More SPASS with Isabelle

Superposition with Hard Sorts and Configurable Simplification

Jasmin Christian Blanchette[1], Andrei Popescu[1],
Daniel Wand[2], and Christoph Weidenbach[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. Sledgehammer for Isabelle/HOL integrates automatic theorem provers to discharge interactive proof obligations. This paper considers a tighter integration of the superposition prover SPASS to increase Sledgehammer's success rate. The main enhancements are native support for hard sorts (simple types) in SPASS, simplification that honors the orientation of Isabelle *simp* rules, and a pair of clause-selection strategies targeted at large lemma libraries. The usefulness of this integration is confirmed by an evaluation on a vast benchmark suite and by a case study featuring a formalization of language-based security.

1 Introduction

The interactive theorem proving community has traditionally put more emphasis on trustworthiness, expressiveness, and flexibility than on raw deductive power. Automation in proof assistants typically takes the form of general-purpose proof methods or tactics, complemented by decision procedures for specific domains. Recent large-scale efforts such as the proofs of the four-color theorem [16], of a C compiler [23], and of a microkernel [20] have highlighted the need for more automation [24].

There have been many attempts at harnessing decades of research in automated reasoning by integrating automatic theorem provers in proof assistants. The most successful integration is undoubtedly Sledgehammer for Isabelle/HOL (Sects. 2.1 and 2.2). The tool invokes several first-order automatic provers in parallel, both superposition-based provers and SMT solvers [5], and reconstructs their proofs in Isabelle. In an evaluation on a representative corpus of older formalizations, Sledgehammer discharged 43% of the goals that could not be solved trivially using an existing Isabelle proof method [5].

Sledgehammer's usefulness is regularly confirmed by users; Guttmann et al. [17] relied almost exclusively on it to derive over 1000 propositions relating to relational and algebraic methods for modeling computing systems. Yet, there are many indications that more can be done. Integrated verification tool chains such as VCC/Boogie/Z3 [12] claim much higher success rates, typically well above 90%. Isabelle goals are certainly more diverse than verification conditions for a fixed programming language, which makes fine-tuning less practicable; however, typical Isabelle goals are not necessarily more difficult than typical verification conditions. Another indicator that Sledgehammer can be significantly improved is that it solves only about 80% of the goals that standard proof methods (such as *simp*, *auto*, and *blast*) solve trivially [5]. This lackluster performance points to weaknesses both in Sledgehammer's translation of higher-order constructs and in the automatic provers themselves.

What is easy in a proof assistant can be surprisingly difficult for first-order automatic theorem provers. For example, Isabelle's simplifier can easily prove goals of the form rev $[x_1, \ldots, x_n] = [x_n, \ldots, x_1]$ (where rev is list reversal) by applying equations as oriented rewrite rules, or *simp* rules. The presence of hundreds of registered *simp* rules hardly slows it down. In contrast, superposition provers such as E [38], SPASS [44], and Vampire [34] perform an unconstrained search involving the supplied background theory. They can use equations as rewrite rules but must often reorient them to obey a specific term ordering. In exchange, these provers are complete for classical first-order logic: Given enough resources, they eventually find a (first-order) proof if one exists.

Much work went into making Sledgehammer's translation from higher-order logic as efficient as possible [6, 27]. However fruitful this research may have been, it appears to have reached a plateau. To achieve higher success rates, a new approach is called for: Implement the features that make proof search in Isabelle successful directly in an automatic prover. By combining the best of both worlds, we target problems that cannot be solved by either tool on its own.

Our vehicle is SPASS (Sect. 2.3), a widely used prover based on a superposition calculus (a generalization of resolution). It is among the best performing automatic provers, and one of the few whose development is primarily driven by applications, including mission-critical computations in industry. Its source code is freely available and well structured enough to form a suitable basis for further development. In the automated reasoning community, SPASS is well known for its soft sorts, which can comfortably accommodate many-sorted, order-sorted, and membership logics, its integrated splitting rule, employing competitive decision procedures for various decidable first-order fragments, and its sophisticated simplification machinery.

This paper describes the first part of our work program. It focuses on three aspects.

- *Hard sorts* (Sect. 3): Soft sorts are overly general for most applications. By supporting a more restrictive many-sorted logic in SPASS and combining it with monomorphization, we get a sound and highly efficient representation of HOL types, without the spurious unreconstructible proofs that have long plagued Sledgehammer users.
- *Configurable simplification* (Sect. 4): Superposition provers reorient the equations in the problem to make the right-hand sides smaller than the left-hand sides with respect to a term ordering. *Advisory* simplification adjusts the ordering to preserve the orientation of *simp* rules as much as possible; *mandatory* simplification forcefully achieves rewriting against the term ordering if necessary.
- *Clause selection for large theories* (Sect. 5): Sledgehammer problems include hundreds of lemmas provided as axioms, sorted by likely relevance. It makes sense for SPASS to focus on the most likely relevant lemmas, rather than hopelessly try to saturate the entire background theory.

A case study demonstrates the SPASS integration on a formalization pertaining to language-based security (Sect. 6), an area that calls for strong automation in combination with the convenience of a modern proof assistant. The new features are evaluated both in isolation and against other popular automatic theorem provers (Sect. 7).

2 Background

2.1 Isabelle/HOL

Isabelle/HOL [30] is a proof assistant based on classical higher-order logic (HOL) extended with rank-1 polymorphism and axiomatic type classes. The term language consists of simply-typed λ-terms augmented with constants (of scalar or function types) and polymorphic types. Function application expects no parentheses around the arguments (e.g., $f\ x\ y$); familiar operators are written in infix notation. Functions may be partially applied (curried), and variables may range over functions. Types can optionally be attached to a term using an annotation $t : \sigma$ to guide type inference.

