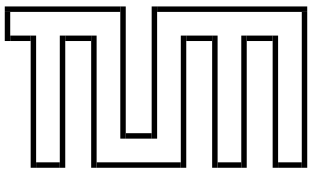


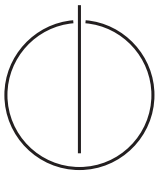
FAKULTÄT FÜR INFORMATIK

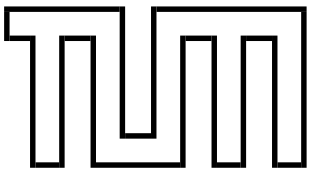
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Visualization Toolkit for Simplifier Traces in Isabelle/jEdit

Lars Hupel





FAKULTÄT FÜR INFORMATIK

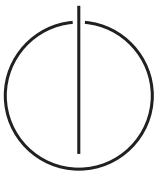
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Visualization Toolkit for Simplifier Traces in Isabelle/jEdit

Ein Visualisierungswerkzeug für Simplifier Traces in Isabelle/jEdit

Author:	Lars Hupel
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Dipl.-Inf. Lars Noschinski
Date:	July 29th, 2013



I assure the single handed composition of this master's thesis only supported by declared resources.

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, July 29th, 2013 Lars Hupel

Abstract

The Isabelle proof assistant comes equipped with some very powerful tactics to discharge goals automatically, or to at least simplify them significantly. One of these tactics is a rewriting engine, called the simplifier. It repeatedly applies rules to a term by replacing the left-hand side of an equation by the right-hand side.

While tremendously useful, the results of simplifying a term not always match the user's expectation: sometimes, the resulting term is not small enough, or the simplifier even failed to apply any rule. For these cases, the simplifier offers a trace which logs all steps which have been made.

However, these traces can be huge, especially because the library of Isabelle/HOL offers many pre-defined rewriting rules. It is often very difficult for the user to find the necessary piece of information about why and what exactly failed. Furthermore, there is no way to inspect or even influence the system while the simplification is still running. Hence, a simple, linear trace is not sufficient in these situations.

In this thesis, a new tracing facility is developed, which offers structure, interactivity and a high amount of configurability. It combines successful approaches from other logic languages and adapts them to the Isabelle setup. Furthermore, it fits neatly into the canonical IDE for Isabelle and is thus easy to use.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Isabelle is a generic theorem proving assistant [16, 17]. The system consists of several layers: fundamentally, it is implemented in the Standard ML programming language (*Isabelle/ML*), the primitive logic *Isabelle/Pure* resides on top of that and is in turn used to provide an implementation of higher-order logic (*Isabelle/HOL*).

1.1 Motivation

Isabelle comes with some very powerful tactics which are able to discharge large classes of proof goals automatically. This work is concerned with the rewriting tactic, often called the *simplifier*. It can be used to rewrite subterms according to a user-definable set of equations, which generally means simplifying a term to a normal form. These equations can have conditions which are recursively solved by the simplifier itself. Hence, there can be quite a huge number of steps between the original term and its normal form. Because of that complex work in the background, it ultimately is not obvious to the user how certain terms are derived from the input.

By default, this process is completely opaque to the user: the only observable effect is – given that the simplification succeeded – the (hopefully) simpler term it produced. If it failed, only an error message without any indication of the reasons is printed, or it might not even terminate at all. Thus, there is a *tracing* facility which can be selectively enabled by the user. It collects data about the steps the simplifier executed, and prints each of them without any high-level structure. The resulting trace can easily contain many screenfuls of items which the user has to laboriously search for interesting pieces of information.

Hence, an improved trace resembling elements of debuggers from imperative programming languages are desirable. The goal of this thesis is to implement exactly that: it should allow the user to define criteria with which the trace gets filtered, focused, or otherwise formatted to make it browsable more easily.

1.2 System Integration

The usual UI for the Isabelle system is the Isabelle/jEdit IDE which is implemented in the JVM language Scala (*Isabelle/jEdit*) [21]. It provides common IDE functionalities, including continuous proof checking which greatly increases interactivity.

The interplay between the ML process and the JVM process hosting the prover and the IDE, respectively, works via an underlying protocol, of which only a reasonably high-level interface is exposed [20, §3]. Interestingly enough, apart from providing distinct features, both sides also offer overlapping functionality. For example, general pretty-printing is implemented in ML, but a subset is also available in Scala. Bot sides provide means to encode somewhat arbitrary data into an XML representation, and the other way around.

The new simplifier trace works on both sides: there are new modules both in the ML and the JVM layer. Both sides require significant effort on top of the naïve approach of simply pre-rendering and sending all data to the IDE. Instead, a more sophisticated, asynchronous approach including staged filtering of messages has been taken.

1.3 Structure

The next section will cover important fundamental concepts from Isabelle. Aftwerwards, we will sketch the features of the new tracing, discuss its design goals, decisions and arising difficulties. Furthermore, this work is compared to related work. Then, we will go into the implementation by giving a high-level overview and walking through central, albeit simplified, code snippets. Before drawing a conclusion, a case study is presented which shows the flexibility of the implemented system. Finally, the appendix of this thesis contains a detailed user manual which explains the use of the tracing in the IDE.

2 Preliminaries

In this section, we will introduce general terminology and concurrency abstractions which have been used in this work.

2.1 Isabelle Theories

A single file of Isabelle source code is called a *theory*. Theories consist of declarations, theorems, proofs, types, and many more elements. While typing text, the prover (which runs in background) continually checks, the syntax and the validity of the proofs.

The smallest element in a theory is a *command*. Typically, a command is a keyword followed by some keyword-specific tokens. An example for a command is `apply simp`, which tells the prover to apply a tactic to the current goal, here: the simplifier tactic.

Conceptually, communication between the prover and the IDE works via the *document model*. In this model, commands can have attached *results*. Results are arbitrary pieces of data which may be interpreted by the IDE. For example, results carry the pretty-printing markup which is used to colour symbols, identifiers or numbers, highlight parts of the source code or display the outcome of the executed command. In order to identify results, they carry a *serial*, which is a unique number. Since the term "result" is highly ambiguous, it is only used in this way in section 4 and within an explicit context of "document" or "command".

2.2 Simplifier Terminology

A *(trace) message* is a piece of structured information about the current state of the simplifier. Each message has a *message type* which signifies its sematics (i. e. whether it means that a step is attempted or has failed). Messages requiring a response are called *interactive* messages. Whether a message requires a response does not depend on its contents, but only on its type and meta data. As a mere technical distinction, an interactive message which will be displayed on the user side is called a *question*.

A *rewrite rule* is a theorem which has a certain form (i. e. it is an equality). A *step* is one atomic application of a rewrite rule on a term. In the current implementation, this is defined as an invocation of the `rewritec` function. The result of a succeeding step is a theorem.

If a rewrite rule is *conditional*, its preconditions are solved by a *prover*. By default, this is just the simplifier itself. Hence, simplification is often nested: while one step is still running, the recursive call to the prover involves starting a new step.

2.3 Contexts

The Isabelle system uses *contexts* to track pieces of information. To quote one of the manuals [22, §1.1], they "represent the background that is required for formulating statements and composing proofs." At any point in the source code, they contain theorems, definitions, configuration data for proof tools and other declarations which have been stated up until that point.

There are basically two types of contexts, *theory* contexts and *proof* contexts. Both of them can be extended with arbitrary, typed data. We call that *theory* and *proof data*, respectively. For example, the simplifier uses the context to manage the set of rewrite rules (*simpset*), but also its current depth.

Theory data is – as the name indicates – attached to an Isabelle theory and can be changed by commands. For example, `datatype` introduces a couple of definitions and theorems. Furthermore, theory data needs to be *mergeable*, because theories can import from more than one other theory.

On the other hand, a proof context is created afresh every time a new proof is started, for example after a `lemma` statement. Proof data needs to provide an initialization function, taking the active theory as input.

2.4 Futures and Promises

The implementation of the new tracing heavily relies on a value-oriented concurrency mechanism provided by Isabelle/ML [15, 19]. In short, it deals with *future* values [1, 13]. This concept is reified by the type constructor `'a future`, which represents a value of type `'a` becoming available at a later time.

The high-level way to use futures is via `fork`: this function takes an (unevaluated) expression of type `'a -> unit` and immediately returns a handle of type `'a future`. Meanwhile, the system computes the value of the expression in the background, usually on a set of worker threads.

In order to observe the result of a future, the `join` function can be used. It blocks the calling thread until a result becomes available (which might be never).

Additionally, there are certain other operations which allow receivers to transform the value of a future without actually observing it. For example, the `map` function takes an `'a future`, a function `'a -> 'b`, and returns a `'b future` instantaneously. Blocking on that resulting future would wait for the original future and the subsequent function application to succeed.

Note that there are other ways to construct future values. However, for the receiver of a future, that is of no interest, since the operations on futures deal with these details automatically.

One example for this is when creating a future via `Future.promise`. While `fork` takes an expression and evaluates it in a background thread, `promise` takes no parameters and creates an "empty" future which can be *fulfilled* by other means. The corresponding function in Isabelle/ML is `Future.fulfill`, which takes a future and a value and stores that value in the future.

