Hammering Away

A User's Guide to Sledgehammer for Isabelle/HOL

Jasmin Blanchette
Institut für Informatik, Ludwig-Maximilians-Universität München

with contributions from

Martin Desharnais
Max-Planck-Institut für Informatik

Lawrence C. Paulson
Computer Laboratory, University of Cambridge

Lukas Bartl
Institut für Informatik, Universität Augsburg

January 18, 2026

Contents

1 Introduction

Sledgehammer is a tool that applies automatic theorem provers (ATPs), satisfiability-modulo-theories (SMT) solvers, and Isabelle proof methods on the current goal, mostly to find proofs but optionally also to refute the goal.[1] The supported ATPs include agsyHOL [12], Alt-Ergo [4], E [14], iProver [11], LEO-II [3], Leo-III [15], Satallax [7], SPASS [17], Vampire [13], Waldmeister [10], and Zipperposition [8]. The ATPs are run either locally or remotely via the SystemOnTPTP web service [16]. The supported SMT solvers are CVC4 [2], cvc5 [1], veriT [6], and Z3 [9]. These are always run locally. The supported proof methods are *algebra*, *argo*, *auto*, *blast*, *fastforce*, *force*, *linarith*, *meson*, *metis*, *order*, *presburger*, *satx*, and *simp*. The proof method support is experimental and disabled by default.

The problem passed to the ATPs, SMT solvers, or proof methods consists of your current goal together with a heuristic selection of facts (theorems) from the current theory context, filtered by likely relevance.

The result of a successful proof search is some source text that typically reconstructs the proof within Isabelle. The reconstructed proof often 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.

Sometimes the automatic provers might detect that the goal is inconsistent with the background facts—or even that the background facts are inconsistent regardless of of the goal.

For Isabelle/jEdit users, Sledgehammer provides an automatic mode that can be enabled via the "Auto Sledgehammer" option under "Plugins > Plugin Options > Isabelle > General." In this mode, a reduced version of Sledgehammer is run on every newly entered theorem for a few seconds.

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 the `src/HOL/Metis_Examples` directory. Comments and bug reports concerning Sledgehammer or this manual should be directed to the author at `jasmin.blanchette@gmail.com`.

[1] The distinction between ATPs and SMT solvers is mostly historical but convenient.

2 Installation

Sledgehammer is part of Isabelle, so you do not need to install it. However, it relies on third-party ATPs and SMT solvers.

Among the ATPs, agsyHOL, Alt-Ergo, E, LEO-II, Leo-III, Satallax, SPASS, Vampire, and Zipperposition can be run locally; in addition, agsyHOL, Alt-Ergo, E, iProver, LEO-II, Leo-III, Satallax, Vampire, Waldmeister, and Zipperposition are available remotely via SystemOnTPTP [16]. The SMT solvers CVC4, cvc5, veriT, and Z3 can be run locally.

There are three main ways to install ATPs or SMT solvers on your machine:

- If you installed an official Isabelle package, it should already include properly set up executables for CVC4, cvc5, E, SPASS, Vampire, veriT, Z3, and Zipperposition ready to use.
- Alternatively, you can download the Isabelle-aware CVC4, cvc5, E, SPASS, Vampire, veriT, Z3, and Zipperposition binary packages from `https://isabelle.in.tum.de/components/`. Extract the archives, then add a line to your `$ISABELLE_HOME_USER/etc/components`[2] file with the absolute path to the system. 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 agsyHOL, Alt-Ergo, E, LEO-II, Leo-III, Satallax, or Zipperposition manually, set the environment variable `AGSYHOL_HOME`, `E_HOME`, `LEO2_HOME`, `LEO3_HOME`, `SATALLAX_HOME`, or `ZIPPERPOSITION_HOME` to the directory that contains the `agsyHOL`, `eprover` (or `eprover-ho`), `leo`, `leo3`, `satallax`, or `zipperposition` executable; for Alt-Ergo, set the environment variable `WHY3_HOME` to the directory that contains the `why3` executable. Ideally, you should also set `E_VERSION`, `LEO2_VERSION`, `LEO3_VERSION`, `SATALLAX_VERSION`, or `ZIPPERPOSITION_VERSION` to the prover's version number (e.g., "3.6").

