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics. pp. 56–68. No. CP-2003-212448 in NASA Technical Reports (2003)
25. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
26. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer (2000)
27. Keller, C.: Cooperation between SAT, SMT provers and Coq
28. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H.C. (ed.) ESEC/FSE '05 (2005)
29. Lindblad, F.: Higher-order proof construction based on first-order narrowing. Electr. Notes Theor. Comput. Sci. 196, 69–84 (2008)
30. Lindblad, F.: Property directed generation of first-order test data. In: Morazán, M. (ed.) TFP 2007. pp. 105–123. Intellect (2008)
31. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Auto. Reas. 40(1), 35–60 (2008)
32. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
33. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
34. Nipkow, T.: A tutorial introduction to structured Isar proofs (2011), available at http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf
35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
36. Owre, S.: Random testing in PVS. In: AFM '06 (2006)
37. Paulson, L.C.: Set theory for verification: I. From foundations to functions. J. Auto. Reas. 11(3), 353–389 (1993)
38. Paulson, L.C.: Set theory for verification: II. Induction and recursion. J. Auto. Reas. 15(2), 167–215 (1995)
39. Paulson, L.C.: Generic automatic proof tools. In: Veroff, R. (ed.) Automated Reasoning and its Applications: Essays in Honor of Larry Wos, pp. 23–47. MIT Press (1997)
40. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. J. Univ. Comp. Sci. 5(3), 73–87 (1999)
41. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) IWIL-2010 (2010)
42. Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 232–245 (2007)
43. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2 (2006), available at http://goedel.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf
44. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. 15(2–3), 91–110 (2002)
45. Rudnicki, P., Urban, J.: Escape to ATP for Mizar. In: PxTP-2011 (2011)
46. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In: Haskell Symposium '08. pp. 37–48. ACM (2008)
47. Rushby, J.M.: Tutorial: Automated formal methods with PVS, SAL, and Yices. In: Hung, D.V., Pandya, P. (eds.) SEFM 2006. p. 262. IEEE (2006)

48. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNAI, vol. 3097, pp. 223–228. Springer (2004)
49. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32 (2008)
50. Sutcliffe, G.: System description: SystemOnTPTP. In: McAllester, D. (ed.) CADE-17. LNAI, vol. 1831, pp. 406–410. Springer (2000)
51. Sutcliffe, G.: The CADE-21 automated theorem proving system competition. AI Commun. 21(1), 71–82 (2008)
52. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP data-exchange formats for automated theorem proving tools. In: Zhang, W., Sorge, V. (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Frontiers in Artificial Intelligence and Applications, vol. 112, pp. 201–215. IOS Press (2004)
53. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer (2007)
54. Weber, T.: SAT-Based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Dept. of Informatics, T.U. München (2008)
55. Weber, T.: SMT solvers: New oracles for the HOL theorem prover. In: VSTTE 2009 (2009)
56. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. pp. 1965–2013. Elsevier (2001)
57. Wenzel, M.: Isabelle/Isar—a generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof—Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar and Rhetoric, vol. 10(23). University of Białystok (2007)
58. Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In: Coen, C.S., Aspinall, D. (eds.) UITP '10 (2010)
59. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A. (eds.) TPHOLs '97. LNCS, vol. 1275, pp. 307–322 (1997)