This abstraction is highly useful for concurrent programs which are not computation-heavy, but employ asynchronous techniques. Instead of having to deal with threads or shared state explicitly, abstract computations can be passed around and be transformed using blocking and non-blocking operations.

3 Supertrace – an Improved Simplifier Trace

Supertrace[1] is an interactive component of Isabelle and is – at the current stage of implementation – only available to the Isabelle/jEdit prover IDE. In this section, we will give an overview of the features of the system, discuss conceptual details and difficulties, and contrast our design with related work.

3.1 Design

The fundamental idea for tracing is to *instrument* the simplifier in order to collect data and influence the flow. The old tracing only does the former: at a set of certain points in the program flow, it prints messages indicating the current state. In Supertrace, we do the same on a higher level, by factoring out the processing into a separate module. The simplifier is then extended with some calls into that module.

To make the conceptual improvement more clear, consider an example. Previously, when the simplifier was invoked, the function `trace_term` was called, which took a `term` and a `string`. The result then was to print the fixed string "Simplifier invoked..." and the term. Now, the function `recurse` is called, also taking a `term`. However, the precise formatting of the message, or if it gets printed at all, is left as an implementation detail to the new module.

This has several advantages: Firstly, it reduces code clutter, because trace formatting and output is not mixed with the program logic. Furthermore, the semantics of the program flow is not lost, since it is clear that a call to `recurse` signifies a recursive invocation of the simplifier. Lastly, influencing the simplifier becomes easy, because these specialized methods can return a meaningful value, which can in turn be interpreted. In our implementation, calls to the tracer frequently return an updated `context`, like many other functions inside Isabelle do.

This conceptual shift does not readily lead to a change in the behaviour of the simplifier, though. The novel concept implemented in this thesis is that the user is queried *interactively* about the current state of the simplifier, instead of just being able to observe it. At certain points in the process, the system presents a question to the user, which has two effects: The simplifier gets stalled until the user answers to the question, and upon

[1] The name can be considered transitional.

answering, the program flow might deviate from the usual one, e. g. certain steps may be skipped.

To implement all this, all components of the system have been extended:

- A new module (`Supertrace`) has been added to the base logic of Isabelle, which adds numerous protocol messages.
- The existing simplifier has been amended to facilitate that module.
- The Isabelle/Scala layer has been modified to recognize the new SUPERTRACE messages, and a couple of managing modules were introduced.
- Finally, the Isabelle/jEdit IDE has got two new "views" to allow the user to inspect the current state.

The new tracing facility is highly configurable and goes to great lengths to keep the number of unwanted messages low. A secondary design goal was efficiency, so that the user does not experience delays which are longer than expected.

3.2 Interactivity

As already mentioned, the SUPERTRACE system is fundamentally interactive. It might present the user some questions about the current progress which it deems to be relevant. The user can then decide how to progress. This is quite like debuggers in imperative programming languages.

The important issue is how to determine which questions are relevant. Naturally, showing a question for each step is not feasible, because it might take thousands of steps until a term is rewritten into normal form. Hence, by default, the system shows no questions at all.

However, the user can accurately specify the set of interesting rewrite steps by defining *breakpoints*. If a step triggers such a breakpoint, the simplifier is intercepted and the system displays a question.

In debuggers for imperative languages, the concept of breakpoints is very well-known. Usually, breakpoints are set to lines in the source code, and when the sequential execution of the program hits a line, the execution is halted. Furthermore, many debuggers support conditional breakpoints, where users can specify a condition, and the breakpoint only triggers if that condition is met.

In SUPERTRACE, the implementation obviously has to differ from traditional debuggers, because it does not follow a strict sequential execution model. The principle is easy, though: each rewrite step in the simplifier consists of a term and a theorem. Breakpoints can be set for either of them. Term breakpoints usually contain schematic variables and trigger when it matches the term to be rewritten. For example, the breakpoint *?x > 0* matches when term *z > 0* is to be rewritten, where *z* can be any fixed or free variable. Term breakpoints can refer to locally bound variables.

On the other hand, a theorem breakpoint is triggered simply when its name is identical to the rewrite rule which is to be applied. As of the current implementation, it is not possible to set a breakpoint on unnamed facts. The reason for that is that in almost all cases, facts fed to the simplifier are slightly changed (e. g. replacing equality with meta-equality), hence a simple comparison of the propositions would fail.

3.3 Message Types

On a high level, these are the types of trace messages the simplifier can send:

recurse tells the system that the main entry point of the simplifier has been invoked.

step guards the application of a (potentially conditional) rewrite rule by the simplifier. It is invoked *before* the rule is applied, and the continuation of the simplification depends on the user input. *(interactive)*

hint indicates whether a rewrite step failed or succeeded. If it failed, the user is given a chance to inspect the failure, and can decide if the failing step should be tried again (with different settings). *(possibly interactive)*

ignore marks a specific part of the trace as obsolete. In this implementation, this message is only produced when the user does not wish to retry a failing step. It is generated by the system and thus cannot be sent explicitly.

log emits an arbitrary log message which will not be further interpreted by the system.

Every message consists of *meta data* and *payload*. The meta data is a list of key-value pairs (some of which are mandatory, refer to table 1), whereas the payload is a chunk of pretty-printed text. As can be seen in that table, messages form a hierarchy. This closely follows the actual, recursive nature of the simplifier.

Key	Value type	Message type	Description
`serial`	`int`	all	unique identifier of a message
`parent`	`int`	all	unique identifier of the parent message
`text`	`string`	all	short title of the message
`conditional`	`bool`	`step`	rewrite rule is conditional?
`trigger_thm`	`bool`	`step`	triggered by a theorem breakpoint
`trigger_term`	`bool`	`step`	triggered by a term breakpoint
`success`	`bool`	`hint`	triggered by a term breakpoint

Table 1: Key-value entries in messages

Messages can have children. For example, `step` messages are naturally associated with the `recurse` message emitted in the simplifier invocation. The trace keeps track of that relationship by adding information about a message's parent. Later on, the IDE will be able to reconstruct the tree structure prior to presenting the full trace output to the user.

3.4 Interactive Messages

As seen earlier, there are two types of interactive messages which allow the user to influence the outcome of the simplifier: one before a simplification step is attempted, and one for when a simplification step failed.

Message Type `step`: "Apply rewrite rule?"

When a step is attempted, the message contains the instantiated theorem, the term to rewrite, and a number of different possible replies. The user can choose to continue the computation, which instructs the simplifier to apply the specified rule and thus does not influence the result of the simplifier. The other option is to skip the current step, even if it would have succeeded.

As a result, the outcome of a simplification run is potentially different from when tracing would be disabled. Hence, skipping should be used sparingly: the most common use case would be to find overlapping rewrite rules.

Message Type `hint(failed)`: "Step failed"

Often, the user wishes for immediate feedback as soon as the simplification failed. Prior to this work, in case of failure the simplifier just does not produce any result, or produces an unwanted result.

On the other hand, with this message type, the new tracing provides new insight into the simplification process: It indicates that the simplifier tried to solve the preconditions of a rewrite rule, but failed. There are a number of different reasons for that, including that the preconditions do not hold or the simplifier could not solve them, or a wrong theorem has been produced in the recursive call, which usually indicates a problem in the tactic. Regardless of the reason, it is possible to *redo* a failed step if (and only if) the original step triggered a question previously.

Consider an example: The term $f\ t_1$ is to be rewritten. The rewrite rule $P_1\ x \Longrightarrow f\ x \equiv g\ x$ is applicable and gets instantiated to $P_1\ t_1 \Longrightarrow f\ t_1 \equiv g\ t_1$. Assume that there is a breakpoint on that particular rule, hence the user is presented a question whether the rule should be applied. The user chooses to continue, and the simplifier recursively tries to solve the precondition $P_1\ t_1$. Now assume that this entails application of another conditional rule which does not trigger a breakpoint (hence no question), but this step failed. In turn, the rewriting of $f\ t_1$ failed. The system now displays the "step failed" message to the user for the outermost failing step. Note that no messages are displayed for the other failing steps which caused the outermost one to fail. This is by design for two reasons:

- Often, the simplifier has to try multiple rules to prove a precondition. This is the case when there are multiple, overlapping rules for a predicate. Were the panel to notify the user for each of those steps, this would quickly become very confusing because of a flood of unrelated, and in the end, unimportant failures.
- If the innermost failure is several layers of recursions away from the original, interesting step, it becomes difficult for the user to establish a causal relationship between the previously answered "step" and the subsequent "failure" message.

Should the user choose to redo the computation, the simplifier state will be rolled back to *before* the last question. In the above example, the system would ask for the application of the rewrite rule $P_1\ t_1 \Longrightarrow f\ t_1 \equiv g\ t_1$ again. Of course, answering that question the same way a second time would not change anything. But it is possible to change the

settings and obtain more detailed information. This would actually cause the simplifier to run anew, which is a consequence of the message filtering (see section 3.6).

3.5 Memoization

Imagine a situation where the user realizes during a simplification run that a rule is missing. The user then adds the rule with `using` and obviously does not wish to be asked the same questions they already answered again. However, that expectation is not matched by the system, since any changes in the proof text causes the system to discard and reinitialize all active command states. This also happens in other circumstances, e. g. when the user moves the cursor position in the IDE.

