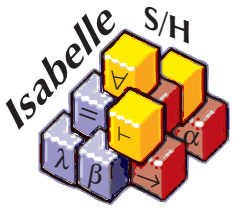

Hammering Away

A User's Guide to Sledgehammer for Isabelle/HOL

Jasmin Christian Blanchette
Institut für Informatik, Technische Universität München

with contributions from

Lawrence C. Paulson
Computer Laboratory, University of Cambridge

May 22, 2012

Contents

1 Introduction

Sledgehammer is a tool that applies automatic theorem provers (ATPs) and satisfiability-modulo-theories (SMT) solvers on the current goal.[1] The supported ATPs are E [13], E-SInE [9], E-ToFoF [15], iProver [10], iProver-Eq [11], LEO-II [2], Satallax [6], SNARK [14], SPASS [17], Vampire [12], and Waldmeister [8]. The ATPs are run either locally or remotely via the SystemOnTPTP web service [16]. In addition to the ATPs, the SMT solvers Z3 [18]

1. The distinction between ATPs and SMT solvers is convenient but mostly historical. The two communities are converging, with more and more ATPs supporting typical SMT features such as arithmetic and sorts, and a few SMT solvers parsing ATP syntaxes. There is also a strong technological connection between instantiation-based ATPs (such as iProver and iProver-Eq) and SMT solvers.

is used by default, and you can tell Sledgehammer to try Alt-Ergo [3], CVC3 [1], and Yices [7] as well; these are run either locally or (for CVC3 and Z3) on a server at the TU München.

The problem passed to the automatic provers consists of your current goal together with a heuristic selection of hundreds of facts (theorems) from the current theory context, filtered by relevance. Because jobs are run in the background, you can continue to work on your proof by other means. Provers can be run in parallel. Any reply (which may arrive half a minute later) will appear in the Proof General response buffer.

The result of a successful proof search is some source text that usually (but not always) reconstructs the proof within Isabelle. For ATPs, the reconstructed proof relies on the general-purpose *metis* proof method, which integrates the Metis ATP in Isabelle/HOL with explicit inferences going through the kernel. Thus its results are correct by construction.

In this manual, we will explicitly invoke the **sledgehammer** command. Sledgehammer also provides an automatic mode that can be enabled via the "Auto Sledgehammer" option in Proof General's "Isabelle" menu. In this mode, Sledgehammer is run on every newly entered theorem. The time limit for Auto Sledgehammer and other automatic tools can be set using the "Auto Tools Time Limit" option.

To run Sledgehammer, you must make sure that the theory *Sledgehammer* is imported—this is rarely a problem in practice since it is part of *Main*. Examples of Sledgehammer use can be found in Isabelle's `src/HOL/Metis_Examples` directory. Comments and bug reports concerning Sledgehammer or this manual should be directed to the author at `blanchette@in.tum.de`.

2 Installation

Sledgehammer is part of Isabelle, so you don't need to install it. However, it relies on third-party automatic provers (ATPs and SMT solvers).

Among the ATPs, E, LEO-II, Satallax, SPASS, and Vampire can be run locally; in addition, E, E-SInE, E-ToFoF, iProver, iProver-Eq, LEO-II, Satallax, SNARK, Vampire, and Waldmeister are available remotely via SystemOnTPTP [16]. If you want better performance, you should at least install E and SPASS locally.

Among the SMT solvers, Alt-Ergo, CVC3, Yices, and Z3 can be run locally, and CVC3 and Z3 can be run remotely on a TU München server. If you want better performance and get the ability to replay proofs that rely on the *smt* proof method without an Internet connection, you should at least install Z3 locally.

There are three main ways to install automatic provers on your machine:

- If you installed an official Isabelle package, it should already include properly setup executables for CVC3, E, SPASS, and Z3, ready to use. [2] For Z3, you must additionally set the variable `Z3_NON_COMMERCIAL` to "yes" to confirm that you are a noncommercial user, either in the environment in which Isabelle is launched or in your `$ISABELLE_HOME_USER/etc/settings` file.

- Alternatively, you can download the Isabelle-aware CVC3, E, SPASS, and Z3 binary packages from `http://www21.in.tum.de/~blanchet/#software`. Extract the archives, then add a line to your `$ISABELLE_HOME_USER/etc/components` [3] file with the absolute path to CVC3, E, SPASS, or Z3. For example, if the `components` file does not exist yet and you extracted SPASS to `/usr/local/spass-3.8ds`, create it with the single line

  ```
  /usr/local/spass-3.8ds
  ```

  in it.

- If you prefer to build E, LEO-II, Satallax, or SPASS manually, or found a Vampire executable somewhere (e.g., `http://www.vprover.org/`), set the environment variable `E_HOME`, `LEO2_HOME`, `SATALLAX_HOME`, `SPASS_HOME`, or `VAMPIRE_HOME` to the directory that contains the `eproof`, `leo`, `satallax`, `SPASS`, or `vampire` executable. Sledgehammer has been tested with E 1.0 to 1.4, LEO-II 1.2.9, Satallax 2.2 and 2.3, SPASS 3.5, 3.7, and 3.8ds, and Vampire 0.6, 1.0, and 1.8 [4]. Since the ATPs' output formats are neither documented nor stable, other versions might not work well with Sledgehammer. Ideally, you should also set `E_VERSION`, `LEO2_VERSION`, `SATALLAX_VERSION`, `SPASS_VERSION`, or `VAMPIRE_VERSION` to the prover's version number (e.g., "1.4").

2. Vampire's and Yices's licenses prevent us from doing the same for these otherwise remarkable tools.

3. The variable `$ISABELLE_HOME_USER` is set by Isabelle at startup. Its value can be retrieved by executing `isabelle getenv ISABELLE_HOME_USER` on the command line.

4. Following the rewrite of Vampire, the counter for version numbers was reset to 0; hence the (new) Vampire versions 0.6, 1.0, and 1.8 are more recent than 9.0 or 11.5.