  Similarly, if you want to install CVC4, cvc5, veriT, or Z3, set the environment variable `CVC4_SOLVER`, `CVC5_SOLVER`, `ISABELLE_VERIT`, or `Z3_SOLVER` to the complete path of the executable, *including the file*

[2] 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.

name. Ideally, also set `CVC4_VERSION`, `CVC5_VERSION`, `VERIT_VERSION`, or `Z3_VERSION` to the solver's version number (e.g., "4.4.0").

To check whether the provers 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.

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 use the Sledgehammer panel in Isabelle/jEdit. Either way, Sledgehammer will produce something like the following output after a few seconds:

e found a proof...
cvc4 found a proof...
z3 found a proof...
vampire found a proof...
e: Try this: **by** *simp (0.3 ms)*
cvc4: Try this: **by** *simp (0.4 ms)*
z3: Try this: **by** *simp (0.5 ms)*
vampire: Try this: **by** *simp (0.3 ms)*

Sledgehammer ran CVC4, E, Vampire, Z3, and possibly other provers in parallel. The list may vary depending on which provers are installed and how many processor cores are available.

For each successful prover, Sledgehammer gives a one-line Isabelle proof. Rough 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 enabling the *isar_proofs* option (§ 7.4):

sledgehammer [*isar_proofs*]

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

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 particularly 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.

4.2 *Familiarize yourself with the main options*

Sledgehammer's options are fully documented in § 7. 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, SMT solvers, or proof methods) that should be run whenever Sledgehammer is invoked (e.g., "*provers = auto cvc4 e vampire zipperposition*").
- ***max_facts*** (§ 7.2) specifies the maximum number of facts that should be passed to the provers. By default, the value is prover-dependent and varies between 0 and about 1000.
- ***isar_proofs*** (§ 7.4) specifies that Isar proofs should be generated, in addition to one-line proofs. The length of the Isar proofs can be controlled by setting *compress* (§ 7.4).

- ***timeout*** (§ 7.6) controls the provers' time limit. It is set to 30 seconds by default.

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* (and any facts chained into the goal).

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 do not have any questions at this stage.

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

Sledgehammer heuristically selects a subset of lemmas from the currently loaded libraries. The component that performs this selection is called *relevance filter* (§ 7.2).

- The traditional relevance filter, *MePo* (Meng–Paulson), assigns a score to every available fact (lemma, theorem, definition, or axiom) based upon how many constants that fact shares with the goal. This process iterates to include facts relevant to those just accepted. The constants are weighted to give unusual ones greater significance. MePo copes best when the goal 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 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.
- An alternative to MePo is *MaSh* (Machine Learner for Sledgehammer). It applies machine learning to the problem of finding relevant facts.
- The *MeSh* filter combines MePo and MaSh. This is the default.

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_facts* option (§ 7.2) 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 syntax "(*add*: *my_facts*)". 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 a given iteration of MePo, they also influence which facts are selected at subsequent iterations.

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 for first-order logic with equality, but ATPs such as E, Vampire, and Zipperposition are higher-order, so Metis might fail at refinding their proofs.

In some rare cases, *metis* fails fairly quickly, and you get the error message "One-line proof reconstruction failed." This 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 (§ 7.3).

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

The *metis* (*full_types*) proof method and its relative *metis* (*mono_tags*) are fully-typed versions of Metis. They are somewhat slower than *metis*, but the

proof search is fully typed, and it also includes more powerful rules such as the axiom "$x = \mathsf{True} \lor x = \mathsf{False}$" for reasoning in higher-order positions (e.g., in set comprehensions). The method is tried 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. 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 (§ 7.3) 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.

5.4 *And what are the lifting and opaque_lifting 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. See the *lam_trans* option (§ 7.3) for details.

5.5 *Are the generated proofs minimal?*

Automatic provers frequently use many more facts than are necessary. Sledgehammer includes a proof minimization tool that takes a set of facts returned by a prover and repeatedly calls a prover or proof method with subsets of those facts to find a minimal set. Reducing the number of facts typically helps reconstruction and declutters the proof documents.