Consider an analogy to debugging an imperative program: While stepping through the execution, an error is found and the source code is changed accordingly. After restarting the process, the user would like to continue at the same point (or a bit earlier) where the debugging session has been exited previously.[2] Granted, the global state of the execution might have changed, but as long as the local variables are the same, this seems to be safe enough.

In Supertrace, a *memoization* system helps in mitigating that issue by trying to reconstruct the original tracing state. Each time the user answers a `step` question, that answer is recorded in a (global) storage. When the same question appears again later, be it in the same simplification run or in another one, it is automatically answered identically.

Redoing a simplification step creates an interesting feature overlap with memoization. Since fundamentally, redoing a step makes the system display the original question again, a naïvely implemented cache would auto-answer that question. As a consequence, the (unchanged) computation would fail again, which would obviously reduce this question type to absurdity. Hence, care has been taken in the implementation so that the cache is partially purged in order to avoid that problem. For implementation details, refer to section 4.4.

Note that despite what the name “memory” might suggest, no fuzzy matching of any sort is done. At the moment, questions are compared using simple textual identity of

[2] In fact, the Java debugger of the *NetBeans* IDE offers a similar feature. Code changes while debugging can be applied to the running program, which requires reloading the affected classes in the JVM instance. This completely avoids the problem of reconstructing the original state after restarting the process, because the process is not even being terminated. Unfortunately, no further documentation of this facility seems to be available, so a thorough analysis cannot be made in this thesis.

their contents. If the text of a message is slightly different, it will not be considered. This is essentially a trade-off: the notion of "fuzziness" is extremely context-dependent. For example, for some predicates, a simple change of a constant is meaningless, whereas for other predicates, a conditional rule depends on it. Designing a reasonable fuzzy matcher is outside of the scope of this thesis, but is an interesting starting point for future work.

3.6 Message Filtering

Based on the settings described in the previous chapters, messages get *filtered*. The process is depicted in figure 1 and consists of multiple steps:[3]

1. Using *normal* verbosity, messages which have not been triggered by a breakpoint are discarded right in the beginning. This happens immediately after the simplifier created them. Due to performance reasons, discarded messages are explicitly not retained anywhere in the system. Unless tracing is disabled completely, accepted interactive messages are then transferred to the IDE, where they will be treated as questions.
2. If the tracer operates without user intervention (e. g. if the user explicitly disabled it earlier), questions are merely logged and answered with a default reply. The default reply is chosen so that it does not influence the simplifier in any way, i. e. it proceeds as if tracing would be disabled.
3. Some questions are eligible for memoization. At this point, the memory is queried to check for a match.
4. If *auto reply* is enabled (see figure 7 in section A) is selected, all remaining questions are also automatically answered with a default reply. Otherwise, they are finally being displayed. This is scoped to the current focus, i. e. only applies to the active questions of the selected command. A use case for this facility arises when interactive tracing is globally enabled, but the user wishes to discharge active questions of some selected commands without having to modify the proof document.

On a first glance, this pipeline might seem a little convoluted. However, these steps are necessary to match the user's expectation to only get asked if desired, which (ideally)

[3]Note that this is a high-level overview. In practice, there are multiple shortcuts in the implementation in order to avoid sending interactive messages which will be filtered out later in the pipeline.

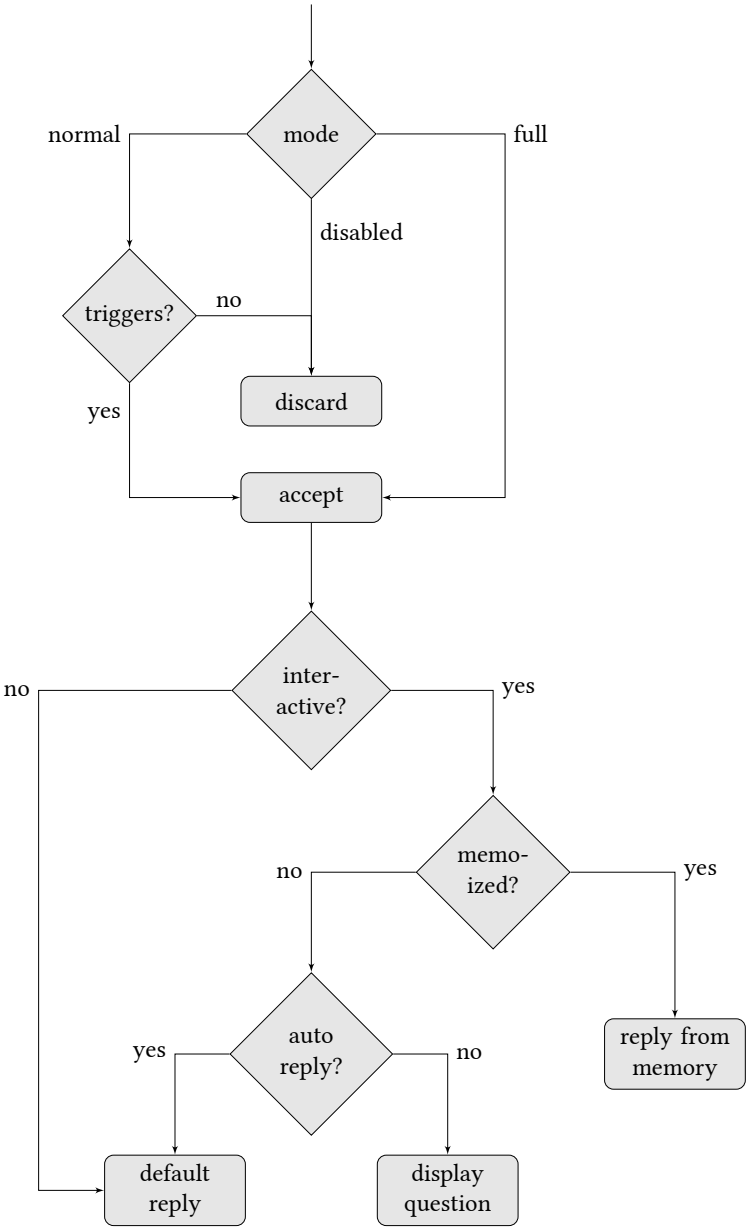


Figure 1: Message filtering

should happen rarely. Filtering helps keeping the number of unwanted messages at a minimum.

3.7 Related Work

In this section, we will compare the contribution of this thesis with the *SWI-Prolog* programming system and the *Maude* rewriting language. Both systems offer tracing and debugging facilities where the user is able to step through the computation.

3.7.1 Debugging and Tracing in SWI/Prolog

Prolog is a logic programming language [8, 18]. A program consists of a set of *clauses*, namely *rules* and *facts*. Rules are simple Horn clauses, with a head and a body. Facts are merely rules with an empty body. Heads may contain variables, and bodies may contain variables not occurring in the head. Variable names must begin with an upper-case letter or an underscore, whereas *atoms* must begin with a lower-case latter.

The example in listing 1 defines a program with two predicates, `child` and `descendant`. A query is basically a predicate with possibly uninstantiated variables, and Prolog tries to instantiate those. In Prolog terminology, such an expression is a *goal*, and the interpreter attempts to *prove* it. (This is similar to declaring a `schematic_lemma` in Isabelle.)

When proving a goal, Prolog tries to unify the current goal with any of the available clause heads, and proceeds recursively with each item in the body as new subgoals. This is similar to how the simplifier works in Isabelle: The left-hand side of a rewrite rule is matched to the current term, and if matches, it tries to solve the preconditions of the rule recursively.

The Prolog implementation *SWI-Prolog* provides a tracing facility for queries [10, §§2.9, 4.38]. An example for the tracing output can be seen in listing 1 (the term `creep` denotes continuing the normal process).[4]

Apart from continuing the process, SWI offers some additional commands. The commands relevant for this work are:

abort exits the whole proof attempt

[4] A discussion of tracing in Prolog can be found in [8, §8], and further analyses in [9]. SWI uses a slightly extended variant thereof.

```
child(a, b).
child(b, c).

descendant(X, X).
descendant(X, Z) :- child(X, Y), descendant(Y, Z).
```

(a) Input database

```
[trace]  ?- descendant(a, c).
   Call: (6) descendant(a, c) ? creep
   Call: (7) child(a, _G1949) ? creep
   Exit: (7) child(a, b) ? creep
   Call: (7) descendant(b, c) ? creep
   Call: (8) child(b, _G1949) ? creep
   Exit: (8) child(b, c) ? creep
   Call: (8) descendant(c, c) ? creep
   Exit: (8) descendant(c, c) ? creep
   Exit: (7) descendant(b, c) ? creep
   Exit: (6) descendant(a, c) ? creep
true ;
   Redo: (8) descendant(c, c) ? creep
   Call: (9) child(c, _G1949) ? creep
   Fail: (9) child(c, _G1949) ? creep
   Fail: (8) descendant(c, c) ? creep
   Fail: (7) descendant(b, c) ? creep
   Fail: (6) descendant(a, c) ? creep
false.
```

(b) Query with tracing enabled

Listing 1: Prolog example

fail the current goal is explicitly marked as failure, regardless whether it could have been proved

ignore the current goal is explicitly marked as success

retry discards the proof of a subgoal and backtracks to the original state of the parent goal

Albeit imaginable, marking a goal as success is not supported in SUPERTRACE. It is not possible per se to implement that in Isabelle, since the inference kernel would not allow such a step. The workaround would be to introduce such theorems "by cheating", i. e. the same strategy which is used by `sorry`. It is an interesting discussion whether or not to grant the simplifier the "privilege" to generate invalid theorems in tracing mode.