Similarly, if you want to build Alt-Ergo or CVC3, or found a Yices or Z3 executable somewhere (e.g., `http://yices.csl.sri.com/download.shtml` or `http://research.microsoft.com/en-us/um/redmond/projects/z3/download.html`), set the environment variable `CVC3_SOLVER`, `YICES_SOLVER`, or `Z3_SOLVER` to the complete path of the executable, *including the file name*. Sledgehammer has been tested with Alt-Ergo 0.93, CVC3 2.2 and 2.4.1, Yices 1.0.28 and 1.0.33, and Z3 3.0 to 3.2. Since the SMT solvers' output formats are somewhat unstable, other versions of the solvers might not work well with Sledgehammer. Ideally, also set `CVC3_VERSION`, `YICES_VERSION`, or `Z3_VERSION` to the solver's version number (e.g., "3.2").

To check whether E, SPASS, Vampire, and/or Z3 are successfully installed, try out the example in §3. If the remote versions of any of these provers is used (identified by the prefix "*remote_*"), or if the local versions fail to solve the easy goal presented there, something must be wrong with the installation.

Remote prover invocation requires Perl with the World Wide Web Library (`libwww-perl`) installed. If you must use a proxy server to access the Internet, set the `http_proxy` environment variable to the proxy, either in the environment in which Isabelle is launched or in your `$ISABELLE_HOME_USER/etc/settings` file. Here are a few examples:

```
http_proxy=http://proxy.example.org
http_proxy=http://proxy.example.org:8080
http_proxy=http://joeblow:pAsSwRd@proxy.example.org
```

3 First Steps

To illustrate Sledgehammer in context, let us start a theory file and attempt to prove a simple lemma:

theory *Scratch*
imports *Main*
begin

lemma "$[a] = [b] \Longrightarrow a = b$"
sledgehammer

Instead of issuing the **sledgehammer** command, you can also find Sledgehammer in the "Commands" submenu of the "Isabelle" menu in Proof General

or press the Emacs key sequence C-c C-a C-s. Either way, Sledgehammer produces the following output after a few seconds:

> *Sledgehammer: "e" on goal*
> $[a] = [b] \Longrightarrow a = b$
> *Try this:* **by** *(metis last_ConsL) (64 ms).*
>
> *Sledgehammer: "z3" on goal*
> $[a] = [b] \Longrightarrow a = b$
> *Try this:* **by** *(metis list.inject) (20 ms).*
>
> *Sledgehammer: "vampire" on goal*
> $[a] = [b] \Longrightarrow a = b$
> *Try this:* **by** *(metis hd.simps) (14 ms).*
>
> *Sledgehammer: "spass" on goal*
> $[a] = [b] \Longrightarrow a = b$
> *Try this:* **by** *(metis list.inject) (17 ms).*
>
> *Sledgehammer: "remote_waldmeister" on goal*
> $[a] = [b] \Longrightarrow a = b$
> *Try this:* **by** *(metis hd.simps) (15 ms).*
>
> *Sledgehammer: "remote_e_sine" on goal*
> $[a] = [b] \Longrightarrow a = b$
> *Try this:* **by** *(metis hd.simps) (18 ms).*

Sledgehammer ran E, E-SInE, SPASS, Vampire, Waldmeister, and Z3 in parallel. Depending on which provers are installed and how many processor cores are available, some of the provers might be missing or present with a *remote_* prefix. Waldmeister is run only for unit equational problems, where the goal's conclusion is a (universally quantified) equation.

For each successful prover, Sledgehammer gives a one-liner proof that uses the *metis* or *smt* proof method. Approximate timings are shown in parentheses, indicating how fast the call is. You can click the proof to insert it into the theory text.

In addition, you can ask Sledgehammer for an Isar text proof by passing the *isar_proof* option (§7.4):

> **sledgehammer** [*isar_proof*]

When Isar proof construction is successful, it can yield proofs that are more readable and also faster than the *metis* or *smt* one-liners. This feature is

experimental and is only available for ATPs.

4 Hints

This section presents a few hints that should help you get the most out of Sledgehammer. Frequently asked questions are answered in §5.

4.1 *Presimplify the goal*

For best results, first simplify your problem by calling *auto* or at least *safe* followed by *simp_all*. The SMT solvers provide arithmetic decision procedures, but the ATPs typically do not (or if they do, Sledgehammer does not use it yet). Apart from Waldmeister, they are not especially good at heavy rewriting, but because they regard equations as undirected, they often prove theorems that require the reverse orientation of a *simp* rule. Higher-order problems can be tackled, but the success rate is better for first-order problems. Hence, you may get better results if you first simplify the problem to remove higher-order features.

4.2 *Make sure E, SPASS, Vampire, and Z3 are locally installed*

Locally installed provers are faster and more reliable than those running on servers. See §2 for details on how to install them.

4.3 *Familiarize yourself with the most important options*

Sledgehammer's options are fully documented in §6. Many of the options are very specialized, but serious users of the tool should at least familiarize themselves with the following options:

- ***provers*** (§7.1) specifies the automatic provers (ATPs and SMT solvers) that should be run whenever Sledgehammer is invoked (e.g., "*provers* = *e spass remote_vampire*"). For convenience, you can omit "*provers* ="

and simply write the prover names as a space-separated list (e.g., "*e spass remote_vampire*").

- ***max_relevant*** (§7.3) specifies the maximum number of facts that should be passed to the provers. By default, the value is prover-dependent but varies between about 150 and 1000. If the provers time out, you can try lowering this value to, say, 100 or 50 and see if that helps.
- ***isar_proof*** (§7.4) specifies that Isar proofs should be generated, instead of one-liner *metis* or *smt* proofs. The length of the Isar proofs can be controlled by setting *isar_shrink_factor* (§7.4).
- ***timeout*** (§7.6) controls the provers' time limit. It is set to 30 seconds, but since Sledgehammer runs asynchronously you should not hesitate to raise this limit to 60 or 120 seconds if you are the kind of user who can think clearly while ATPs are active.