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

Sledgehammer tries to give informative error messages. Please report any strange error to the author at `jasmin.blanchette@gmail.com`.

5.7 *Why are there so many options?*

Sledgehammer's philosophy is that it should work out of the box, without user guidance. Most of the options are meant to be used by the Sledgehammer developers for experiments.

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*⟩?

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.
- ***supported_provers*:** Prints the list of automatic provers supported by Sledgehammer. See § 2 and § 7.1 for more information on how to install ATPs and SMT solvers.
- ***refresh_tptp*:** Refreshes the list of remote ATPs available at System-OnTPTP [16].

In addition, the following subcommands provide finer control over machine learning with MaSh:

- ***unlearn*:** Resets MaSh, erasing any persistent state.
- ***learn_isar*:** Invokes MaSh on the current theory to process all the available facts, learning from their Isabelle/Isar proofs. This happens automatically at Sledgehammer invocations if the *learn* option (§ 7.2) is enabled.
- ***learn_prover*:** Invokes MaSh on the current theory to process all the available facts, learning from proofs generated by automatic provers. The prover to use and its timeout can be set using the *prover* (§ 7.1) and *timeout* (§ 7.6) options.

- ***relearn_isar*:** Same as *unlearn* followed by *learn_isar*.
- ***relearn_prover*:** Same as *unlearn* followed by *learn_prover*.

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_proofs*, *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.

If you use Isabelle/jEdit, Sledgehammer also provides an automatic mode that can be enabled via the "Auto Sledgehammer" option under "Plugins > Plugin Options > Isabelle > General." For automatic runs, only the first prover set using *provers* (§ 7.1) is considered, *dont_slice* (§ 7.6) is set, fewer facts are passed to the prover, *fact_filter* (§ 7.2) is set to *mepo*, *strict* (§ 7.3) is enabled, *verbose* (§ 7.4) and *debug* (§ 7.4) are disabled, and *timeout* (§ 7.6) is superseded by the "Auto Time Limit" option in jEdit. 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 (§ 7.3) and at most one type encoding specification with the same semantics as Sledgehammer's *type_enc*

option (§ 7.3). The supported λ translation schemes are *opaque_lifting*, *lifting*, and *combs* (the default). All the untyped type encodings listed in § 7.3 are supported. For convenience, the following aliases are provided:

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

The *metis* method also supports the Isabelle option [[*metis_instantiate*]], which tells *metis* to infer and suggest instantiations of facts using **of** from a successful proof.

7 Option Reference

Sledgehammer's options are categorized as follows: mode of operation (§ 7.1), problem encoding (§ 7.3), relevance filter (§ 7.2), output format (§ 7.4), regression testing (§ 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***⟩: A floating-point number (e.g., 2.5 or 60) expressing a number of seconds.
- ⟨***float_pair***⟩: A pair of floating-point numbers (e.g., 0.6 0.95).
- ⟨***smart_int***⟩: An integer or *smart*.

Default values are indicated in curly brackets ({}). Boolean options have a negative counterpart (e.g., *minimize* vs. *dont_minimize*). When setting Boolean options or their negative counterparts, "= *true*" may be omitted.

7.1 Mode of Operation

[***provers*** =] ⟨***string***⟩

Specifies the automatic provers to use as a space-separated list (e.g., "*auto cvc4 e spass vampire*"). Provers can be run locally or remotely; see § 2 for installation instructions.