The dominant general-purpose proof method is the simplifier, which applies equations as oriented rewrite rules to rewrite the goal. It performs conditional, contextual rewriting with hooks for customizations based on the vast library of registered *simp* rules. At the user level, the simplifier is upstaged by *auto*, a method that interleaves simplification with proof search. Other commonly used methods are the tableau prover (*blast*) and the arithmetic decision procedures (*linarith* and *presburger*).

2.2 Sledgehammer

Sledgehammer [31] harnesses the power of superposition provers and SMT solvers. Given a conjecture, it heuristically selects a few hundred facts (lemmas, definitions, or axioms) from Isabelle's libraries [28], translates them to first-order logic along with the conjecture, and delegates the proof search to external provers—by default, E [38], Vampire [34], Z3 [29], and of course SPASS [44]. Because automated deduction involves a large share of heuristic search, the combination of provers is much more effective than any single one of them. Proof reconstruction relies on the built-in resolution prover *metis* [19, 32] and the Z3-based *smt* proof method [11].

Given that automatic provers are very sensitive to the encoding of problems, the translation from higher-order logic to unsorted first-order logic used for E, SPASS, and Vampire is a crucial aspect of Sledgehammer. It involves two steps [27]:

1. *Eliminate the higher-order features of the problem.* Curried functions are passed varying numbers of arguments using a deeply embedded application operator, and λ-abstractions are rewritten to SK combinators or supercombinators (λ-lifting).
2. *Encode polymorphic types and type classes.* Type information can be encoded in a number of ways. Traditionally, it has been supplied as explicit type arguments to the function and predicate symbols corresponding to HOL constants. In conjunction with Horn clauses representing the type class hierarchy, this suffices to enforce correct type class reasoning and overload resolution, but not to prevent ill-typed variable instantiations. Unsound proofs are discarded at reconstruction time.

Example 1. In the recursive specification of map on lists, the variable $f : \alpha \to \beta$ is higher-order both in virtue of its function type and because it occurs partially applied:

$$\mathsf{map}\ f\ \mathsf{Nil} = \mathsf{Nil}$$
$$\mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

Step 1 translates the second equation to

$$\mathsf{map}(F, \mathsf{cons}(X, Xs)) = \mathsf{cons}(\mathsf{app}(F, X), \mathsf{map}(F, Xs))$$

where $F : \mathit{fun}(\alpha, \beta)$ is a deeply embedded function (or "array") and app is the embedded application operator (or "array read").[1] Step 2 introduces type arguments encoded as terms, with term variables A, B for α, β:

$$\mathsf{map}(A, B, F, \mathsf{cons}(A, X, Xs)) = \mathsf{cons}(B, \mathsf{app}(A, B, F, X), \mathsf{map}(A, B, F, Xs))$$

2.3 SPASS

SPASS (= Spaß = "fun" in German) is an implementation of the superposition calculus with various refinements, including unique support for soft (monadic) sorts and splitting [15,43,44]. It is a semi-decision procedure for classical first-order logic and a decision procedure for various first-order logic fragments.

The input is a list of axioms and a conjecture expressed either in the TPTP FOF syntax [40] or in a custom syntax called DFG. SPASS outputs either a proof (a derivation of the empty, contradictory clause from the axioms and the negated conjecture) or a saturation (an exhaustive list of all normalized clauses that can be derived); it may also diverge for unprovable problems with no finite saturation.

Well-founded term orderings are crucial to the success of the superposition calculus. For example, from the pair of clauses $\mathsf{p}(\mathsf{a})$ and $\neg\,\mathsf{p}(X) \vee \mathsf{p}(\mathsf{f}(X))$, resolution will derive infinitely many facts of the form $\mathsf{p}(\mathsf{f}^i(\mathsf{a}))$, whereas for superposition $\mathsf{p}(\mathsf{f}(X))$ is maximal and no inferences can be performed. Nonetheless, superposition-based reasoning is very inefficient when combined with order-sorted signatures, because completeness requires superposition into variables, which dramatically increases the search space. Soft sorts were designed to remedy this problem: When they are enabled, SPASS views every monadic predicate as a sort and applies optimized inference and simplification rules [15]. Monadic predicates can be used to emulate a wide range of type systems.

3 Hard Sorts

After eliminating the higher-order features of a problem, Sledgehammer is left with first-order formulas in which Isabelle's polymorphism and axiomatic type classes still occupy a prominent place. The type argument scheme presented in Section 2.2 is unsound, and the traditional sound encodings of polymorphic types introduce too much clutter to be useful [6, 10, 27]. This state of affairs is unsatisfactory. Even with proof reconstruction, there are major drawbacks to unsound type encodings.

First, finite exhaustion rules of the form $x = \mathsf{c}_1 \vee \cdots \vee x = \mathsf{c}_n$ or $(x = \mathsf{c}_1 \Longrightarrow P) \Longrightarrow \cdots \Longrightarrow (x = \mathsf{c}_n \Longrightarrow P) \Longrightarrow P$ must be left out because they force an upper bound on the cardinality of the universe, rapidly leading to unsound cardinality reasoning; for example, automatic provers can easily derive a contradiction from $(u : \mathit{unit}) = ()$ and $(0 : \mathit{nat}) \neq \mathsf{Suc}\ n$ if type information is simply omitted. The inability to encode such rules prevents the discovery of proofs by case analysis on finite types.

[1] Following a common convention in the automated reasoning and logic programming communities, we start first-order variable names with an upper-case letter, keeping lower-case for function and predicate symbols. Nullary functions (constants) are written without parentheses.

Second, spurious proofs are distracting and sometimes conceal more difficult sound proofs. Users eventually learn to recognize facts that lead to unsound reasoning and mark them with a special attribute to remove them from the scope of Sledgehammer's relevance filter, but this remains a stumbling block for novices.

Third, it would be desirable to let SPASS itself perform relevance filtering, or even use a sophisticated system based on machine learning, where successful proofs guide subsequent ones. However, such approaches tend to quickly detect and exploit contradictions in the large translated axiom set if type information is omitted.