Options can be set globally using **sledgehammer_params** (§6). The command also prints the list of all available options with their current value. Fact selection can be influenced by specifying "(*add*: *my_facts*)" after the **sledgehammer** call to ensure that certain facts are included, or simply "(*my_facts*)" to force Sledgehammer to run only with *my_facts*.

5 Frequently Asked Questions

This sections answers frequently (and infrequently) asked questions about Sledgehammer. It is a good idea to skim over it now even if you don't have any questions at this stage. And if you have any further questions not listed here, send them to the author at `blanchette@in.tum.de`.

5.1 *Which facts are passed to the automatic provers?*

The relevance filter assigns a score to every available fact (lemma, theorem, definition, or axiom) based upon how many constants that fact shares with the conjecture. This process iterates to include facts relevant to those just accepted, but with a decay factor to ensure termination. The constants are weighted to give unusual ones greater significance. The relevance filter copes best when the conjecture contains some unusual constants; if all the constants are common, it is unable to discriminate among the hundreds of facts that

are picked up. The relevance filter is also memoryless: It has no information about how many times a particular fact has been used in a proof, and it cannot learn.

The number of facts included in a problem varies from prover to prover, since some provers get overwhelmed more easily than others. You can show the number of facts given using the *verbose* option (§7.4) and the actual facts using *debug* (§7.4).

Sledgehammer is good at finding short proofs combining a handful of existing lemmas. If you are looking for longer proofs, you must typically restrict the number of facts, by setting the *max_relevant* option (§7.3) to, say, 25 or 50.

You can also influence which facts are actually selected in a number of ways. If you simply want to ensure that a fact is included, you can specify it using the "(*add*: *my_facts*)" syntax. For example:

> **sledgehammer** (*add*: *hd.simps tl.simps*)

The specified facts then replace the least relevant facts that would otherwise be included; the other selected facts remain the same. If you want to direct the selection in a particular direction, you can specify the facts via **using**:

> **using** *hd.simps tl.simps*
> **sledgehammer**

The facts are then more likely to be selected than otherwise, and if they are selected at iteration j they also influence which facts are selected at iterations $j + 1$, $j + 2$, etc. To give them even more weight, try

> **using** *hd.simps tl.simps*
> **apply** −
> **sledgehammer**

5.2 *Why does Metis fail to reconstruct the proof?*

There are many reasons. If Metis runs seemingly forever, that is a sign that the proof is too difficult for it. Metis's search is complete, so it should eventually find it, but that's little consolation. There are several possible solutions:

- Try the *isar_proof* option (§7.4) to obtain a step-by-step Isar proof where each step is justified by *metis*. Since the steps are fairly small, *metis* is more likely to be able to replay them.

- Try the *smt* proof method instead of *metis*. It is usually stronger, but you need to either have Z3 available to replay the proofs, trust the SMT solver, or use certificates. See the documentation in the *SMT* theory (`$ISABELLE_HOME/src/HOL/SMT.thy`) for details.
- Try the *blast* or *auto* proof methods, passing the necessary facts via **unfolding**, **using**, *intro*:, *elim*:, *dest*:, or *simp*:, as appropriate.

In some rare cases, *metis* fails fairly quickly, and you get the error message

> *One-line proof reconstruction failed.*

This message indicates that Sledgehammer determined that the goal is provable, but the proof is, for technical reasons, beyond *metis*'s power. You can then try again with the *strict* option (§7.2).

If the goal is actually unprovable and you did not specify an unsound encoding using *type_enc* (§7.2), this is a bug, and you are strongly encouraged to report this to the author at `blanchette@in.tum.de`.

5.3 *Why are the generated Isar proofs so ugly/broken?*

The current implementation of the Isar proof feature, enabled by the *isar_proof* option (§7.4), is highly experimental. Work on a new implementation has begun. There is a large body of research into transforming resolution proofs into natural deduction proofs (such as Isar proofs), which we hope to leverage. In the meantime, a workaround is to set the *isar_shrink_factor* option (§7.4) to a larger value or to try several provers and keep the nicest-looking proof.

5.4 *How can I tell whether a suggested proof is sound?*

Earlier versions of Sledgehammer often suggested unsound proofs—either proofs of nontheorems or simply proofs that rely on type-unsound inferences. This is a thing of the past, unless you explicitly specify an unsound encoding using *type_enc* (§7.2). Officially, the only form of "unsoundness" that lurks in the sound encodings is related to missing characteristic theorems of datatypes. For example,

> **lemma** "$\exists xs.\ xs \neq []$"
> **sledgehammer** ()

suggests an argumentless *metis* call that fails. However, the conjecture does actually hold, and the *metis* call can be repaired by adding *list.distinct*. We hope to address this problem in a future version of Isabelle. In the meantime, you can avoid it by passing the *strict* option (§7.2).

5.5 *What are the full_types, no_types, and mono_tags arguments to Metis?*

The *metis* (*full_types*) proof method and its cousin *metis* (*mono_tags*) are fully-typed version of Metis. It is somewhat slower than *metis*, but the proof search is fully typed, and it also includes more powerful rules such as the axiom "$x = \mathsf{True} \vee x = \mathsf{False}$" for reasoning in higher-order places (e.g., in set comprehensions). The method kicks in automatically as a fallback when *metis* fails, and it is sometimes generated by Sledgehammer instead of *metis* if the proof obviously requires type information or if *metis* failed when Sledgehammer preplayed the proof. (By default, Sledgehammer tries to run *metis* with various options for up to 3 seconds each time to ensure that the generated one-line proofs actually work and to display timing information. This can be configured using the *preplay_timeout* and *dont_preplay* options (§7.6).) At the other end of the soundness spectrum, *metis* (*no_types*) uses no type information at all during the proof search, which is more efficient but often fails. Calls to *metis* (*no_types*) are occasionally generated by Sledgehammer. See the *type_enc* option (§7.2) for details.