The following local ATPs and SMT solvers are supported:

- ***agsyhol*:** agsyHOL is an automatic higher-order prover developed by Fredrik Lindblad [12]. To use agsyHOL, set the environment variable `AGSYHOL_HOME` to the directory that contains the `agsyHOL` executable.
- ***alt_ergo*:** Alt-Ergo is a polymorphic ATP developed by Bobot et al. [4]. It supports the TPTP polymorphic typed first-order format (TF1) via Why3 [5]. To use Alt-Ergo, set the environment variable `WHY3_HOME` to the directory that contains the `why3` executable. Sledgehammer requires Alt-Ergo 0.95.2 and Why3 0.83.
- ***cvc4*:** CVC4 is an SMT solver developed by Barrett et al. [2]. To use CVC4, set the environment variable `CVC4_SOLVER` to the complete path of the executable, including the file name, or install the prebuilt CVC4 package from `https://isabelle.in.tum.de/components/`.
- ***cvc5*:** cvc5 is an SMT solver developed by Barbosa et al. [1]. To use cvc5, set the environment variable `CVC5_SOLVER` to the complete path of the executable, including the file name.
- ***e*:** E is a higher-order superposition prover developed by Stephan Schulz [14]. 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., "3.0"), or install the prebuilt E package from `https://isabelle.in.tum.de/components/`.
- ***iprover*:** iProver is a first-order instantiation-based prover developed by Konstantin Korovin [11]. To use iProver, set the environment variable `IPROVER_HOME` to the directory that contains the `iproveropt` executable. iProver depends on Vampire to clausify problems, so make sure that Vampire is installed as well.
- ***leo2*:** LEO-II is an automatic higher-order prover developed by Christoph Benzmüller et al. [3], with support for the TPTP typed higher-order syntax (TH0). To use LEO-II, set the environment

variable `LEO2_HOME` to the directory that contains the `leo` executable.

- ***leo3*:** Leo-III is an automatic higher-order prover developed by Alexander Steen, Christoph Benzmüller, et al. [15], with support for the TPTP typed higher-order syntax (TH0). To use Leo-III, set the environment variable `LEO3_HOME` to the directory that contains the `leo3` executable.
- ***satallax*:** Satallax is an automatic higher-order prover developed by Chad Brown et al. [7], with support for the TPTP typed higher-order syntax (TH0). To use Satallax, set the environment variable `SATALLAX_HOME` to the directory that contains the `satallax` executable.
- ***spass*:** SPASS is a first-order superposition 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 `https://isabelle.in.tum.de/components/`.
- ***vampire*:** Vampire is a higher-order superposition prover developed by Andrei Voronkov and his colleagues [13]. 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., "4.8").
- ***verit*:** veriT [6] is a first-order SMT solver developed by David Déharbe, Pascal Fontaine, and their colleagues. It is designed to produce detailed proofs for reconstruction in proof assistants. To use veriT, set the environment variable `ISABELLE_VERIT` to the complete path of the executable, including the file name.
- ***z3*:** Z3 is an SMT solver developed at Microsoft Research [9]. To use Z3, set the environment variable `Z3_SOLVER` to the complete path of the executable, including the file name.
- ***zipperposition*:** Zipperposition [8] is a higher-order superposition prover developed by Simon Cruanes, Petar Vukmirović, and colleagues. To use Zipperposition, set the environment variable `ZIPPERPOSITION_HOME` to the directory that contains the `zipperposition` executable and `ZIPPERPOSITION_VERSION` to the version number (e.g., "2.1").

The following remote ATPs are supported:

- ***remote_agsyhol*:** The remote version of agsyHOL runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_alt_ergo*:** The remote version of Alt-Ergo runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_e*:** The remote version of E runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_iprover*:** The remote version of iProver 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_leo3*:** The remote version of Leo-III runs on Geoff Sutcliffe's Miami servers [16].
- ***remote_waldmeister*:** Waldmeister is a unit equality prover developed by Hillenbrand et al. [10]. 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_zipperposition*:** The remote version of Zipperposition runs on Geoff Sutcliffe's Miami servers.

By default, *provers* is set to a subset of CVC4, E, SPASS, Vampire, veriT, Z3, and Zipperposition, to be run in parallel, either locally or remotely—depending on the number of processor cores available and on which provers are actually installed. Proof methods are currently not included, due to their experimental status. (Proof methods can nevertheless appear in Isabelle proofs that reconstruct proofs originally found by ATPs or SMT solvers.)