How can we provide SPASS with the necessary type information in a sound, complete, and efficient manner? The original plan was to exploit SPASS's soft sorts, by monomorphizing the problem (i.e., heuristically instantiating the type variables with ground types) and inserting monadic predicates, or guards, $\mathsf{p}_\sigma(X)$ to ensure that a given variable X has type σ. Following this scheme, the *nat* → *int* instance of the second equation in Example 1 would be translated to the unsorted formula

$$\begin{aligned}&\mathsf{p}_{\mathsf{fun(nat,\,int)}}(F) \land \mathsf{p}_{\mathsf{nat}}(X) \land \mathsf{p}_{\mathsf{list(nat)}}(Xs) \longrightarrow\\ &\qquad \mathsf{map}_{\mathsf{nat,\,int}}(F, \mathsf{cons}_{\mathsf{nat}}(X, Xs)) = \mathsf{cons}_{\mathsf{int}}(\mathsf{app}_{\mathsf{nat,\,int}}(F, X), \mathsf{map}_{\mathsf{nat,\,int}}(F, Xs))\end{aligned}$$

where subscripts distinguish instances of polymorphic symbols. The guards are discharged by deeply embedded typing rules for the function symbols occurring in the problem. SPASS views each p_σ predicate as a sort.

Monomorphization is necessarily incomplete [9, §2] and often dismissed because it quickly leads to an explosion in the number of formulas. Nonetheless, with suitable bounds on the number of monomorphic instances generated, our experience is that it vastly outperforms complete encodings of polymorphism [6]. It also relieves SPASS of having to reason about type classes—only the monomorphizer needs to consider them.

The outcome of experiments with SPASS quickly dashed our hopes: Sure enough, soft sorts were helping SPASS, but the resulting encoding was still no match for the unsound scheme based on type arguments. Explicit typing requires a particular form of contextual rewriting to simulate typed rewriting efficiently. The needed mechanisms are not available in any of today's first-order theorem provers, not even in SPASS's soft typing machinery. For example, consider a constant a of sort σ and an unconditional equation $\mathsf{f}(X) = X$ where $X : \sigma$. Sorted rewriting transforms $\mathsf{f}(\mathsf{a})$ into a in one step. In contrast, the soft typing version of the example is a conditional equation $\mathsf{p}_\sigma(X) \longrightarrow \mathsf{f}(X) = X$ and the typing axiom $\mathsf{p}_\sigma(\mathsf{a})$. Rewriting $\mathsf{f}(\mathsf{a})$ requires showing $\mathsf{p}_\sigma(\mathsf{a})$ to discharge the condition in the instance $\mathsf{p}_\sigma(\mathsf{a}) \longrightarrow \mathsf{f}(\mathsf{a}) = \mathsf{a}$.

We came up with a new plan: Provide *hard* sorts directly in SPASS, orthogonally to soft sorts. Hard sorts can be checked directly to detect type mismatches early and avoid ill-sorted inferences. We focused on monomorphic sorts, which require no matching or unification. The resulting many-sorted first-order logic corresponds to that offered by the TPTP TFF0 format [42]. Polymorphism is eliminated by monomorphization.[2]

[2] For future work, we want to extend the hard sorts to ML-style polymorphism as provided by Alt-Ergo [8] and TPTP TFF1 [7]. Besides the expected performance benefits [14], this is a necessary step toward mirroring Isabelle's hierarchical theory structure on the SPASS side: Once theories are known to SPASS, they can be preprocessed (e.g., finitely saturated) and reused, which would greatly speed up subsequent proof searches.

Although superposition with hard sorts is well understood, adding sorts to a highly optimized theorem prover is a tedious task. Predicate and function symbols must be declared with sort signatures. Variables must carry sorts, and unification must respect them. The term index that underlies most inference rules must take sort constraints into consideration; the remaining rules must be adapted individually. There are many other technical aspects related to variable renaming, skolemization, and of course parsing and printing of formulas. We made all these changes and found that hard sorts are much more efficient than their soft cousins, and even than the traditional unsound scheme, which we so desperately wanted to abolish.

In a fortuitous turn of events, a group of researchers including the first author recently discovered a lightweight yet sound guard-based encoding as well as many variants [6]. These are now implemented in Sledgehammer. They work well in practice but fall short of outperforming hard sorts. Moreover, hard sorts are more suitable for applications that require not only soundness of the overall result but also type-correctness of the individual inferences, such as step-by-step proof replay [32].

4 Configurable Simplification

The superposition calculus is parameterized by a well-founded total ordering on ground terms. Like most other provers, SPASS employs the Knuth–Bendix ordering [22], which itself is determined by a weight function w from symbols to $\mathbb{N}$ and a total precedence order $\prec$ on function symbols. Weights are lifted to terms by taking the sum of the weights of the symbols that occur in it, counting duplicates.

Let $s = \mathsf{f}(s_1, \ldots, s_m)$ and $t = \mathsf{g}(t_1, \ldots, t_n)$ be two terms. The (*basic*) *Knuth–Bendix ordering* (*KBO*) induced by $(w, \prec)$ is the relation $\prec$ such that $s \prec t$ if and only if for any variable occurring in s it has at least as many occurrences in t and a, b, or c is satisfied:

a. $w(s) < w(t)$;
b. $w(s) = w(t)$ and $\mathsf{f} \prec \mathsf{g}$;
c. $w(s) = w(t)$, $\mathsf{f} = \mathsf{g}$, and there exists i such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$, and $s_i \prec t_i$.

Assuming w and $\prec$ meet basic requirements, the corresponding KBO is a well-founded total order on ground terms that embeds the subterm relation and is stable under substitution, as required by superposition. The main proviso for the application of an equation $l = r$ to simplify a clause is that l must be larger than r with respect to the given KBO.