Incidentally, if you ever see warnings such as

> *Metis: Falling back on "metis (full_types)".*

for a successful *metis* proof, you can advantageously pass the *full_types* option to *metis* directly.

5.6 *And what are the lifting and hide_lams arguments to Metis?*

Orthogonally to the encoding of types, it is important to choose an appropriate translation of λ-abstractions. Metis supports three translation schemes, in decreasing order of power: Curry combinators (the default), λ-lifting, and a "hiding" scheme that disables all reasoning under λ-abstractions. The more powerful schemes also give the automatic provers more rope to hang themselves. See the *lam_trans* option (§7.2) for details.

5.7 *Are generated proofs minimal?*

Automatic provers frequently use many more facts than are necessary. Sledgehammer inclues a minimization tool that takes a set of facts returned by a given prover and repeatedly calls the same prover, *metis*, or *smt* with subsets of those axioms in order to find a minimal set. Reducing the number of axioms typically improves Metis's speed and success rate, while also removing superfluous clutter from the proof scripts.

In earlier versions of Sledgehammer, generated proofs were systematically accompanied by a suggestion to invoke the minimization tool. This step is now performed implicitly if it can be done in a reasonable amount of time (something that can be guessed from the number of facts in the original proof and the time it took to find or preplay it).

In addition, some provers (e.g., Yices) do not provide proofs or sometimes produce incomplete proofs. The minimizer is then invoked to find out which facts are actually needed from the (large) set of facts that was initially given to the prover. Finally, if a prover returns a proof with lots of facts, the minimizer is invoked automatically since Metis would be unlikely to re-find the proof. Automatic minimization can be forced or disabled using the *minimize* option (§7.1).

5.8 *A strange error occurred—what should I do?*

Sledgehammer tries to give informative error messages. Please report any strange error to the author at `blanchette@in.tum.de`. This applies double if you get the message

> *The prover found a type-unsound proof involving "foo", "bar", and "baz" even though a supposedly type-sound encoding was used (or, less likely, your axioms are inconsistent). You might want to report this to the Isabelle developers.*

5.9 *Auto can solve it—why not Sledgehammer?*

Problems can be easy for *auto* and difficult for automatic provers, but the reverse is also true, so don't be discouraged if your first attempts fail. Because the system refers to all theorems known to Isabelle, it is particularly suitable when your goal has a short proof from lemmas that you don't know about.

5.10 *Why are there so many options?*

Sledgehammer's philosophy should work out of the box, without user guidance. Many of the options are meant to be used mostly by the Sledgehammer developers for experimentation purposes. Of course, feel free to experiment with them if you are so inclined.

6 Command Syntax

6.1 Sledgehammer

Sledgehammer can be invoked at any point when there is an open goal by entering the **sledgehammer** command in the theory file. Its general syntax is as follows:

sledgehammer ⟨*subcommand*⟩$^?$ ⟨*options*⟩$^?$ ⟨*facts_override*⟩$^?$ ⟨*num*⟩$^?$

For convenience, Sledgehammer is also available in the "Commands" submenu of the "Isabelle" menu in Proof General or by pressing the Emacs key sequence C-c C-a C-s. This is equivalent to entering the **sledgehammer** command with no arguments in the theory text.

In the general syntax, the ⟨*subcommand*⟩ may be any of the following:

- ***run* (the default):** Runs Sledgehammer on subgoal number ⟨*num*⟩ (1 by default), with the given options and facts.
- ***min*:** Attempts to minimize the facts specified in the ⟨*facts_override*⟩ argument to obtain a simpler proof involving fewer facts. The options and goal number are as for *run*.
- ***messages*:** Redisplays recent messages issued by Sledgehammer. This allows you to examine results that might have been lost due to Sledgehammer's asynchronous nature. The ⟨*num*⟩ argument specifies a limit on the number of messages to display (10 by default).
- ***supported_provers*:** Prints the list of automatic provers supported by Sledgehammer. See §2 and §7.1 for more information on how to install automatic provers.
- ***running_provers*:** Prints information about currently running automatic provers, including elapsed runtime and remaining time until timeout.

- ***kill_provers*:** Terminates all running automatic provers.
- ***refresh_tptp*:** Refreshes the list of remote ATPs available at System-OnTPTP [16].

Sledgehammer's behavior can be influenced by various ⟨*options*⟩, which can be specified in brackets after the **sledgehammer** command. The ⟨*options*⟩ are a list of key–value pairs of the form "$[k_1 = v_1, \ldots, k_n = v_n]$". For Boolean options, "= *true*" is optional. For example:

sledgehammer [*isar_proof*, *timeout* = 120]

Default values can be set using **sledgehammer_params**:

sledgehammer_params ⟨*options*⟩

The supported options are described in §7.

The ⟨*facts_override*⟩ argument lets you alter the set of facts that go through the relevance filter. It may be of the form "(⟨*facts*⟩)", where ⟨*facts*⟩ is a space-separated list of Isabelle facts (theorems, local assumptions, etc.), in which case the relevance filter is bypassed and the given facts are used. It may also be of the form "(*add*: ⟨*facts*$_1$⟩)", "(*del*: ⟨*facts*$_2$⟩)", or "(*add*: ⟨*facts*$_1$⟩ *del*: ⟨*facts*$_2$⟩)", where the relevance filter is instructed to proceed as usual except that it should consider ⟨*facts*$_1$⟩ highly-relevant and ⟨*facts*$_2$⟩ fully irrelevant.

You can instruct Sledgehammer to run automatically on newly entered theorems by enabling the "Auto Sledgehammer" option in Proof General's "Isabelle" menu. For automatic runs, only the first prover set using *provers* (§7.1) is considered, fewer facts are passed to the prover, *slice* (§7.1) is disabled, *strict* (§7.2) is enabled, *verbose* (§7.4) and *debug* (§7.4) are disabled, and *timeout* (§7.6) is superseded by the "Auto Tools Time Limit" in Proof General's "Isabelle" menu. Sledgehammer's output is also more concise.