Finally, it is possible to declare breakpoints on predicates. SWI allows to refine breakpoints with the specific event (referred to as *port*). For example, the user can specify that they are interested only in `fail` messages, but not `call` messages. However, as soon as such a breakpoint is set, the tracing ceases to be interactive and switches to a log-only mode.[5] In SUPERTRACE, the filtering concept is more sophisticated and allows a fine-grained control over what is being asked, the precise conditions of which can also be modified during a simplification run. In SWI, it is also not possible to set a breakpoint on terms, or even on terms with schematic variables.

In summary, SWI's features are quite similar to what SUPERTRACE offers, but differ in a few conceptual points. First and foremost, the "execution" model of Prolog and Isabelle theories differ significantly. Evaluation of Prolog queries happens sequentially, and any changes in the underlying source code must be explicitly reloaded. That also means that running queries need to be aborted. On the other hand, using Isabelle/jEdit, changes in a theory get reloaded immediately and affects pending computations directly. Hence, a memory as implemented in SUPERTRACE is not necessary in Prolog.

3.7.2 Debugging and Tracing in Maude

Maude is a logic language based on term rewriting [6, 7]. A program consists of datatype declarations and equations. Then, the user can issue a `reduce` command, which successively applies rewrite rules to an input term. A short example modelling natural numbers can be seen in listing 2.

This snippet defines a data type for natural numbers, along with its two constructors `zero` and `s`. Additionally, an addition function and a predicate to check whether a number is non-zero and two equalities to (partially) defining that predicate. Note that the addition function does not have any defining equalities in this example (which are in fact unneeded).

Similarly to the Isabelle simplifier, rewrite rules can be *conditional*. In the trace, it becomes obvious that those are handled exactly like in Isabelle. A potentially applicable rule gets instantiated with the concrete term, and the preconditions are solved recursively.

However, how they appear in the trace is more interesting. As can be seen, the Maude tracing is purely sequential and offers little to no insight into the hierarchical structure.

[5] This is possible with SUPERTRACE, too. When in non-interactive mode, trace is only produced for steps matching breakpoints, but no questions are presented to the user. This is actually the default behaviour.

```
fmod SIMPLE-NAT is
  sort Nat .
  op zero : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  op nonzero_ : Nat -> Bool .

  vars N M : Nat .
  ceq nonzero (N + M) = true
    if nonzero N /\ nonzero M .
  eq nonzero (s N) = true .
endfm
```

(a) Programme (based on [6, §2.2])

```
Maude> reduce nonzero (s zero + s s zero) .
********** trial #1
ceq nonzero (N + M) = true
  if nonzero N = true /\ nonzero M = true .
N --> s zero
M --> s s zero
********** solving condition fragment
nonzero N = true
********** equation
eq nonzero s N = true .
N --> zero
nonzero s zero
--->
true
(...)
********** success #1
********** equation
(...)
nonzero (s zero + s s zero)
--->
true
```

(b) Reducing a term with trace enabled (excerpt)

Listing 2: Maude example

The trace can be tuned in various ways [6, §§14.1, 18.6]: for example, Maude allows filtering for named rules and operations (albeit only the outermost operation in the redex is considered). There is also a wealth of settings which control verbosity of the trace, e. g. whether solving preconditions or the definition of rules should be included in the trace.

Apart from controlling the textual output, it is also possible to enable output colouring, similarly to the highlighting in Isabelle. The major difference here is that Maude distinguishes between *constructor* and *nonconstructor* symbols, where the latter maps to regular "functions" in Isabelle/HOL parlance. An indicator for problems in the set of rewrite rules is when a term does not get fully rewritten, which is defined as nonconstructor symbols still occur after reducing [6, §14.1.2]. Hence, colouring symbols differently greatly improves debugging experience in Maude, because it also gives hints into when exactly a problem has been introduced.

Additionally to tracing, there is also a debugging facility. It can be configured with breakpoints in the same way as the tracing. When a breakpoint is hit, the user can resume or abort the whole computation, but also (on request) observe the call stack. The latter also

includes a textual explanation, e. g. that the current term is being rewritten to solve a precondition.

A distinguishing feature of the debugger is that it allows to execute a new `reduce` command when a debugging session is active. This allows the user to quickly check related terms and hence better understand an issue with the original term.

Maude has been an active target for research for refining the trace even further, providing insights into when and how a particular term emerged during reducing (refer to e. g. [2] and related work).

4 Implementation

In this section, the underlying implementation will be explored. Unless otherwise noted, all presented code snippets are excerpts from the actual code with reduced complexity and represent original work by the author of this thesis.

4.1 Interaction Model

The fundamental idea is that the simplifier communicates bi-directionally with the user working with the IDE, where the `Supertrace` ML module acts as a "broker". The simplifier continually emits tracing information in a structured and hierarchical manner, but only with a limited amount of concrete formatting. For some types of messages, it also expects a *response*, i. e. some guidance how the rewriting process should proceed. (Other types of messages are merely for reporting purposes , e. g. that a certain step succeeded.)

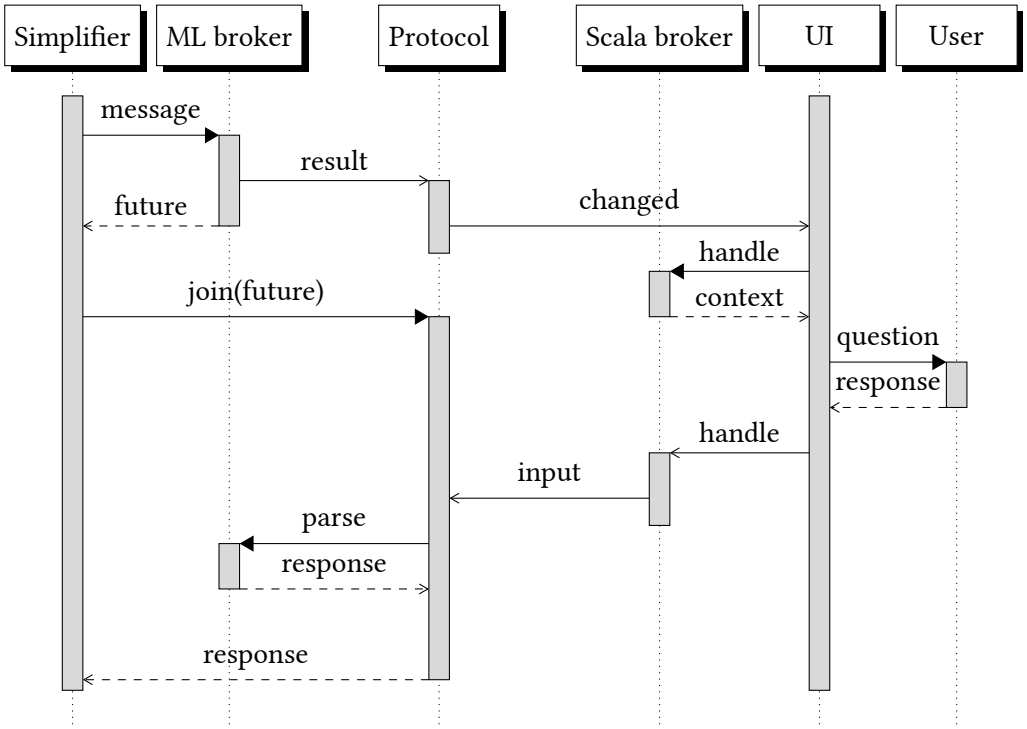
The ML broker then decides, based on user configuration, what should happen to messages. In the normal, interactive mode, the messages get filtered for certain criteria, formatted (using the built-in pretty-printing mechanism), and subsequently sent to the IDE. If a message got filtered out, the simplifier gets a *default* response. If not, the ML broker waits for an actual response from the IDE, which it parses and sends back to the simplifier.

On the IDE side, there is also a broker which acts as a counterpart to the ML broker. It collects the messages from the simplifier and provides a high-level view on the set of current messages, and forwards responses from the user, or even responds early if necessary.

This whole process is completely transparent to the simplification engine. Hence it would be somewhat trivial to change the current implementation to, say, post the messages to a web service instead of to the user.

There is another type of interaction happening outside that model, but this is currently only used for notifying the IDE of obsolete messages for which the broker does not expect a response any more.

In the following sections, we will describe all stages of the user interaction, starting from the simplifier emitting trace and ending with the user clicking on a button. The overall process of a single interactive message is depicted as a sequence diagram in figure 2. The



Protocol refers to the low-level communication between the ML and the JVM process, thick arrows denote function calls with return values (with the return value denoted by a dashed line below), thin arrows denote function calls without return values.