By default, SPASS simply assigns a weight of 1 to every symbol and heuristically selects a precedence order. Then it reviews each equation $l = r$ in the light of the induced KBO to determine whether $l = r$, $r = l$, or neither of them can be applied as a left-to-right rewrite rule to simplify terms. Since the left-hand side of an Isabelle definition tends to be smaller than the right-hand side, SPASS will often reorient definitions, making it much more difficult to derive long chains of equational reasoning.

Intuitively, a better strategy would be to select a weight function and a precedence order that maximize the number of definitions and *simp* rules that SPASS can use for simplification with their original orientation. For example, to keep the equation

$$\mathsf{shift}(\mathsf{cons}(X, Xs)) = \mathsf{append}(Xs, \mathsf{cons}(X, \mathsf{nil}))$$

oriented, SPASS could take $w(\mathsf{shift}) \geq 3$ (or even $w(\mathsf{shift}) = 2$ with $\mathsf{append} \prec \mathsf{shift}$) while setting $w(\mathsf{nil}) = w(\mathsf{cons}) = w(\mathsf{append}) = 1$; the occurrences of X and Xs on either side cancel each other out. The weight function normally plays a greater role than the precedence order, but for some equations precedence is needed to break a tie—for example:

$$\mathsf{append}(\mathsf{cons}(X, Xs), Ys) = \mathsf{cons}(X, \mathsf{append}(Xs, Ys))$$

Our approach for computing a suitable weight function w is to build a dependency graph, in which edges $\mathsf{f} \leftarrow \mathsf{g}$ indicate that f is simpler than g. The procedure first considers definitional equations of the form $\mathsf{f}(s_1, \ldots, s_m) = t$, including simple definitions and equational specifications of recursive functions, and adds edges $\mathsf{f} \leftarrow \mathsf{g}$ for each symbol g that occurs in s_1, $\ldots$, s_m, or t, omitting any edge that would complete a cycle (which may happen if f is recursive through g). In a second step, *simp* rules are considered in the same way to further enrich the graph. The cycle-detection mechanism is robust enough to cope with nondefinitional lemmas such as $\mathsf{rev}(\mathsf{rev}(Xs)) = Xs$.

Once the graph is built, the procedure assigns weight 2^{d+1} to symbols with depth d and uses a topological order for symbol precedence. For example, given the usual recursive definitions of append (in terms of nil and cons) and rev (in terms of append, nil, and cons), it computes $w(\mathsf{nil}) = w(\mathsf{cons}) = 1$, $w(\mathsf{append}) = 2$, $w(\mathsf{rev}) = 4$, and either $\mathsf{nil} \prec \mathsf{cons} \prec \mathsf{append} \prec \mathsf{rev}$ or $\mathsf{cons} \prec \mathsf{nil} \prec \mathsf{append} \prec \mathsf{rev}$ for the precedence. With these choices of w and $\prec$, SPASS proves $\mathsf{rev}\ [x_1, \ldots, x_n] = [x_n, \ldots, x_1]$ in no time even for large values of n (e.g., 50) and in the presence of hundreds of axioms, whereas the other automatic provers time out.

Regrettably, there are many equations that cannot be oriented in the desired way with this approach. KBO cannot orient an equation such as

$$\mathsf{map}(F, \mathsf{cons}(X, Xs)) = \mathsf{cons}(\mathsf{app}(F, X), \mathsf{map}(F, Xs))$$

in a left-to-right fashion because of the two occurrences of F on the right-hand side. It will also fail with

$$\mathsf{rev}(\mathsf{cons}(X, Xs)) = \mathsf{append}(\mathsf{rev}(Xs), \mathsf{cons}(X, \mathsf{nil}))$$

because the occurrences of rev and cons on the left-hand side are canceled out by those on the right-hand side; no matter how heavy we make these, the right-hand side will weigh even more due to append's and nil's contributions.

An especially thorny yet crucial example is the S combinator, defined in HOL as $\lambda x\, y\, z.\ x\ z\ (y\ z)$. It manifests itself in most problems generated by Sledgehammer to encode λ-abstractions. In first-order logic, it is specified by the axiom

$$\mathsf{app}(\mathsf{app}(\mathsf{app}(\mathsf{s}, X), Y), Z) = \mathsf{app}(\mathsf{app}(X, Z), \mathsf{app}(Y, Z))$$

For simplification, the left-to-right orientation is clearly superior, because it eliminates the combinator whenever the third argument is supplied, emulating β-reduction. Unfortunately, the duplication of Z on the right-hand side makes this orientation incompatible with KBO; in fact, either orientation is incompatible with the subterm condition and substitution stability requirements on admissible term orderings.

All is not lost for equations that cannot be ordered in the natural way. It is possible to extend superposition with controlled simplification against the term ordering. To achieve this, we extended SPASS's input syntax with annotations for both *advisory* simplifications, which only affect the term ordering, and *mandatory* simplifications, which force rewriting against the ordering if necessary.

To avoid infinite looping, the mandatory simplification rules must terminate. Isabelle ensures that simple definitions have acyclic dependencies and recursive function specifications are well-founded, so these can safely be made mandatory. Artifacts of the translation to first-order logic, such as the SK combinators, can also be treated in this way. We could even trust the Isabelle user and make all *simp* rules mandatory, but it is safer to keep the advisory status for these. However, even assuming termination of mandatory simplifications, our implementation is generally incomplete; to ensure completeness, we would need to treat such simplifications as a separate inference rule of the superposition calculus, rather than as a postprocessing step.

Of course, excessive rewriting, especially of the mandatory kind, can give rise to large terms that hamper abstract reasoning. We encountered a striking example of this in the innocuous-looking HOL definitions $\mathsf{Bit0}\ k = k + k$ and $\mathsf{Bit1}\ k = 1 + k + k$, which together with Pls and Min (i.e., 0 and -1) encode signed numerals—for example, '4' is surface syntax for $\mathsf{Bit0}(\mathsf{Bit0}(\mathsf{Bit1}(\mathsf{Pls})))$. Rewriting huge numerals to sums of 1s is obviously detrimental to SPASS's performance, so we disabled mandatory simplification for these two definitions.