6.2 Metis

The *metis* proof method has the syntax

metis (⟨*options*⟩)$^?$ ⟨*facts*⟩$^?$

where ⟨*facts*⟩ is a list of arbitrary facts and ⟨*options*⟩ is a comma-separated list consisting of at most one λ translation scheme specification with the same semantics as Sledgehammer's *lam_trans* option (§7.2) and at most one type encoding specification with the same semantics as Sledgehammer's *type_enc*

option (§7.2). The supported λ translation schemes are *hide_lams*, *lifting*, and *combs* (the default). All the untyped type encodings listed in §7.2 are supported. For convenience, the following aliases are provided:

- ***full_types*:** Synonym for *poly_guards_query*.
- ***partial_types*:** Synonym for *poly_args*.
- ***no_types*:** Synonym for *erased*.

7 Option Reference

Sledgehammer's options are categorized as follows: mode of operation (§7.1), problem encoding (§7.2), relevance filter (§7.3), output format (§7.4), authentication (§7.5), and timeouts (§7.6).

The descriptions below refer to the following syntactic quantities:

- ⟨***string***⟩: A string.
- ⟨***bool***⟩: *true* or *false*.
- ⟨***smart_bool***⟩: *true*, *false*, or *smart*.
- ⟨***int***⟩: An integer.
- ⟨***float_pair***⟩: A pair of floating-point numbers (e.g., 0.6 0.95).
- ⟨***smart_int***⟩: An integer or *smart*.
- ⟨***float_or_none***⟩: A floating-point number (e.g., 60 or 0.5) expressing a number of seconds, or the keyword *none* (∞ seconds).

Default values are indicated in curly brackets ({}). Boolean options have a negated counterpart (e.g., *blocking* vs. *non_blocking*). When setting them, "= *true*" may be omitted.

7.1 Mode of Operation

[***provers*** =] ⟨***string***⟩

Specifies the automatic provers to use as a space-separated list (e.g., "*e spass remote_vampire*"). Provers can be run locally or remotely; see §2 for installation instructions.

The following local provers are supported:

- ***alt_ergo*:** Alt-Ergo is a polymorphic SMT solver developed by Bobot et al. [3]. It supports the TPTP polymorphic typed first-order format (TFF1) via Why3 [4]. It is included for experimental purposes. To use Alt-Ergo, set the environment variable `WHY3_HOME` to the directory that contains the `why3` executable. Sledgehammer has been tested with Alt-Ergo 0.93 and an unidentified development version of Why3.
- ***cvc3*:** CVC3 is an SMT solver developed by Clark Barrett, Cesare Tinelli, and their colleagues [1]. To use CVC3, set the environment variable `CVC3_SOLVER` to the complete path of the executable, including the file name, or install the prebuilt CVC3 package from `http://www21.in.tum.de/~blanchet/#software`. Sledgehammer has been tested with version 2.2.
- ***e*:** E is a first-order resolution prover developed by Stephan Schulz [13]. To use E, set the environment variable `E_HOME` to the directory that contains the `eproof` executable and `E_VERSION` to the version number (e.g., "1.4"), or install the prebuilt E package from `http://www21.in.tum.de/~blanchet/#software`. Sledgehammer has been tested with versions 1.0 to 1.4.
- ***leo2*:** LEO-II is an automatic higher-order prover developed by Christoph Benzmüller et al. [2], with support for the TPTP typed higher-order syntax (THF0). To use LEO-II, set the environment variable `LEO2_HOME` to the directory that contains the `leo` executable. Sledgehammer requires version 1.2.9 or above.
- ***metis*:** Although it is much less powerful than the external provers, Metis itself can be used for proof search.
- ***satallax*:** Satallax is an automatic higher-order prover developed by Chad Brown et al. [6], with support for the TPTP typed higher-order syntax (THF0). To use Satallax, set the environment variable `SATALLAX_HOME` to the directory that contains the `satallax` executable. Sledgehammer requires version 2.2 or above.
- ***smt*:** The *smt* proof method with the current settings (usually: Z3 with proof reconstruction).
- ***spass*:** SPASS is a first-order resolution prover developed by Christoph Weidenbach et al. [17]. To use SPASS, set the environment variable `SPASS_HOME` to the directory that contains the `SPASS` executable and `SPASS_VERSION` to the version number (e.g., "3.8ds"), or install the prebuilt SPASS package from `http://www21.in.tum.de/~blanchet/#software`. Sledgehammer requires version 3.5 or above.

- ***vampire*:** Vampire is a first-order resolution prover developed by Andrei Voronkov and his colleagues [12]. To use Vampire, set the environment variable `VAMPIRE_HOME` to the directory that contains the `vampire` executable and `VAMPIRE_VERSION` to the version number (e.g., "1.8"). Sledgehammer has been tested with versions 0.6, 1.0, and 1.8. Versions above 1.8 support the TPTP typed first-order format (TFF0).
- ***yices*:** Yices is an SMT solver developed at SRI [7]. To use Yices, set the environment variable `YICES_SOLVER` to the complete path of the executable, including the file name. Sledgehammer has been tested with version 1.0.28.
- ***z3*:** Z3 is an SMT solver developed at Microsoft Research [18]. To use Z3, set the environment variable `Z3_SOLVER` to the complete path of the executable, including the file name, and set `Z3_NON_COMMERCIAL` to "yes" to confirm that you are a noncommercial user. Sledgehammer has been tested with versions 3.0 to 3.2.
- ***z3_tptp*:** This version of Z3 pretends to be an ATP, exploiting Z3's support for the TPTP untyped and typed first-order formats (FOF and TFF0). It is included for experimental purposes. It requires version 3.0 or above. To use it, set the environment variable `Z3_HOME` to the directory that contains the `z3` executable.