The following proof methods are supported: ***algebra***, ***argo***, ***auto***, ***blast***, ***fastforce***, ***force***, ***linarith***, ***meson***, ***metis***, ***order***, ***presburger***, ***satx***, ***simp***.

prover = ⟨***string***⟩

Alias for *provers*.

cache_dir = ⟨***string***⟩ **{ ""}**

Specifies whether Sledgehammer should cache the result of the external provers or not and, if yes, where. If the option is set to the empty string (i.e., ""), then no caching takes place. Otherwise, the string is interpreted as a path to a directory where the cached result will be

saved. The content of the cache directory can be deleted at any time to reset the cache.

falsify [= ⟨***smart_bool***⟩] **{*false*}** **(neg.: *dont_falsify*)**

Specifies whether Sledgehammer should look for falsifications or for proofs. If the option is set to *smart*, it looks for both.

A falsification indicates that the goal, taken as an axiom, would be inconsistent with some specific background facts if it were added as a lemma, indicating a likely issue with the goal. Sometimes the inconsistency involves only the background theory; this might happen, for example, if a flawed axiom is used or if a flawed lemma was introduced with **sorry**.

abduce = ⟨***smart_int***⟩ **{*0*}**

Specifies the maximum number of candidate missing assumptions that may be displayed. These hypotheses are discovered heuristically by a process called abduction (which stands in contrast to deduction)—that is, they are guessed and found to be sufficient to prove the goal.

Abduction is currently supported only by E. If the option is set to *smart*, abduction is enabled only in some of the E time slices, and at most one candidate missing assumption is displayed. You can disable abduction altogether by setting the option to 0 or enable it in all time slices by setting it to a nonzero value.

dont_abduce [= ***true***]

Alias for "*abduce* = 0".

minimize [= ⟨***bool***⟩] **{*true*}** **(neg.: *dont_minimize*)**

Specifies whether the proof minimization tool should be invoked automatically after proof search.

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

spy [= ⟨***bool***⟩] **{*false*}** **(neg.: *dont_spy*)**

Specifies whether Sledgehammer should record statistics in `$ISABELLE_HOME_USER/spy_sledgehammer`. These statistics can be useful to the developers of Sledgehammer. If you are willing to have your interactions recorded in the name of science, please enable this feature and send the statistics file every now and then to the author of this manual (`jasmin.blanchette@gmail.com`). To change the default value of

this option globally, set the environment variable `SLEDGEHAMMER_SPY` to *yes*.

See also *debug* (§ 7.4).

overlord [= ⟨***bool***⟩] **{*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 prefixes `prob_` and `mash_`; you may safely remove them after Sledgehammer has run.

Warning: This option is not thread-safe. Use at your own risks.

See also *debug* (§ 7.4).

7.2 Relevance Filter

fact_filter = ⟨***string***⟩ **{*smart*}**

Specifies the relevance filter to use. The following filters are available:

- ***mepo*:** The traditional memoryless MePo relevance filter.
- ***mash*:** The MaSh machine learner. Three learning algorithms are provided:
  - ***nb*** is an implementation of naive Bayes.
  - ***knn*** is an implementation of k-nearest neighbors.
  - ***nb_knn*** (also called ***yes*** and ***sml***) is a combination of naive Bayes and k-nearest neighbors.

  In addition, the special value *none* is used to disable machine learning by default (cf. *smart* below).

  The default algorithm is *nb_knn*. The algorithm can be selected by setting the "MaSh" option under "Plugins > Plugin Options > Isabelle > General" in Isabelle/jEdit. Persistent data for both algorithms is stored in the directory `$ISABELLE_HOME_USER/mash`.
- ***mesh*:** The MeSh filter, which combines the rankings from MePo and MaSh.
- ***smart*:** A combination of MePo, MaSh, and MeSh. If the learning algorithm is set to be *none*, *smart* behaves like MePo.

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

Specifies the maximum number of facts that may be returned by the relevance filter. If the option is set to *smart* (the default), it effectively takes a value that was empirically found to be appropriate for the prover. Typical values lie between 0 and 1000.

fact_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.

learn [= ⟨***bool***⟩] **{*true*}** **(neg.: *dont_learn*)**

Specifies whether Sledgehammer invocations should run MaSh to learn the available theories (and hence provide more accurate results). Learning takes place only if MaSh is enabled.