5 Clause Selection for Large Theories

Superposition provers work by exhaustively deriving all possible (normalized) clauses from the supplied axioms and the negated conjecture, aiming at deriving the empty clause. New clauses are produced by applying inference rules on already derived pairs of clauses. The order in which clauses are selected to generate further inferences is crucial to a prover's performance and completeness.

At the heart of SPASS is a set of *usable* (passive) clauses $\mathcal{U}$ and a set of *worked-off* (active) clauses $\mathcal{W}$ [43]. The set $\mathcal{U}$ is initialized with the axioms and negated conjecture, whereas $\mathcal{W}$ starts empty. The prover iteratively performs the following steps:

1. Heuristically select a clause C from $\mathcal{U}$.
2. Perform all possible inferences between C and each member of $\mathcal{W} \cup \{C\}$ and insert the resulting clauses into $\mathcal{U}$.
3. Simplify $\mathcal{U}$ and $\mathcal{W}$ using $\mathcal{W} \cup \{C\}$ and move C to $\mathcal{W}$.

A popular variant, the *set of support* (SOS) strategy [45], keeps the search more goal-oriented by initially moving the axioms into the worked-off set rather than into the usable set; only the negated conjecture is considered usable. This disables inferences between axioms, allowing only inferences that directly or indirectly involve the negated conjecture. SOS is complete for resolution but incomplete for superposition. It often terminates very fast, with either a proof or an incomplete saturation. A study found it advantageous for Sledgehammer problems [10, §3].

The heuristic that chooses a usable clause in step 1 is called the *clause-selection strategy*. SPASS's default strategy alternately chooses light clauses (to move toward the empty clause) and shallow clauses (to broaden the search) in a fair way. The *weight* of a clause is the sum of the weights of its terms (cf. Sect. 4); the *depth* is the height of its derivation tree. However, this strategy scales poorly with the number of facts provided by Sledgehammer; beyond about 150 facts (before monomorphization), additional facts harm more than they help. To help SPASS cope better with large theories, we experimented with two custom strategies.

The *goal-based* strategy first chooses the negated conjecture and each axiom in turn from the usable set; this way, single-inference proofs are found early if they exist (i.e., if the conjecture is implied by an axiom). From then on, only clauses employing allowed symbols may be selected. Initially, the set of allowed symbols consists of those appearing in the conjecture. If no appropriate clause is available, the strategy looks for inferences that produce such a clause; failing that, the lightest clause is chosen, its symbols are added to the allowed symbols, and the maximal depth is incremented.

The *ranks-based* strategy requires each clause to carry a *rank* indicating its likely relevance. Like the goal-based strategy, it first selects the negated conjecture and the axioms. From then on, it always chooses the clause that minimizes the product $\mathit{weight} \times \mathit{depth} \times \mathit{rank}$. The rank of a derived clause is the minimum of the ranks of its parents. Conveniently, the facts returned by Sledgehammer's relevance filter are ordered by syntactic closeness to the goal to prove, a rough measure of relevance. The formula for assigning ranks interpolates linearly between .25 (for the first fact) and 1 (for the last fact). We have yet to experiment with other coefficients and interpolation methods.

Having several strategies to choose from may seem a luxury, but in conjunction with a simple optimization, *time slicing*, it helps find more proofs, especially if the strategies complement each other well. Automatic provers rarely find proofs after having run unsuccessfully for a few seconds, so it usually pays off to schedule a sequence of strategies, each with a fraction (or slice) of the total time limit. This idea can be extended to other aspects of the translation and proof search: the number of facts, type encodings, and λ-abstraction translation schemes, as well as various prover options.

Sledgehammer implements time slicing for any automatic prover, so that it is run with its own optimized schedule. We used a greedy algorithm to compute a schedule for SPASS based on a standard benchmark suite (Judgment Day [10]). The schedule incorporates both the goal- and the rank-based strategies, sometimes together with SOS.

6 Case Study: Formalization of Language-Based Security

As part of a global trend toward formalized computer science, the interactive theorem proving community has in recent years become interested in formalizing aspects of language-based security [37]. The pen-and-paper proofs of security results (typically, the soundness of a security type system) tend to be rather involved, requiring case analyses on the operational semantics rules and tedious bisimilarity reasoning. Formalizations of these results remain rare heroic efforts.

To assist such efforts, we are developing a framework to reason uniformly about a variety of security type systems and syntax-directed quantitative analyses. The central notion is that of *compositional bisimilarity relations*, which yield sound type systems that enforce security properties. The bulk of the development establishes compositionality of language constructs (e.g., sequential and parallel composition, 'while') with respect to bisimilarity relations (e.g., strong, weak, 01-bisimilarity) [33].

We rely heavily on Sledgehammer to automate the tedious details. Besides easing the proof development, automation helps keep the formal proofs in close correspondence with the original pen-and-paper proof. To illustrate this point, we review a typical proof of compositionality: sequential composition (;) with respect to strong bisimilarity ($\approx$). The goal is $c_1 \approx d_1 \wedge c_2 \approx d_2 \Longrightarrow c_1;c_2 \approx d_1;d_2$. The proof involves three steps:

1. Define a relation R that witnesses bisimilarity of $c_1;c_2$ and $d_1;d_2$.
2. Show that R is a bisimulation.
3. Conclude that $R \subseteq \approx$ by the definition of $\approx$ as greatest bisimulation.

Step 1 is the creative part of the argument. Here, the witness is unusually straightforward: $R = \{(c,d).\ \exists c_1\, c_2\, d_1\, d_2.\ c = c_1;c_2 \ \wedge\ d = d_1;d_2 \ \wedge\ c_1 \approx d_1 \ \wedge\ c_2 \approx d_2\}$. Steps 2 and 3 are left to the reader in textbook presentations and constitute good candidates for full automation. Unfortunately, the goal is beyond the reach of Isabelle's proof methods, and the translated first-order goals are too difficult for the automatic provers.