The following remote provers are supported:

- ***remote_cvc3*:** The remote version of CVC3 runs on servers at the TU München (or wherever `REMOTE_SMT_URL` is set to point).
- ***remote_e*:** The remote version of E runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_e_sine*:** E-SInE is a metaprover developed by Kryštof Hoder [9] based on E. It runs on Geoff Sutcliffe's Miami servers.
- ***remote_e_tofof*:** E-ToFoF is a metaprover developed by Geoff Sutcliffe [15] based on E running on his Miami servers. This ATP supports the TPTP typed first-order format (TFF0). The remote version of E-ToFoF runs on Geoff Sutcliffe's Miami servers.
- ***remote_iprover*:** iProver is a pure instantiation-based prover developed by Konstantin Korovin [10]. The remote version of iProver runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_iprover_eq*:** iProver-Eq is an instantiation-based prover with native support for equality developed by Konstantin Korovin and Christoph Sticksel [11]. The remote version of iProver-Eq runs on Geoff Sutcliffe's Miami servers [16].

- ***remote_leo2*:** The remote version of LEO-II runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_satallax*:** The remote version of Satallax runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_snark*:** SNARK is a first-order resolution prover developed by Stickel et al. [14]. It supports the TPTP typed first-order format (TFF0). The remote version of SNARK runs on Geoff Sutcliffe's Miami servers.
- ***remote_vampire*:** The remote version of Vampire runs on Geoff Sutcliffe's Miami servers. Version 1.8 is used.
- ***remote_waldmeister*:** Waldmeister is a unit equality prover developed by Hillenbrand et al. [8]. It can be used to prove universally quantified equations using unconditional equations, corresponding to the TPTP CNF UEQ division. The remote version of Waldmeister runs on Geoff Sutcliffe's Miami servers.
- ***remote_z3*:** The remote version of Z3 runs on servers at the TU München (or wherever `REMOTE_SMT_URL` is set to point).
- ***remote_z3_tptp*:** The remote version of "Z3 with TPTP syntax" runs on Geoff Sutcliffe's Miami servers.

By default, Sledgehammer runs E, E-SInE, SPASS, Vampire, Z3 (or whatever the SMT module's *smt_solver* configuration option is set to), and (if appropriate) Waldmeister in parallel—either locally or remotely, depending on the number of processor cores available. For historical reasons, the default value of this option can be overridden using the option "Sledgehammer: Provers" in Proof General's "Isabelle" menu.

It is generally a good idea to run several provers in parallel. Running E, SPASS, and Vampire for 5 seconds yields a similar success rate to running the most effective of these for 120 seconds [5].

For the *min* subcommand, the default prover is *metis*. If several provers are set, the first one is used.

prover $= \langle$***string***$\rangle$

Alias for *provers*.

blocking $[= \langle$***bool***$\rangle]$ **{*false*}** **(neg.: *non_blocking*)**

Specifies whether the **sledgehammer** command should operate synchronously. The asynchronous (non-blocking) mode lets the user start proving the putative theorem manually while Sledgehammer looks for a proof, but it can also be more confusing. Irrespective of the value of this option, Sledgehammer is always run synchronously for the new jEdit-based user interface or if *debug* (§7.4) is enabled.

slice $[= \langle bool \rangle]$ **{*true*}** **(neg.: *dont_slice*)**

Specifies whether the time allocated to a prover should be sliced into several segments, each of which has its own set of possibly prover-dependent options. For SPASS and Vampire, the first slice tries the fast but incomplete set-of-support (SOS) strategy, whereas the second slice runs without it. For E, up to three slices are tried, with different weighted search strategies and number of facts. For SMT solvers, several slices are tried with the same options each time but fewer and fewer facts. According to benchmarks with a timeout of 30 seconds, slicing is a valuable optimization, and you should probably leave it enabled unless you are conducting experiments. This option is implicitly disabled for (short) automatic runs.

See also *verbose* (§7.4).

minimize $[= \langle smart\_bool \rangle]$ **{*smart*}** **(neg.: *dont_minimize*)**

Specifies whether the minimization tool should be invoked automatically after proof search. By default, automatic minimization takes place only if it can be done in a reasonable amount of time (as determined by the number of facts in the original proof and the time it took to find or preplay it) or the proof involves an unreasonably large number of facts.

See also *preplay_timeout* (§7.6) and *dont_preplay* (§7.6).

overlord $[= \langle bool \rangle]$ **{*false*}** **(neg.: *no_overlord*)**

Specifies whether Sledgehammer should put its temporary files in `$ISABELLE_HOME_USER`, which is useful for debugging Sledgehammer but also unsafe if several instances of the tool are run simultaneously. The files are identified by the prefix `prob_`; you may safely remove them after Sledgehammer has run.

See also *debug* (§7.4).

7.2 Problem Encoding

lam_trans $= \langle string \rangle$ **{*smart*}**

Specifies the λ translation scheme to use in ATP problems. The supported translation schemes are listed below:

- ***hide_lams*:** Hide the λ-abstractions by replacing them by unspecified fresh constants, effectively disabling all reasoning under λ-abstractions.

- ***lifting*:** Introduce a new supercombinator c for each cluster of n λ-abstractions, defined using an equation $\mathsf{c}\ x_1\ \ldots\ x_n = t$ (λ-lifting).
- ***combs*:** Rewrite lambdas to the Curry combinators (I, K, S, B, C). Combinators enable the ATPs to synthesize λ-terms but tend to yield bulkier formulas than λ-lifting: The translation is quadratic in the worst case, and the equational definitions of the combinators are very prolific in the context of resolution.
- ***combs_and_lifting*:** Introduce a new supercombinator c for each cluster of λ-abstractions and characterize it both using a lifted equation $\mathsf{c}\ x_1\ \ldots\ x_n = t$ and via Curry combinators.
- ***combs_or_lifting*:** For each cluster of λ-abstractions, heuristically choose between λ-lifting and Curry combinators.
- ***keep_lams*:** Keep the λ-abstractions in the generated problems. This is available only with provers that support the THF0 syntax.
- ***smart*:** The actual translation scheme used depends on the ATP and should be the most efficient scheme for that ATP.