Figure 2: Sequence diagram of the interaction

description will follow the implementation, i. e. not in chronological order of the steps of the interaction.

The general interplay between the prover and the IDE is laid out in [20, 23].

4.2 Simplifier interaction

Sending a non-interactive message is trivial: it merely involves calling a function, but its result can be ignored, since no answer is expected.

On the other hand, sending an interactive message requires waiting for the user. As already indicated, communication between the simplifier and the user happens in at least two stages. The first stage is the most straightforward one, as it does not have to cross any language or process barrier, because both the simplifier and the broker live inside the

prover in the same (ML) world. Hence, "emitting" an interactive trace message from the simplifier is as easy as for a non-interactive one. However, instead of returning `unit`, the corresponding function will immediately return an asynchronous future (recall figure 2). Meanwhile, the message is transmitted to the IDE, and is waiting for the user to act upon it. As soon as that happens, the future value becomes available, which in turn allows the simplifier to observe the user's response.

At the moment, the simplifier is purely sequential, i. e. it needs to know the response in order to continue, instead of being able to do other meaningful work (e. g. speculatively trying to solve the preconditions in parallel). As a result, the simplifier has to block on the future right after receiving it. However, the infrastructure in this work is explicitly designed to take parallelism into account. If at some point the simplifier will be parallelized, it would be easy to present multiple queries to the user simultaneously (see section 5 for a case study).

Previously, the trace was *read only*, i. e. merely presenting the user its activity. Now, the user can actively *influence* the outcome of the process by two means:

1. specifying whether a given rewrite rule should be applied to a given term
2. requesting that a failed step should be redone

In fact, the user has more ways to interact with the system, but those are about the level of detail in the trace, which is of no concern to the simplifier.[6]

Observe that the actual work – invoking the prover on the preconditions of a rewrite rule – is *bracketed* by two messages: the `step` and `hint` messages are sent beforehand and afterwards, respectively (as depicted in figure 3). In total, that can amount to two interactive messages, and depending on the outcome of `hint` in case of failure, even more. Because of that, the resulting control flow gets quite complex. Its execution is therefore abstracted in the function `supertrace_if_continue` (see listing 3 for the stripped-down implementation) which gets called whenever the simplifier attempts to apply a rewrite rule.

The function takes a `context`, the term to be simplified, the candidate rewrite rule, and two *continuations* for invoking the prover and skipping the step, respectively.

As a consequence, this means that the necessary amount of changes for hooking into the simplifier is greatly reduced: it operates as it used to, except that some calls are wrapped

[6]The simplifier naïvely emits messages for each step, whereas the filtering is done by the system.

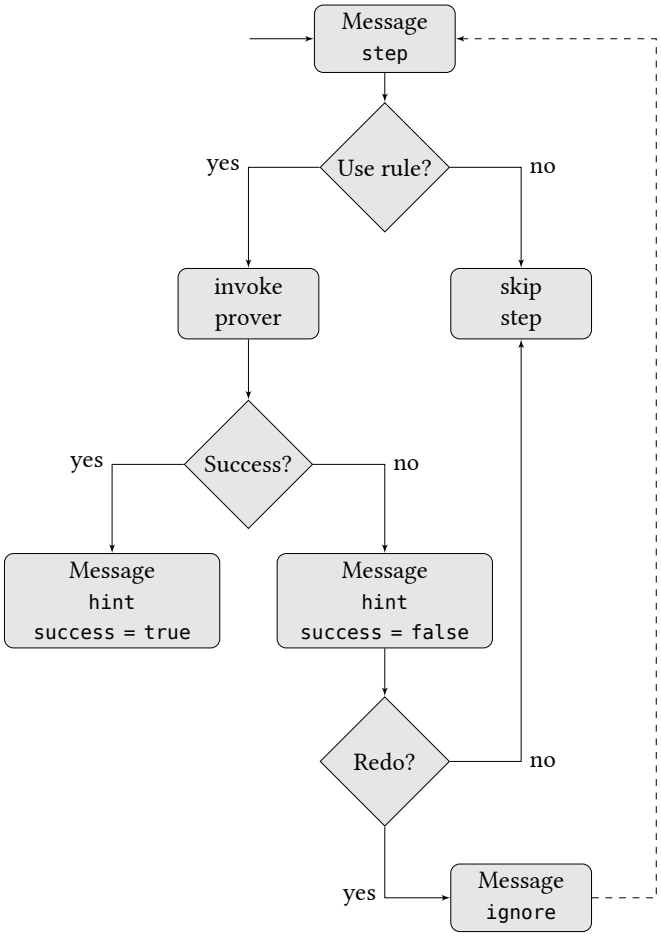


Figure 3: Bracketed execution of a rewrite rule

by that function. For example, should the simplification step fail, and the user requests redoing the step, the function merely calls itself, without giving control flow back to the simplifier.

4.3 System Interaction

While the simplifier only emits messages, the "heavy lifting" is done by the module `Supertrace` which lives inside *Isabelle/Pure*. Its tasks are to:

- store the current configuration,

```
fun supertrace_if_continue ctxt term thm skip cont =
  let
    val data = (* ... *)
  in
    case Future.join (Supertrace.step data) of
      Skip =>
        skip ctxt
    | Continue ctxt' =>
        let val res = cont ctxt'
        in
          case res of
            NONE =>
              (case Future.join (Supertrace.indicate_failure data ctxt') of
                 Exit =>
                   skip ctxt'
               | Redo =>
                   supertrace_if_continue ctxt term thm skip cont)
          | SOME (thm, _) =>
              (Supertrace.indicate_success thm ctxt'; res)
        end
  end
```

Listing 3: Implementation of the bracketed execution

- track the state of the simplification process, i. e. record the `recurse` message to preserve the hierarchical call stack,
- filter the `step` messages for potentially interesting content; not all of them are being presented to the user,
- format high-level messages and converting them to an interchange format (XML), and
- send data to and receive data from the IDE, while maintaining a list of interactive messages with pending responses

4.3.1 Data Model

To fulfil these tasks, the module needs to attach certain pieces of data to the proof context, following the usual Isabelle conventions of declaring appropriate structures in the module. As an implementation detail, that is split up into three types (listing 4):

```
datatype mode = Disabled | Legacy | Normal | Full

type config =
  {max_depth: int,
   init_mode: mode,
   init_interactive: bool}

type runtime =
  {max_depth: int,
   depth: int,
   mode: mode,
   interactive: bool,
   parent: int}

type breakpoints =
  {terms: term Net.net,
   thms: string Ord_List.T}
```

Listing 4: Excerpt of the context data

config Global, mergable configuration. This contains all options specified via the `supertrace` attribute (see section A.2).

breakpoints Term and theorem breakpoints, declared as *generic* data, i. e. exists at the theory level, but allows for local changes in a proof block.

runtime Local configuration and runtime data. This is separate from the other structures because configuration can be changed inside of a proof or even during a single invocation (see section A.3) by the user and by the system, but these changes should not affect the toplevel, and runtime data is strictly local, i. e. does not exist at the toplevel.[7]

For changing `config` and `runtime`, the module exposes certain attributes (listing 5), which are in turn set up in a theory file.

[7] `runtime` cannot be defined as generic data because firstly, it is not mergeable and secondly, the mergeable parts need to be changed inside a `context`, and the ML signature of `Generic_Data` does not provide a `put` function on `context`, but only on `generic`. This leads to the somewhat clunky duplication of certain fields. Obviously, the `put` function returns a wrapped `context` again when passed in a wrapped `context`, but that is not communicated in the type signature. Relying on that behaviour is therefore considered unsafe, and the proper fix would be to amend the signature of `Generic_Data`.

```
val config: string -> bool -> int -> attribute
val config_legacy: int -> attribute
val thm_breakpoint: attribute
val term_breakpoint: term list -> attribute
```

Listing 5: Attributes in the `Supertrace` module

4.3.2 Public Message Interface

The module aims to provide a neutral interface which is not tied to the existing simplifier in any way.[8] It models typical operations for any tactic: recursion, steps, success and failure. Therefore, we will generically refer to the simplifier as a "tactic".

A common theme across the interface is that some functions return `contexts`: it is crucial for the tactic to continue with that possibly updated `context`, or else it becomes impossible to track the actual hierarchical course of events and configuration changes.

Responses

Responses are usually supplied by the user, but can also be generated by the system under certain circumstances. The tactic is generally expected to obey the response in order to not confuse the user with further unexpected messages, e. g. when a "skipped" process is continued.

```
datatype step_response = Continue of Proof.context | Skip
```

Indicates whether the tactic should proceed with a given piece of work. The default response is `Continue` with unchanged `context`.

```
datatype fail_response = Exit | Redo
```

Indicates whether the tactic should backtrack to a state before attempting a failed step. It needs to be made sure that immediately after rolling back, the `step` function is called with the same parameters again. Usually, that requires the tactic to save a reference to the `context` used earlier. If the response is `Exit`, the tactic should abandon the step and clear any references to the `context` produced by the earlier call to `step`.

[8] Granted, the strings used to build up parts of the message contents are not configurable at the moment.