For step 2, we must show that if $(c,d) \in R$ and s and t are states indistinguishable to the attacker, written $s \sim t$, then any step taken by c in state s is matched by a step taken by d in state t such that the continuations are again in R and the resulting states are again indistinguishable. Assume $c = c_1;c_2$ and $(c,s) \to (c',s')$ represents a step taken by c in state s with continuation c' and resulting state s'.

2.1. By rule inversion for the semantics of sequential composition, either
   a. c' has the form $c'_1;c'_2$ and $(c_1,s) \to (c'_1,s')$; or
   b. $(c_1,s) \to s'$ (i.e., c_1 takes in s a terminating step to s') and $c' = c_2$.

2.2. Assume case a. From $c_1 \approx d_1$ we obtain d'_1 and t' such that $(d_1,t) \to (d'_1,t')$, $c'_1 \approx d'_1$, and $s' \sim t'$. (Case b is analogous, with $c_2 \approx d_2$ instead of $c_1 \approx d_1$.)

2.3. By one of the *intro* rules for the semantics of sequential composition, namely, $(c_1,s) \to (c'_1,s') \Longrightarrow (c_1;c_2,s) \to (c'_1;c_2,s')$, we obtain $(d_1;d_2,t) \to (d'_1;d_2,t')$. Hence, the desired matching step for $d = d_1;d_2$ is $(d,t) \to (d'_1;d_2,t')$.

Until recently, we would discharge 2.2 and 2.3 by invoking the simplifier enriched with the *intro* rules for the language construct (here, sequential composition) followed by Sledgehammer. The SPASS integration now allows us to discharge 2.2 and 2.3 without the need to customize and invoke the simplifier. In addition, if we are ready to wait a full minute, SPASS can discharge the entire step 2 in a single invocation, replacing the old cliché "left to the reader" with the more satisfying "left to the machine."

SPASS also eases reasoning about execution traces, modeled as lazy lists. Isabelle's library of lazy list lemmas is nowhere as comprehensive as that of finite lists. Reasoning about lazy lists can often exploit equations relating finite and lazy list operations via coercions. For example, under the assumption that the lazy lists xs and ys are finite, the following equations push the list_of coercion inside terms:

$$\mathsf{list\_of}\ (\mathsf{LCons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{list\_of}\ xs)$$
$$\mathsf{list\_of}\ (\mathsf{lappend}\ xs\ ys) = \mathsf{append}\ (\mathsf{list\_of}\ xs)\ (\mathsf{list\_of}\ ys)$$

The proper orientation of such equations helps discharge many goals for which we previously needed to engage at a tiresome level of detail.

7 Evaluation

This section attempts to quantify the enhancements described in Sections 3 to 5, both by evaluating each new feature in isolation and by letting the new SPASS compete against other automatic provers.[3] Our benchmarks are partitioned into three sets:

- *Judgment Day* (JD, 1268 goals) consists of seven theories from the Isabelle distribution and the *Archive of Formal Proofs* [21] that served as the main benchmark suite for Sledgehammer over the last two years [5, 6, 10, 31]. It covers areas as diverse as the fundamental theorem of algebra, the completeness of a Hoare logic, and the type soundness of a Java subset.
- *Arithmetic Extension of Judgment Day* (AX, 616 goals) consists of three Isabelle theories involving both linear and nonlinear arithmetic that were used in evaluations of SMT solvers and type encodings [5, 6].
- *Language-Based Security* (LS, 1042 goals) consists of five Isabelle theories belonging to the development described in Section 6.

Running Sledgehammer on a theory means launching it on the first subgoal at each position where a proof command appears in the theory's script.

To test the new SPASS features, we defined a base configuration and test variants of the configuration where one feature is disabled or replaced by another. Hard sorts, advisory and mandatory simplification, and goal-based clause selection are all part of the base configuration. The generated problems include up to 500 facts (700 after monomorphization). For each problem, SPASS is given 60 seconds of one thread on 64-bit Linux systems equipped with two Quad-Core Intel Xeon processors running at 2.4 GHz. We are interested in proof search alone, excluding proof reconstruction.

When analyzing enhancements to automatic provers, it is important to remember what difference a modest-looking gain of a few percentage points can make to users. The benchmarks were chosen to be representative of typical Isabelle goals and include many that are either too easy or too hard to effectively evaluate automatic provers. Indeed, some of the most essential tools in Isabelle, such the arithmetic decision procedures, score well below 10% when applied indiscriminately to the entire Judgment Day suite. Furthermore, SPASS has been fine-tuned over nearly two decades; it would be naive to expect enormous gains from isolated enhancements.

With this caveat in mind, let us review Figure 1. It considers six representations of types: three polymorphic and three monomorphic. Guards and tags are two traditional encodings. "Type arguments" was until recently the default in Sledgehammer; its actual success rate after reconstruction is lower than indicated, because some of the proofs found with it are unsound. "Light guards" is the new lightweight guard-based encoding. Many other encodings are implemented; in terms of performance, they are sandwiched between polymorphic guards and monomorphic light guards [6]. The results are fairly consistent across benchmark sets and confirm that hard sorts are superior to any encoding. The better translation schemes are also noticeably faster: Proofs with hard sorts require 3.0 seconds on average, compared with 5.0 for monomorphic guards.

[3] The test data set is available at `http://www21.in.tum.de/~blanchet/itp2012-data.tgz`.