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

Specifies the maximum number of monomorphic instances to generate beyond *max_facts*. The higher this limit is, the more monomorphic instances are potentially generated. Whether monomorphization takes place depends on the prover and possibly the specified type encoding. If the option is set to *smart* (the default), it takes a value that was empirically found to be appropriate for the prover. For most provers, this value is 100.

See also *type_enc* (§ 7.3).

max_mono_iters = ⟨***int***⟩ **{*smart*}**

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 prover and possibly the specified type encoding. If the option is set to *smart* (the default), it takes a value that was empirically found to be appropriate for the prover.

See also *type_enc* (§ 7.3).

induction_rules = ⟨***string***⟩ **{*exclude*}**

Specifies whether induction rules should be considered as relevant facts. The following behaviors are available:

- ***exclude*:** Induction rules are ignored by the relevance filter.
- ***instantiate*:** Induction rules are instantiated based on the goal and then considered by the relevance filter.
- ***include*:** Induction rules are considered by the relevance filter.

7.3 Problem Encoding

lam_trans = ⟨***string***⟩ **{*smart*}**

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

- ***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).
- ***opaque_lifting*:** Same as *lifting*, except that the supercombinators are kept opaque, i.e. they are unspecified fresh constants. This effectively disables all reasoning under λ-abstractions.
- ***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.
- ***opaque_combs*:** Same as *combs*, except that the combinators are kept opaque, i.e. without equational definitions.
- ***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 λs.
- ***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 type encodings are listed below, with an indication of their soundness in parentheses. An asterisk (*) indicates that the encoding is slightly incomplete for reconstruction with *metis*, unless the *strict* option (described below) is enabled.

- ***erased*** **(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 extra 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 is the default encoding used by the *metis* proof method.
- ***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

$\mathsf{g}(\tau, t)$ becomes a unary predicate $\mathsf{g\_}\tau(t)$, and the binary function $\mathsf{t}(\tau, t)$ becomes a unary function $\mathsf{t\_}\tau(t)$.

- ***mono_native* (sound):** Exploits native first-order types if the prover supports the TF0, TF1, TH0, or TH1 syntax; otherwise, falls back on *mono_guards*. The problem is monomorphized.
- ***mono_native_fool* (sound):** Exploits native first-order types, including Booleans, if the prover supports the TFX0, TFX1, TH0, or TH1 syntax; otherwise, falls back on *mono_native*. The problem is monomorphized.
- ***mono_native_higher*, *mono_native_higher_fool* (sound):** Exploits native higher-order types, including Booleans if ending with "*_fool*", if the prover supports the TH0 syntax; otherwise, falls back on *mono_native* or *mono_native_fool*. The problem is monomorphized.
- ***poly_native*, *poly_native_fool*, *poly_native_higher*, *poly_native_higher_fool* (sound):** Exploits native first-order polymorphic types if the prover supports the TF1, TFX1, or TH1 syntax; otherwise, falls back on *mono_native*, *mono_native_fool*, *mono_native_higher*, or *mono_native_higher_fool*.
- ***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@*, *poly_tags@*, *raw_mono_guards@*, *raw_mono_tags@* (sound*):**

Alternative versions of the '??' encodings. As argument to the *metis* proof method, the '@' suffix is replaced by "*_at*".

- ***poly_args*?, *raw_mono_args*? (unsound):**
  Lighter versions of *poly_args* and *raw_mono_args*.
- ***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* (§ 7.2) and *max_mono_iters* (§ 7.2).

strict [= ⟨***bool***⟩] **{*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.

7.4 Output Format

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

Specifies whether the **sledgehammer** command should explain what it does.

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

Specifies whether Sledgehammer should display additional debugging information beyond what *verbose* already displays. Enabling *debug* also enables *verbose* behind the scenes.