Operations without User Interaction

```
val recurse: string -> term -> Proof.context -> Proof.context
```

This function should be called whenever a new recursive invocation of a tactic is started – in particular the first invocation – and has no observable side-effects. The tactic should then continue with the returned `context`, otherwise further messages do not get properly associated with that specific invocation. `recurse` updates the `context` with an increased `depth` counter. Should the `depth` counter exceed `max_depth`, tracing will be disabled.

```
val indicate_success: thm -> Proof.context -> unit
```

This function can be called when an (attempted) piece of work has been finished. The `thm` argument should be the proved theorem. The `context` argument must be a `context` obtained from the preceding call of `step`. After the call, the tactic should not do anything else with that piece of work, instead continuing with the next piece. If the tactic has no notion of "failed" or "successful" steps, it is not necessary to call this function.

Operations with (Potential) User Interaction

These operations have in common that their return value is wrapped inside the `future` type constructor. If tracing is disabled, they return values which contain a reasonable default, i. e. no waiting or blocking occurs.

```
type trace_data = {term: term, conditional: bool, ctxt: Proof.context,
                   thm: thm, name: string}
val step: trace_data -> step_response future
```

This function should be called when the tactic attempts to solve a piece of work, specified by `term`. The `conditional` parameter indicates whether the step will likely induce a recursive call of the tactic.[9] `thm` and `name` are the theorem and its name which is used to solve the work. Both `term` and `name` are used to match the step against the set of defined breakpoints. Because some tactics mangle the internal name of theorems, the original name has to be specified explicitly.

[9] currently unused

```
val indicate_failure: trace_data -> Proof.context -> fail_response future
```

Communicates that the execution of a step failed, as opposed to the `indicate_success` function (same conventions apply). It is not necessary, but recommended to pass the same `trace_data` as to the earlier call of `step`. Note that the `context` parameter and the `context` field in the `trace_data` parameter are different from each other. (They denote the context after and before the step was attempted, respectively.)

Example Implementation

For illustration purposes, we will explain a stripped-down implementation of the `step` function.

```
fun step {term, ctxt, ...} =
  let
    val body_term =
      Syntax.pretty_term ctxt term
```

This generates the content of the message. Because the IDE is unable to pretty-print messages by itself, that needs to be done directly in the ML layer.

```
    fun payload () =
      {props = flags,
       body = (* ... *)}
```

Apart from `body_term`, there are many other components which constitute a message. Together, they are referred to as *payload* and wrapped in a thunk to avoid unnecessary pretty-printing (which can slow down the process significantly when the terms grow in size).

```
    fun mk_promise result =
      let
        val mode = (* extract current mode from ctxt *)
        fun to_response str = case str of
          "skip"           => Skip
        | "continue"       => Continue (put mode true)
        | "continue_trace" => Continue (put Full true)
        (* ... *)
      in
```

```
      (* send request, then: *)
      if not interactive
        then Future.value (Continue (put mode false))
        else Future.map to_response promise
    end
```

This is the heart and soul of the procedure. It sends the data to the IDE, and depending on whether the user enabled interactive mode, it immediately returns a value or generates a `promise`. The `to_response` helper function parses the plaintext reply from the user. Here, `put` is a function which updates the current `context` with new configuration.

```
  in mk_promise (mk_generic_result payload ctxt) end
```

In the end, the payload is packaged into a message using a suitable representation. There is another shortcut (omitted for brevity) here which avoids sending a request under even more circumstances.[10]

4.3.3 Message Filtering

Under certain circumstances, messages are being filtered and not presented to the user. There are a couple of ways messages might get rejected, with the most prominent example being breakpoints when normal verbosity is selected (refer to section 3.6 for an overview). Since it is generally desirable to avoid generating messages which will be rejected later, but rejecting happens in multiple stages, the code which avoids generating them is spread across the implementation. Here, we will explore the term breakpointing mechanism more closely (theorem breakpointing works similarly, but does not require sophisticated data structures).

Term breakpoints are kept in *discrimination nets* [5, §11]. Their Isabelle implementation lives in the `Net` module. Basically, nets are a map-like structure with terms (with free variables) used for indexing data. The most fundamental operations are:

```
val empty: 'a net
val insert_term: ('a * 'a -> bool) -> term * 'a -> 'a net -> 'a net
val match_term: 'a net -> term -> 'a list
```

[10] Basically, `mk_generic_result` returns an `option`. The actual code performs a pattern match, where the `NONE` represents ineligibility of the message (not triggered by a breakpoint, depth exceeded, …), and continues by `put`ting and returning an unchanged context.

The function passed to `insert_term` denotes equality of the indexed values. This is needed because nets do not associate terms with a multiset of values, but rather only a set. To quote the implementation, `match_term` "returns items whose key could match [the given term]".[11]

Given these operations, it becomes almost trivial to obtain the set of matching breakpoints given a term:

```
fun find_term_breakpoints thy term net =
  let fun matches pattern = Pattern.matches thy (pattern, term)
  in List.filter matches (Net.match_term net term) end
```

Observe that it is necessary to filter the results, because nets are allowed to return an overapproximation.

This function is called for each `step` message. Should it return a non-empty set, the message is *triggered*.

4.3.4 Message Passing from the ML to the JVM Layer

All these operations attach result data to the corresponding command. Emitting a result is easy: it merely involves calling the `Output.result` function, which takes a `serial` number and some content.

For a non-interactive message, that is all, because no response is expected. On the other hand, when an interactive message is sent, the caller expects a (future) response. This is realized in the `send_request` function (listing 6).

First, it creates a new and empty `promise`, which will be added to a global table of active promises. The key for the new entry is the same number as the identifier for the result. Because that table contains mutable state shared across (potentially) different threads, it is wrapped into a synchronized `var` [22, §0.7.3] which provides atomic access via the `Synchronized.change` function. Secondly, the given content is added to the document model. Finally, the new `promise` is returned to the caller.

Producing a result generates a `Commands_Changed` event in the IDE. The handling of this and other events is described in section 4.4.

[11] File `~~/src/Pure/net.ML` by Lawrence C Paulson

```
1  val futures =
2    Synchronized.var "Supertrace.futures" (Inttab.empty: string future Inttab.table)
3
4  fun send_request (result_id, content) =
5    let
6      fun break () =
7        (Output.protocol_message
8           [(Markup.functionN, cancelN),
9            (futureN, Markup.print_int result_id)]
10          "";
11         Synchronized.change futures (Inttab.delete_safe result_id))
12     val promise = Future.promise break : string future
13   in
14     Synchronized.change futures (Inttab.update_new (result_id, promise));
15     Output.result (result_id, content);
16     promise
17   end
```

Listing 6: Sending a request to the IDE

A slight complication arises when creating a `promise`, though. Because a running command could become stale at some point during execution (for example, when the user changed the source code above the invocation, changing the parameters), the system needs to know how to *cancel* a `future`. For regular `futures`, that is trivial, because it only requires cancelling the running computation on the corresponding worker thread. However, for `promises`, a cancellation behaviour has to be specified explicitly (function `break` in listing 6). In the context of the tracing, it means telling the IDE that interactive messages which do not have a response yet can be deleted.

The naïve approach would be to add a dedicated message to the document using the result mechanism, too. This fails, because as soon as a command becomes stale, it will be removed from the document model. For the IDE, that means that all results disappear and it will thus be unable to observe the cancellation message.

Hence, an out-of-band mechanism is needed. In that case, it is done via a low-level protocol message. Those are completely independent from the document model and can thus still be parsed by the IDE.

4.3.5 Message Passing from the JVM to the ML Layer

The reverse direction is completely done via protocol messages. There is just one message type here: `reply`. It carries just the `serial` used to create the promise, and a `string` denoting an automatically generated answer or the answer from the user.

Fortunately, setting this up is rather simple (reproduced here without error handling):

```
fun react xs = case xs of
  [s, r] =>
    let
      val serial = Markup.parse_int s
```

The `serial` and plaintext reply is obtained from the message.

```
      fun lookup_delete tab =
        (Inttab.lookup tab serial, Inttab.delete_safe serial tab)
```

In turn, the `serial` is used to retrieve and remove the pending future from the global table.

```
      fun apply_result promise =
        Future.fulfill (the promise) r
    in Synchronized.change_result futures lookup_delete |> apply_result end
```

Then, that `promise` is fulfilled with the (unparsed) reply of the IDE.

```
val _ =
  Isabelle_Process.protocol_command "Document.supertrace_reply" react;
```

Finally, the procedure has to be registered with Isabelle.

4.4 IDE Interaction

The code responsible for the IDE is divided into a general part which does not depend on jEdit or GUI features, and one that does. The former mirrors the data model of trace messages (section 3.3), tracks the current interactive state and communicates with the prover. The latter consumes that state and converts the abstract data into user-facing text and control elements.