* unsound	JD	AX	LS	All
Guards	28.7	17.9	29.7	25.6
Tags	37.3	23.8	39.1	33.5
Type args.*	**46.9**	**30.8**	**51.6**	**42.6**

(a) Polymorphic

	JD	AX	LS	All
Guards	40.7	29.8	46.1	37.9
Light guards	50.5	33.5	50.0	45.6
Hard sorts	**52.0**	**34.1**	**50.8**	**46.7**

(b) Monomorphic

Figure 1. Success rates (%) for the main type encodings

	JD	AX	LS	All
Advisory	+2.7	−2.9	+0.4	+0.9
Mandatory	+3.2	−**1.4**	+1.5	+1.8
Both	+**3.8**	−1.9	+**1.9**	+**2.2**

(a) Simplification

	JD	AX	LS	All
SOS	+9.9	−10.4	−4.2	+1.1
Goal-based	+15.3	+2.0	−1.5	+6.5
Rank-based	+**17.6**	+**6.9**	+**9.2**	+**12.6**

(b) Clause selection

Figure 2. Improvements (%) over the default setup for each proof heuristic

Figure 2(a) presents the impact of advisory and mandatory simplification as a percentage improvement over a configuration with both features disabled. The overall gain is 2.2% (i.e., SPASS solves 102.2 goals with both mechanisms enabled whenever it would solve 100 goals without them). Figure 2(b) compares three clause selection strategies with SPASS's default strategy. Our custom goal- and rank-based strategies are considerably more successful than the traditional SOS approach.

Figure 3 shows how the main clause-selection strategies scale when passed more facts, compared with the default setting. Both custom strategies degrade much less sharply than the default. The goal-based strategy scales the best, with a peak at 800 facts compared with 150 for the default strategy and 400 for the rank-based strategy. There is potential for improvement: With rank annotations, it should be possible to design a strategy that actually improves with the number of facts. As a thought experiment, a variant of the goal-based strategy that simply ignores all facts beyond the 800th would flatly outclass every strategy on Figure 3 when given, say, 900 or 1000 facts.

Finally, we measure the new version of SPASS against the latest versions of E (1.4), SPASS (3.7), Vampire (1.8 rev. 1435), and Z3 (3.2). Vampire and Z3 support hard sorts, and Z3 implements arithmetic decision procedures. This evaluation was conducted on the same hardware as the original Judgment Day study: 32-bit Linux systems with a Dual-Core Intel Xeon processor running at 3.06 GHz. The time limit is 60 seconds for proof search, potentially followed by minimization and reconstruction.

Figure 4(a) gives the success rate for each prover on each benchmark set. Overall, SPASS now solves 13% more goals than before, or 6.1 percentage points; this is enough to make it the single most useful automatic prover. About 45% of the goals from the chosen Isabelle theories are "trivial" in the sense that they can be solved directly by standard proof methods invoked with no arguments. If we ignore these and focus on the more interesting 1625 nontrivial goals, SPASS's improvements are even more significant: It solves 21% more goals than before (corresponding to 6.5 percentage points) and 10% more than the closest competitor (3.3 percentage points), as shown in Figure 4(b).

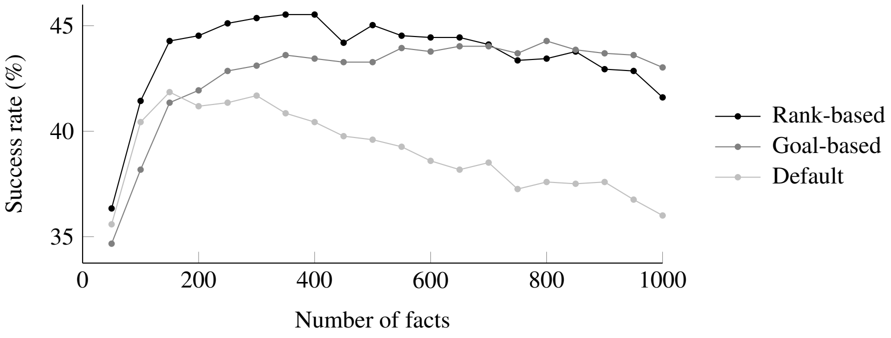

Figure 3. Scalability of each clause-selection strategy

	JD	AX	LS	All
New SPASS	**56.2**	38.3	**60.4**	**53.9**
Z3	53.5	**40.4**	57.6	52.2
Vampire	54.2	35.2	57.9	51.5
E	53.0	39.0	56.1	51.2
Old SPASS	50.0	31.8	54.6	47.8
Together	63.6	51.3	67.8	62.5

(a) All goals

	JD	AX	LS	All
New SPASS	**41.4**	**23.7**	**41.3**	**37.4**
Z3	38.6	21.3	36.7	34.1
Vampire	38.7	19.9	37.4	34.0
E	37.0	22.4	38.2	34.0
Old SPASS	34.2	19.7	34.2	30.9
Together	49.3	32.6	46.6	44.7

(b) Nontrivial goals

Figure 4. Success rates (%) with proof reconstruction for selected provers

Two years ago, the combination of (older versions of) E, SPASS, and Vampire running in parallel for 120 seconds solved 48% of Judgment Day [10]. Largely thanks to the developments presented in this paper, SPASS alone now solves 56% of the benchmark suite in half of the time, on the very same hardware.

8 Related Work

The most notable integrations of automatic provers in proof assistants, either as oracles or with proof reconstruction, are probably Otter in ACL2 [25]; Bliksem and veriT in Coq [2, 4]; Gandalf in HOL98 [18]; Z3 in HOL4 [11]; CVC Lite in HOL Light [26]; Vampire in Mizar [35]; Bliksem, EQP, LEO, Otter, PROTEIN, SPASS, TPS, and Waldmeister in ΩMEGA [39]; and Yices in PVS [36]. Of these, only LEO and Yices appear to have been significantly tailored to their host system. For program verification, Z3 in Boogie [3] and Alt-Ergo in Why3 [8] are examples of integrated proof environments.

Much of the developments currently taking place in first-order automatic theorem provers focus on solving particular classes of problems. This includes, for example, the automatic generation of inductive invariants for some theory or the efficient decision of large ontologies belonging to some decidable first-order fragment. From this point of view, our work on tailoring SPASS toward a better combination with Isabelle is the first dedicated contribution of its kind.