For SMT solvers, the λ translation scheme is always *lifting*, irrespective of the value of this option.

uncurried_aliases $[= \langle$***smart_bool***$\rangle]$ **{*smart*}**
(neg.: *no_uncurried_aliases*)

Specifies whether fresh function symbols should be generated as aliases for applications of curried functions in ATP problems.

type_enc $= \langle$***string***$\rangle$ **{*smart*}**

Specifies the type encoding to use in ATP problems. Some of the type encodings are unsound, meaning that they can give rise to spurious proofs (unreconstructible using *metis*). The supported type encodings are listed below, with an indication of their soundness in parentheses. An asterisk (*) means that the encoding is slightly incomplete for reconstruction with *metis*, unless the *strict* option (described below) is enabled.

- ***erased*** **(very unsound):** No type information is supplied to the ATP, not even to resolve overloading. Types are simply erased.
- ***poly_guards*** **(sound):** Types are encoded using a predicate $\mathsf{g}(\tau, t)$ that guards bound variables. Constants are annotated with their types, supplied as additional arguments, to resolve overloading.
- ***poly_tags*** **(sound):** Each term and subterm is tagged with its type using a function $\mathsf{t}(\tau, t)$.

- ***poly_args* (unsound):** Like for *poly_guards* constants are annotated with their types to resolve overloading, but otherwise no type information is encoded. This coincides with the default encoding used by the *metis* command.
- ***raw_mono_guards*, *raw_mono_tags* (sound); *raw_mono_args* (unsound):**
  Similar to *poly_guards*, *poly_tags*, and *poly_args*, respectively, but the problem is additionally monomorphized, meaning that type variables are instantiated with heuristically chosen ground types. Monomorphization can simplify reasoning but also leads to larger fact bases, which can slow down the ATPs.
- ***mono_guards*, *mono_tags* (sound); *mono_args* (unsound):**
  Similar to *raw_mono_guards*, *raw_mono_tags*, and *raw_mono_args*, respectively but types are mangled in constant names instead of being supplied as ground term arguments. The binary predicate $\mathtt{g}(\tau, t)$ becomes a unary predicate $\mathtt{g\_}\tau(t)$, and the binary function $\mathtt{t}(\tau, t)$ becomes a unary function $\mathtt{t\_}\tau(t)$.
- ***mono_native* (sound):** Exploits native first-order types if the prover supports the TFF0, TFF1, or THF0 syntax; otherwise, falls back on *mono_guards*. The problem is monomorphized.
- ***mono_native_higher* (sound):** Exploits native higher-order types if the prover supports the THF0 syntax; otherwise, falls back on *mono_native* or *mono_guards*. The problem is monomorphized.
- ***poly_native* (sound):** Exploits native polymorphic first-order types if the prover supports the TFF1 syntax; otherwise, falls back on *mono_native*.
- ***poly_guards*?, *poly_tags*?, *raw_mono_guards*?, *raw_mono_tags*?, *mono_guards*?, *mono_tags*?, *mono_native*? (sound*):**
  The type encodings *poly_guards*, *poly_tags*, *raw_mono_guards*, *raw_mono_tags*, *mono_guards*, *mono_tags*, and *mono_native* are fully typed and sound. For each of these, Sledgehammer also provides a lighter variant identified by a question mark ('?') that detects and erases monotonic types, notably infinite types. (For *mono_native*, the types are not actually erased but rather replaced by a shared uniform type of individuals.) As argument to the *metis* proof method, the question mark is replaced by a "*_query*" suffix.
- ***poly_guards*??, *poly_tags*??, *raw_mono_guards*??, *raw_mono_tags*??, *mono_guards*??, *mono_tags*?? (sound*):**

Even lighter versions of the '?' encodings. As argument to the *metis* proof method, the '??' suffix is replaced by "*_query_query*".

- ***poly_guards@?*, *raw_mono_guards@?* (sound*):**
  Alternative versions of the '??' encodings. As argument to the *metis* proof method, the '@?' suffix is replaced by "*_at_query*".
- ***poly_guards!*, *poly_tags!*, *raw_mono_guards!*, *raw_mono_tags!*, *mono_guards!*, *mono_tags!*, *mono_native!*, *mono_native_higher!* (mildly unsound):**
  The type encodings *poly_guards*, *poly_tags*, *raw_mono_guards*, *raw_mono_tags*, *mono_guards*, *mono_tags*, *mono_native*, and *mono_native_higher* also admit a mildly unsound (but very efficient) variant identified by an exclamation mark ('!') that detects and erases erases all types except those that are clearly finite (e.g., *bool*). (For *mono_native* and *mono_native_higher*, the types are not actually erased but rather replaced by a shared uniform type of individuals.) As argument to the *metis* proof method, the exclamation mark is replaced by the suffix "*_bang*".
- ***poly_guards!!*, *poly_tags!!*, *raw_mono_guards!!*, *raw_mono_tags!!*, *mono_guards!!*, *mono_tags!!* (mildly unsound):**
  Even lighter versions of the '!' encodings. As argument to the *metis* proof method, the '!!' suffix is replaced by "*_bang_bang*".
- ***poly_guards@!*, *raw_mono_guards@!* (mildly unsound):**
  Alternative versions of the '!!' encodings. As argument to the *metis* proof method, the '@!' suffix is replaced by "*_at_bang*".
- ***smart*:** The actual encoding used depends on the ATP and should be the most efficient sound encoding for that ATP.

For SMT solvers, the type encoding is always *mono_native*, irrespective of the value of this option.