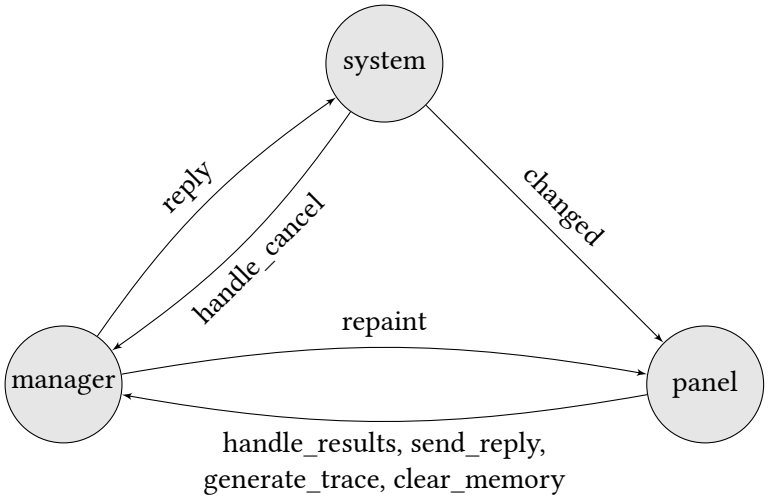


Figure 4: Actor communication

To facilitate the asynchronous nature of the interaction between prover and user, the Isabelle/Scala layer heavily relies on *actors* [11, 12, 20, 23]. Actors are a concurrency abstraction and can be seen as individual lightweight threads which are activated when a message is sent. Usually, actors have some hidden internal state. This greatly reduces the amount of coupling and helps avoid shared mutable state. Actors can reply to the sender of a message, but are also allowed to initiate communication with other actors.

Without going into too much detail of the internals of the IDE implementation, the basic infrastructure consists of a session actor which sends and receives protocol messages and actors for each part of the window (e. g. some of the dockable panels in jEdit have their own actors) to which processed messages are dispatched to. In this work, that infrastructure is extended by a Supertrace *manager* actor which serves as the "broker" between the system and the GUI, and a *panel* actor which coordinates the Supertrace panel in jEdit.

That particular actor is bound to the session actor, but also interacts with a newly introduced dockable Supertrace panel. The communication is depicted in figure 4.

Unfortunately, there is a naming clash in the following sections. We have been referring to the pieces of trace information sent to the IDE as "messages", but here, that term is used for the more low-level data which is exchanged between actors. However, since the transmission of trace information is internally realized via `Output.result`, we will call them *results*.

4.4.1 Manager Actor

The manager actor both receives and sends messages. Probably coming from the Erlang tradition of actors,[12] the interface of an actor is usually untyped, that is, anybody with a reference to an actor can send arbitrary messages, but get no guarantees whether a message will bring about a reply or what type that reply has.

Traditionally, an actor only exposes a `tell` function (abbreviated with `!`) to send a message. The Scala actor library additionally offers a `!?` function, which waits for the actor to finish processing the message and returns its reply. These functions take and return `Any`, which in practice turns out to be a huge burden for clients of the interface, especially in a strongly-typed language like Scala.

Hence, a decision has been made to hide the actor-based implementation of the manager and provide just a strongly-typed interface.[13] That greatly reduces time spent on debugging, because if programmed carelessly, untyped actors which receive an unknown message might become unresponsive for the rest of their lifecycle.

Incoming Messages

As already indicated, different actors communicate with the manager actor. No access control is needed: The side-effecting methods are idempotent, i. e. they return the same result when invoked with the same arguments repeatedly. In the following, the public API of the manager actor is described.

```
type Question = (Data, List[String], String)
type Context = Map[Long, Question]
type Results = Map[Long, XML.Tree]
```

A *question* consists of parsed result data (representing the trace information), a list of possible answers, and a default answer. A *context* (not to be confused with proof or

[12] The author of the current actor implementation in Scala, Philipp Haller, draws parallels to Erlang and writes that "in that model, it is common to have nested receives [...] Now, if [their types] are unrelated, then certainly, the actor must be able to accept messages of type `Any`". (`Any`, quite similarly to Java's `Object`, models the top of the inheritance hierarchy. It is the supertype of all Scala types.) The archived post can be found at `http://article.gmane.org/gmane.comp.lang.scala/12988`.

[13] As a side note, the current (2.10) implementation of Scala actors are deprecated in favour of the *Akka* library [14]. That library provides a notion of *typed channels*, which encode the message type in the type of an actor [4, §4.4]. This constitutes a great opportunity for an overall design improvement of Isabelle/Scala by leveraging compile-time type checking.

theory contexts) contains all open questions for a certain scope. The type `Results` is merely associating raw results with their serials.

```
def handle_results(id: Document.Command_ID, results: Results): Context
```

This can be called by any entity which is interested in displaying the currently unanswered questions (i. e. interactive messages from the simplifier which have not been replied to yet). The `id` parameter represents the current position in the document, and `results` holds all results for this command. This call side-effects by updating internal state and might reply to some results, which in turn do not appear as questions in the returned context. The method does not process all results, but only new results since the last invocation. Since results are added incrementally (new results are guaranteed to have a higher serial number than the old ones), this is safe.

The processing deals with the different result types as follows:

step Looks up the result in the memory. If it has already been answered, it will answer exactly the same way as earlier. Otherwise, it will be converted to a question and added to the context.

hint Success hints can be ignored, because they are not interactive. On the other hand, failure hints are interactive.

For those, it is checked whether the step they are referring to is actually known. If yes, a question is added to the context. If not, the default reply is sent.

(The latter happens e. g. when a step failed which has not been triggered by a breakpoint; such an event is still logged by the system, but not interesting to the user.)

ignore An `ignore` result is emitted by the `Supertrace` module if and only if a failed step is about to be repeated and marks the subtree of the failed step as stale.

This involves a partial clearing of the memory, based on the *provenance* of a memoized question. All answers to question which originated from the stale subtree will be purged.

Other result types are inherently non-interactive and can be ignored.

In the current implementation, this function is only called by the panel actor with the results of the active command. Conceptually, this has the advantage that only the results which are actually needed by the user are being processed, thus avoiding spending time producing superfluous data.

However, this leads to a somewhat odd effect: If a question can be answered from memory, this only happens when this method is called, i. e. when the user clicks on the corresponding command and thus moves the focus in the editor.

The alternative would be to let the session actor invoke this method as soon as a `Commands_Changed` message is sent, shifting the above trade-off away from unexpected behaviour at the cost of a perhaps negligible performance drop. It is not possible to implement at the moment, though: The information about which commands have updated results is not available for that actor. Obtaining that information requires a `Document_View` which conceptually lives in the Isabelle/jEdit layer which is on top of the Isabelle/Pure layer.

```
type Cancel = Long
def handle_cancel(cancel: Cancel): Unit
```

This function takes a serial, and removes it from the internal state if necessary. It is invoked by the session actor, because it receives that message directly via the system protocol.

Since cancellation indicates a stale command, a `Commands_Changed` message for the new command will be issued. Hence, the management actor does not need to issue a repainting message to the panel.

```
type Trace = List[Data]
def generate_trace(results: Results): Trace
```

Called by the panel actor when the user requests a full view of all trace information up to that point. This function does not update internal state in any way and is pure.

The interesting behaviour is that at this point, the hierarchical structure of the trace has to be reconstructed. In our data model, each message also contains the serial of its parent message to allow for precisely that. On the ML side, observe that the functions for each message type which may have children (`recurse`, `step`) return an updated `context`: in there, the `parent` is set to the serial of the message. Then, when attaching a result to the command, both serial and `parent` are included.

The Scala side just sees a list of results, sorted by serial. Tree reconstruction works on a tree with a focussed node, initially set to root element without children. It then performs these steps for each result:

1. Extract the `parent` from the result and find the node with that serial.
2. Add the result as a child to that node.
3. Set the focus to the newly generated child.

The first step does not need to search the whole tree, but just the parents of the focussed node. This works because the list of results represents an in-order traversal of the original tree.

```
def clear_memory(): Unit
```

This is called by the panel actor and completely purges all memoized questions from the memory, including provenance.

```
type Reply = (Long, Answer)
def send_reply(reply: Reply): Unit
```

In order, a call performs the following steps:

1. looks up the corresponding question from the set of unanswered questions
2. if the question is related to a `step` message record the reply and its provenance in the memory (replies to `hint` messages are not memoized)
3. remove the question from the set of unanswered questions
4. reply to the prover by sending actual `reply` message to the session actor
5. send `repaint` message to the panel actor

Conceptually speaking, this method is only called by the panel actor. For details, see section 4.4.2.

Outgoing Messages

There are two outgoing message types:

```
sealed trait Event
```

An *event* merely indicates that the GUI should repaint itself because its contents have been changed. Emitted during a `send_reply` call.

```
type Reply = (Long, Answer)
```

Issued to the session actor during a call to `handle_results` (reply found in the memory) or to `send_reply` (reply determined by the user).

4.4.2 Panel Actor

Every Supertrace dockable panel has its own actor. Note that there might be multiple instances open at the same time. By default, that is not very useful because all of them share the same cursor, but when `Auto update` is disabled, their focus will not be changed when a different command is selected in the editor area. This is very useful when tracking multiple simplifier states in different places.

The panel actor performs no communication apart from what is already described in the previous section and what the other panel types (e. g. the `Output` panel) do.