See also *spy* (§ 7.1) and *overlord* (§ 7.1).

max_proofs = ⟨***int***⟩ **{4}**

Specifies the maximum number of proofs to display before stopping. This is a soft limit.

isar_proofs [= ⟨***smart_bool***⟩] **{*smart*}** **(neg.: *no_isar_proofs*)**

Specifies whether Isar proofs should be output in addition to one-line proofs. The construction of Isar proof is still experimental and may sometimes fail; however, when they succeed they can be faster and sometimes more intelligible than one-line proofs. If the option is set to

smart (the default), Isar proofs are generated only when no working one-line proof is available.

compress = ⟨***int***⟩ **{*smart*}**

Specifies the granularity of the generated Isar proofs if *isar_proofs* is explicitly enabled. 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. If the option is set to *smart* (the default), the compression factor is 10 if the *isar_proofs* option is explicitly enabled; otherwise, it is ∞.

dont_compress [= ***true***]

Alias for "*compress* = 1".

try0 [= ⟨***bool***⟩] **{*true*}** **(neg.: *dont_try0*)**

Specifies whether standard proof methods such as *auto* and *blast* should be tried as alternatives to *metis* or *smt*. The collection of methods is roughly the same as for the **try0** command.

smt_proofs [= ⟨***bool***⟩] **{*true*}** **(neg.: *no_smt_proofs*)**

Specifies whether the *smt* proof method should be tried in addition to Isabelle's built-in proof methods.

instantiate [= ⟨***smart_bool***⟩] **{*smart*}** **(neg.: *dont_instantiate*)**

Specifies whether Metis should try to infer variable instantiations before proof reconstruction, which results in instantiations of facts using **of** (e.g. *map_prod_surj_on*[**of** *f A* "*f* `*A*" *g B* "*g* `*B*"]). This can make the proof methods faster and more intelligible. If the option is set to *smart* (the default), variable instantiations are inferred only if proof reconstruction failed or timed out.

7.5 Regression Testing

expect = ⟨***string***⟩

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

- ***some*:** Sledgehammer found a proof.
- ***some_preplayed*:** Sledgehammer found a proof that was successfully preplayed.
- ***none*:** Sledgehammer found no proof.

- ***timeout*:** Sledgehammer timed out.
- ***resources_out*:** Sledgehammer ran out of resources.
- ***unknown*:** Sledgehammer encountered some problem.

Sledgehammer emits an error if the actual outcome differs from the expected outcome. This option is useful for regression testing.

The expected outcomes are not mutually exclusive. More specifically, *some* is accepted whenever *some_preplayed* is accepted as the former has strictly fewer requirements than the later.

See also *timeout* (§ 7.6).

7.6 Timeouts

***timeout* = ⟨*float*⟩ {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.

***slices* = ⟨*int*⟩ {24 times the number of cores detected}**

Specifies the number of time slices. Time slices are the basic unit for prover invocations. They are divided among the available provers. A single prover invocation can occupy a single slice, two slices, or more, depending on the prover. Slicing (and thereby parallelism) can be disable by setting *slices* to 1. Since slicing is a valuable optimization, you should probably leave it enabled unless you are conducting experiments.

See also *verbose* (§ 7.4).

***dont_slice* [= *true*]**

Alias for "*slices* = 1".

***preplay_timeout* = ⟨*float*⟩ {1}**

Specifies the maximum number of seconds that *metis* or other proof methods 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 proof method calls.

See also *minimize* (§ 7.1).

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

Alias for "*preplay_timeout* = 0".

8 Mirabelle Testing Tool

The `isabelle mirabelle` tool executes Sledgehammer or other advisory tools (e.g., Nitpick) or proof methods (e.g., *auto*) on all subgoals emerging in a theory. It is typically used to measure the success rate of a proof tool on some benchmark. Its command-line usage is as follows:

```
Usage: isabelle mirabelle [OPTIONS] [SESSIONS ...]

  Options are:
    -A ACTION    add to list of actions
    -O DIR       output directory for log files (default:
      "mirabelle")
    -T THEORY    theory restriction: NAME or
      NAME[FIRST_LINE:LAST_LINE]
    -m INT       max. no. of calls to each action (0: unbounded)
      (default 0)
    -s INT       run actions on every nth goal (0: uniform
      distribution) (default 1)
    -t SECONDS   timeout in seconds for each action (default 30)
    ...

  Apply the given ACTIONs at all theories and proof steps of the
  specified sessions.

  The absence of theory restrictions (option -T) means to check all
  theories fully. Otherwise only the named theories are checked.
```

Option `-A ACTION` specifies an action to run on all subgoals. When specified multiple times, all actions are performed in parallel on all selected subgoals. Available actions are `arith`, `metis`, `quickcheck`, `sledgehammer`, `sledgehammer_filter`, and `try0`.