See also *max_new_mono_instances* (§7.3) and *max_mono_iters* (§7.3).

strict $[= \langle \textit{bool} \rangle]$ **{*false*}** **(neg.: *non_strict*)**

Specifies whether Sledgehammer should run in its strict mode. In that mode, sound type encodings marked with an asterisk (*) above are made complete for reconstruction with *metis*, at the cost of some clutter in the generated problems. This option has no effect if *type_enc* is deliberately set to an unsound encoding.

7.3 Relevance Filter

relevance_thresholds = ⟨***float_pair***⟩ **{0.45 0.85}**

Specifies the thresholds above which facts are considered relevant by the relevance filter. The first threshold is used for the first iteration of the relevance filter and the second threshold is used for the last iteration (if it is reached). The effective threshold is quadratically interpolated for the other iterations. Each threshold ranges from 0 to 1, where 0 means that all theorems are relevant and 1 only theorems that refer to previously seen constants.

max_relevant = ⟨***smart_int***⟩ **{*smart*}**

Specifies the maximum number of facts that may be returned by the relevance filter. If the option is set to *smart*, it is set to a value that was empirically found to be appropriate for the prover. A typical value would be 250.

max_new_mono_instances = ⟨***int***⟩ **{200}**

Specifies the maximum number of monomorphic instances to generate beyond *max_relevant*. The higher this limit is, the more monomorphic instances are potentially generated. Whether monomorphization takes place depends on the type encoding used.

See also *type_enc* (§7.2).

max_mono_iters = ⟨***int***⟩ **{3}**

Specifies the maximum number of iterations for the monomorphization fixpoint construction. The higher this limit is, the more monomorphic instances are potentially generated. Whether monomorphization takes place depends on the type encoding used.

See also *type_enc* (§7.2).

7.4 Output Format

verbose [= ⟨***bool***⟩] **{*false*}** **(neg.: *quiet*)**

Specifies whether the **sledgehammer** command should explain what it does. This option is implicitly disabled for automatic runs.

debug [= ⟨***bool***⟩] **{*false*}** **(neg.: *no_debug*)**

Specifies whether Sledgehammer should display additional debugging information beyond what *verbose* already displays. Enabling *debug*

also enables *verbose* and *blocking* (§7.1) behind the scenes. The *debug* option is implicitly disabled for automatic runs.

See also *overlord* (§7.1).

isar_proof [= ⟨***bool***⟩] **{*false*}** **(neg.: *no_isar_proof*)**

Specifies whether Isar proofs should be output in addition to one-liner *metis* proofs. Isar proof construction is still experimental and often fails; however, they are usually faster and sometimes more robust than *metis* proofs.

isar_shrink_factor = ⟨***int***⟩ **{1}**

Specifies the granularity of the Isar proof. A value of n indicates that each Isar proof step should correspond to a group of up to n consecutive proof steps in the ATP proof.

7.5 Authentication

expect = ⟨***string***⟩

Specifies the expected outcome, which must be one of the following:

- ***some*****:** Sledgehammer found a proof.
- ***none*****:** Sledgehammer found no proof.
- ***timeout*****:** Sledgehammer timed out.
- ***unknown*****:** Sledgehammer encountered some problem.

Sledgehammer emits an error (if *blocking* is enabled) or a warning (otherwise) if the actual outcome differs from the expected outcome. This option is useful for regression testing.

See also *blocking* (§7.1) and *timeout* (§7.6).

7.6 Timeouts

timeout = ⟨***float_or_none***⟩ **{30}**

Specifies the maximum number of seconds that the automatic provers should spend searching for a proof. This excludes problem preparation and is a soft limit. For historical reasons, the default value of this option can be overridden using the option "Sledgehammer: Time Limit" in Proof General's "Isabelle" menu.

preplay_timeout $= \langle$***float_or_none***$\rangle$ **{3}**

Specifies the maximum number of seconds that *metis* or *smt* should spend trying to "preplay" the found proof. If this option is set to 0, no preplaying takes place, and no timing information is displayed next to the suggested *metis* calls.

See also *minimize* (§7.1).

dont_preplay $[=$ ***true***$]$

Alias for "*preplay_timeout* = 0".

References

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

[2] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II—a cooperative automatic theorem prover for higher-order logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning: IJCAR 2008*, volume 5195 of *Lecture Notes in Computer Science*, pages 162–170. Springer-Verlag, 2008.

[3] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT '08*, ICPS, pages 1–5. ACM, 2008.

[4] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In K. R. M. Leino and M. Moskal, editors, *Boogie 2011*, pages 53–64, 2011.

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

[6] C. E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction — CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 147–161. Springer-Verlag, 2011.

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

[8] T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. Waldmeister: High-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.

[9] K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction — CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer-Verlag, 2011.

[10] K. Korovin. Instantiation-based automated reasoning: From theory to practice. In R. A. Schmidt, editor, *Automated Deduction — CADE-22*, volume 5663 of *LNAI*, pages 163–166. Springer, 2009.

[11] K. Korovin and C. Sticksel. iProver-Eq: An instantiation-based theorem prover with equality. In J. Giesl and R. Hähnle, editors, *Automated Reasoning: IJCAR 2010*, volume 6173 of *Lecture Notes in Computer Science*, pages 196–202. Springer-Verlag, 2010.

[12] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *Journal of AI Communications*, 15(2/3):91–110, 2002.

[13] S. Schulz. E—a brainiac theorem prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[14] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *Automated Deduction — CADE-12 International Conference*, LNAI 814, pages 341–355. Springer, 1994.

[15] G. Sutcliffe. ToFoF. `http://www.cs.miami.edu/~tptp/ATPSystems/ToFoF/`.

[16] G. Sutcliffe. System description: SystemOnTPTP. In D. McAllester, editor, *Automated Deduction — CADE-17 International Conference*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 406–410. Springer-Verlag, 2000.

[17] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. `http://www.spass-prover.org/publications/spass.pdf`.

[18] Z3: An efficient SMT solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.