9 Conclusion

This paper described a tight, dedicated integration of Isabelle and SPASS. The heart of the approach is to communicate in a rich format that augments classical first-order logic with annotations for sorts, simplification rules, and fact relevance, and to let that information guide the proof search. The new version of SPASS outperforms E, Vampire, and Z3 on our extensive benchmark suites, and it is already helping us fill in the tedious details in language-based security proofs.

There is much room for future work, notably to support polymorphism and to extend the configurable simplification mechanisms to inductive and coinductive predicates and their *intro* and *elim* rules. It would also be desirable to polish and exploit SPASS's hierarchical support for linear and nonlinear arithmetic [1,13] and accommodate additional theories, such as algebraic datatypes, that are ubiquitous in formal proof developments. Finally, a promising avenue of work that could help derive deeper proofs within the short time allotted by Sledgehammer would be to have SPASS cache inferences across invocations, instead of re-deriving the same consequences from the same background theories over and over again.

We hope this research will serve as a blueprint for others to tailor their provers for proof assistants. Interactive theorem proving provides at least as many challenges as the annual competitions that are so popular in the automated reasoning community. Granted, there are no trophies or prizes attached to these challenges (a notable exception being the ISA category at CASC-23 [41]), but the satisfaction of assisting formalization efforts should be its own reward.

Acknowledgment. We thank Tobias Nipkow for making this collaboration possible and discussing the draft of this paper. We are also grateful to Lukas Bulwahn, Laura Faust, Lawrence Paulson, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements. This research was supported by the project Security Type Systems and Deduction (grants Ni 491/13-1 and We 4473/1-1) as part of the program Reliably Secure Software Systems (RS^3, Priority Program 1496) of the Deutsche Forschungsgemeinschaft (DFG). The first author was supported by the DFG project Quis Custodiet (grant Ni 491/11-2).

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer (2009)
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer (2011)
3. Barnett, M., Chang, B.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer (2006)
4. Bezem, M., Hendriks, D., de Nivelle, H.: Automatic proof construction in type theory using resolution. J. Autom. Reas. 29(3-4), 253–275 (2002)

5. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, vol. 6803, pp. 207–221. Springer (2011)
6. Blanchette, J.C., Böhme, S., Smallbone, N.: Monotonicity or how to encode polymorphic types safely and efficiently. http://www21.in.tum.de/~blanchet/mono-trans.pdf
7. Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. http://www21.in.tum.de/~blanchet/tff1spec.pdf
8. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT '08. pp. 1–5. ICPS, ACM (2008)
9. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNAI, vol. 6989, pp. 87–102. Springer (2011)
10. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNAI, vol. 6173, pp. 107–121. Springer (2010)
11. 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)
12. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer (2009)
13. Eggers, A., Kruglov, E., Kupferschmid, S., Scheibler, K., Teige, T., Weidenbach, C.: Superposition modulo non-linear arithmetic. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 119–134. Springer (2011)
14. Filliâtre, J.C.: Private communication (March 2012)
15. Ganzinger, H., Meyer, C., Weidenbach, C.: Soft typing for ordered resolution. In: McCune, W. (ed.) CADE-14. LNCS, vol. 1249, pp. 321–335. Springer (1997)
16. Gonthier, G.: Formal proof—The four-color theorem. N. AMS 55(11), 1382–1393 (2008)
17. Guttmann, W., Struth, G., Weber, T.: Automating algebraic methods in Isabelle. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 617–632. Springer (2011)
18. Hurd, J.: Integrating Gandalf and HOL. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs '99. LNCS, vol. 1690, pp. 311–321 (1999)
19. 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)
20. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. C. ACM 53(6), 107–115 (2010)
21. Klein, G., Nipkow, T., Paulson, L. (eds.): The Archive of Formal Proofs. http://afp.sf.net/
22. Knuth, D.E., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
23. Leroy, X.: A formally verified compiler back-end. J. Autom. Reas. 43(4), 363–446 (2009)
24. Leroy, X.: Private communication (October 2011)
25. McCune, W., Shumsky, O.: System description: IVY. In: McAllester, D. (ed.) CADE-17. LNAI, vol. 1831, pp. 401–405. Springer (2000)
26. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. ENTCS 144(2), 43–51 (2006)
27. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reas. 40(1), 35–60 (2008)
28. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)

29. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
30. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
31. 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)
32. 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. Springer (2007)
33. Popescu, A., Hölzl, J., Nipkow, T.: Proving possibilistic, probabilistic noninterference. Submitted. `http://www21.in.tum.de/~popescua/pos.pdf`
34. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. 15(2-3), 91–110 (2002)
35. Rudnicki, P., Urban, J.: Escape to ATP for Mizar. In: Fontaine, P., Stump, A. (eds.) PxTP-2011 (2011)
36. 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)
37. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Comm. 21(1), 5–19 (2003)
38. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNAI, vol. 3097, pp. 223–228. Springer (2004)
39. Siekmann, J., Benzmüller, C., Fiedler, A., Meier, A., Normann, I., Pollet, M.: Proof development with ΩMEGA: The irrationality of $\sqrt{2}$. In: Kamareddine, F. (ed.) Thirty Five Years of Automating Mathematics. Applied Logic, vol. 28, pp. 271–314. Springer (2003)
40. Sutcliffe, G.: The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. J. Autom. Reas. 43(4), 337–362 (2009)
41. Sutcliffe, G.: The CADE-23 automated theorem proving system competition—CASC-23. AI Comm. 25(1), 49–63 (2012)
42. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 406–419. Springer (2012)
43. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, pp. 1965–2013. Elsevier (2001)
44. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNAI, vol. 5663, pp. 140–145. Springer (2009)
45. Wos, L., Robinson, G.A., Carson, D.F.: Efficiency and completeness of the set of support strategy in theorem proving. J. ACM 12(4), 536–541 (1965)