Option `-O DIR` specifies the output directory, which is created if needed. In this directory, a log file named "mirabelle.log" records the position of each tested subgoal and the result of executing the actions.

Option `-T THEORY` restricts the subgoals to those emerging from this theory. When not provided, all subgoals from are theories are selected. When provided multiple times, the union of all specified theories' subgoals is selected.

Option `-m INT` specifies a maximum number of goals on which the action are run.

Option `-s INT` specifies a stride, effectively running the actions on every nth goal.

Option `-t SECONDS` specifies a generic timeout that the actions may interpret differently.

More specific documentation about low-level options, the `ACTION` parameter, and its corresponding options can be found in the Isabelle tool usage by entering `isabelle mirabelle -?` on the command line.

The following subsections assume that the environment variable `AFP` is defined and points to a release of the Archive of Formal Proofs.

8.1 Example of Benchmarking Sledgehammer

```
isabelle mirabelle -d '$AFP' -O output \
  -A "sledgehammer[provers = e, timeout = 30]" \
  VeriComp
```

This command specifies to run the Sledgehammer action, using the E prover with a timeout of 30 seconds, on all subgoals emerging from all theory in the AFP session VeriComp. The results are written to `output/mirabelle.log`.

```
isabelle mirabelle -d '$AFP' -O output \
  -T Semantics -T Compiler \
  -A "sledgehammer[provers = e, timeout = 30]" \
  VeriComp
```

This command also specifies to run the Sledgehammer action, but this time only on subgoals emerging from theories Semantics or Compiler.

8.2 Example of Benchmarking Multiple Tools

```
isabelle mirabelle -d '$AFP' -O output -t 10 \
  -A "try0" -A "metis" \
  VeriComp
```

This command specifies two actions running the **try0** and **metis** commands, respectively, each with a timeout of 10 seconds. The results are written to `output/mirabelle.log`.

8.3 Example of Generating TPTP Files

```
isabelle mirabelle -d '$AFP' -O output \
  -A "sledgehammer[provers = e, timeout = 5, keep_probs = true]" \
  VeriComp
```

This command generates TPTP files using Sledgehammer. Since the file is generated at the very beginning of every Sledgehammer invocation, a timeout of five seconds making the prover fail faster speeds up processing the subgoals. The results are written in an action-specific subdirectory of the specified output directory (`output`). A TPTP file is generated for each subgoal.

References

[1] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems: TACAS 2022 (I)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[3] 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.

[4] 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.

[5] 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.

[6] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Automated Deduction — CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[7] 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.

[8] S. Cruanes. Logtk: A Logic ToolKit for automated reasoning, and its implementation. In *4th Workshop on Practical Aspects of Automated Reasoning, PAAR@IJCAR 2014, Vienna, Austria, 2014*, 2014. Presented at the Practical Aspects of Automated Reasoning (PAAR) workshop.

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

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

[11] 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.

[12] F. Lindblad. A focused sequent calculus for higher-order logic. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning — IJCAR 2014*, volume 8562 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2014.

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

[14] S. Schulz, S. Cruanes, and P. Vukmirović. Faster, higher, stronger: E 2.3. In P. Fontaine, editor, *Automated Deduction — CADE-27*, volume 11716 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2019.

[15] A. Steen, M. Wisniewski, and C. Benzmüller. Agent-based HOL reasoning. In G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, editors, *Mathematical Software – ICMS 2016*, volume 9725 of *LNCS*, pages 75–81. Springer, 2016.

[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. In R. A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.