However, the panel communicates with an out-of-system actor: the user. Every time a command has changed, the panel actor obtains the current context from the management actor and renders its questions. The rendering target is a `Pretty_Text_Area`, a built-in component which supports standard Isabelle syntactic and semantic markup. However, it is extensible: It also allows for custom *markup*, thus attaching arbitrary information to the output. These "special" ranges in the output text are highlighted differently and thus signal "clickability" to the user. Despite that the implementation uses standard Java/Swing technology, clicking on such an active range does not produce a regular Swing event, but triggers the `action` method in the `Active` object. In order to enable this method to react properly to the clicking event, the panel actor attaches the serial and the plaintext string to each possible answer:

```
def make_button(answer: Supertrace.Answer): Tree =
  XML.Wrapped_Elem(
    Markup(SUPERTRACE_BUTTON, Future(data.serial)),
    answer.string,
    answer.label
  )
```

The `action` method in turns extracts this information:

```
def action(view: View, text: String, elem: XML.Elem)
{
  // ...
  elem match {
    // ...
    case Supertrace.Button(reply) =>
      PIDE.session.supertrace_manager.send_reply(reply)
  }
}
```

4.5 Summary

This concludes the full interaction between the system, the IDE and the user. Recall the sequence diagram in figure 2. Let us briefly summarize the course of actions:

- During a run of the simplifier (or any other tactic, for that matter), tracing information accumulates, some of which necessitates user interaction.
- The tactic then calls an asynchronous API which manages pre-filtering of "interesting" messages, converts them to a suitable representation, sends those over to the IDE, and provides the tactic with a read handle on the reply. The only tactic supporting Supertrace as of now, the simplifier, uses that API synchronously though.
- The message is received and parsed by the IDE and more filtering is done. Accepted messages are turned into questions and presented to the user, along with a number of possible answers.
- The user eventually clicks on an answer. This triggers a cascade of events, most notably sending the reply back to the prover.
- The prover receives and parses the reply and hands it back to the tactic, which can now continue working. The further activity of the tactic may or may not be influenced by the answer the user gave, including backtracking or early exiting.

All in all, this constitutes a full round-trip.

5 Case study: A Parallelized Simplifier

For testing purposes, an extremely stripped-down version of the simplifier has been implemented to demonstrate the capabilities of the tracing infrastructure. It uses exactly the same interface as the "real" simplifier, but is not nearly as powerful (it consists only of 100 lines of code). However, it splits off some of the work into parallel tasks. In the simple test cases, that does not translate to a speed-up of the simplification, but rather interesting insights into the concurrency features of this work.

An example of that interaction can be seen in figure 5. Clicking on an answer makes the question disappear as usual, but the others remain there.

Parallelism is introduced in two distinct places: when solving multiple preconditions, and when trying multiple matching rules.

For illustration purposes, we will briefly discuss the first of them.

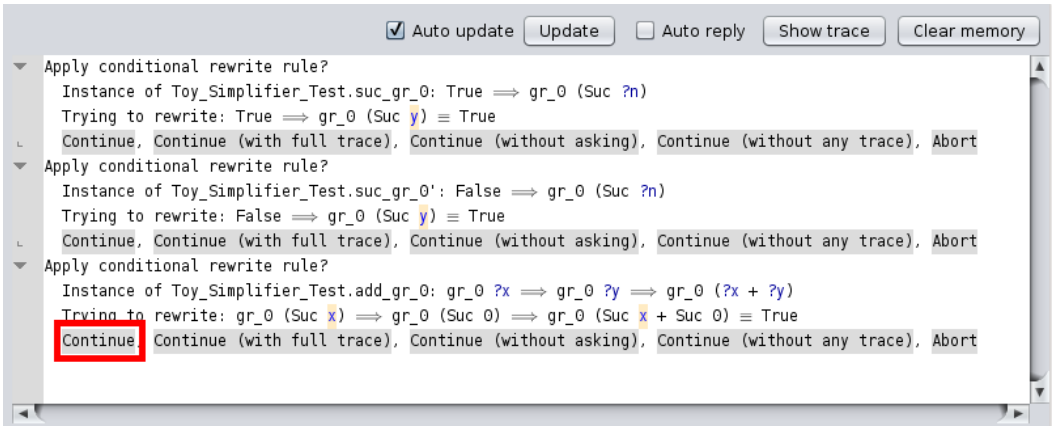
```
fun try_solve_prems thm conv =
```

The function `try_solve_prems` attempts to solve all premises of `thm` using the *conversion* `conv` (the details of which are unimportant).

```
  let
    fun solve_future ct = Future.fork (fn () => try_solve_prem ct conv)
    val futures = map solve_future (cprems_of thm)
```

Here, the parallel computations are initiated using `Future.fork`. `try_solve_prem` itself is a helper function which massages the term before it is being passed into the conversion, and checks the result afterwards. The type of `futures` is `thm option future list`: a list of results (represented by an optional theorem) which are wrapped into a `future` each.

```
    fun aux (future, thm) = case Future.join future of
      NONE => raise Fail "Could not solve subgoal"
    | SOME prem => prem COMP thm
  in List.foldl aux thm futures end
```



(a) … before clicking on the answer marked red

(b) … and after

Figure 5: Multiple questions presented at the same time

The last step consists of collecting all the values. The end result of that function is a theorem. Note that this function is blocking, that is, it does not return immediately, but rather waits for all subgoals to finish.[14]

Apart from that, the above code snippet nicely demonstrates that adding parallelism to the simplifier is not very hard. However, care needs to be taken about the user interface and interaction. Important issues are:

- At most how many questions should be presented to the user at any given time? The goal should be to not overburden the user. Should the number of questions be limited, how are those selected? Alternatives are "most recent", "least recent", or some grouping depending on the proximity in the tree.
- Should similar or identical questions be folded together? The answer here is probably "yes", but once again, careful tuning is needed: the user might expect to see two instances of the same question if they originated at different points in the trace.
- How can the context of an active question be depicted? In a sequential setup, this is not really important, because the user can always open the trace window to recall the overall state. If the simplifier runs in parallel though, there are multiple nodes in that tree which are currently pending.

There are no clear answers to these yet. The course of action should be to collect more data about the actual usage patterns of the simplifier trace first. Also, several extension of the concurrency library in Isabelle are necessary to constitute a fully asynchronous system.

[14] With the current `future` implementation in Isabelle, a workaround for that is conceptually impossible without blocking a thread internally. Observe that the implementation uses `Future.join` to turn a `'a future list` to a `'a list future`. In Haskell, such a function is called `sequence` [24, p. 47] and requires the `future` implementation to offer a function with type `'a future -> ('a -> 'b future) -> 'b future`. Unfortunately, such a function is currently not implemented.

6 Evaluation and Future Work

In this section, we will briefly discuss the performance of the Supertrace extension and the practical usability. Furthermore, possibilities for future work are explored.

6.1 Performance

Logging the simplification process obviously incurs a measurable overhead. For example, consider the expression $10^x \cdot 10^y$. The test machine is an Intel Core i7-2600 with a peak frequency of about 3.5 GHz. Execution times have been collected using the `Timing` panel in Isabelle/jEdit without having the IDE render the trace data. The results can be seen in table 2.

As can be seen in the table, the simplifier itself is pretty fast, but enabling the trace slows down the process significantly. Note that for measuring in `normal` mode, no breakpoints have been set, hence these numbers show just the overhead of the Supertrace module. The most interesting comparison is between `legacy` and `full` mode, where the ratio is roughly $1 : 2$. This can be explained by the fact that the full mode collects more information and processes them more thoroughly than the legacy mode.

The slowest component overall is the GUI though, which requires about 7 s to display the full trace for $x = y = 10$. This is at least double the time the legacy trace needs. However – once rendered – scrolling, collapsing and expanding nodes is instantaneous.

An important fact to keep in mind is that GUI actions which take longer than half a second are generally perceived to be "laggy", i. e. they disrupt the usual workflow. Hence, it is generally advisable to enable the simplifier trace only if necessary.

6.2 Future Work

There are multiple dimensions in which this work can be extended in the future.

Integration into the Isabelle system would benefit by adapting more tactics to use the new tracing mechanisms, since many of them can be modelled in a hierarchical manner, and more message types could be introduced.

The user experience could be improved by asking even less questions (introduce fuzzy matching in the memoization) or providing more information. Interesting context data

x	y	disabled	legacy	normal	full
10	10	0.0	1.0	0.8	1.9
20	10	0.1	3.1	2.5	6.3
20	20	0.2	6.3	5.3	12.1

Table 2: Execution times by verbosity mode

includes for example term provenance, i. e. tracking how a particular subterm in the result emerged during the rewriting process.

From an implementation perspective, tighter integration with the Isabelle/jEdit would be desirable, that is, to avoid the delay effects described in section 4.4. However, that would most likely involve significant changes in the Isabelle/Scala layer.

As already laid out in section 5, there is much work which can be done in parallelizing the simplifier. Furthermore, the current implementation would hang indefinitely if the simplifier fails to terminate. A facility which could perform some basic termination checks, along with a new message type informing the user of the problem, would thus be of great benefit.

Support for the other Isabelle IDE, Proof General [3], is currently not planned. Albeit the ML code written for this thesis could largely stay the same, a complete reimplementation of the manager actor (amongst others) in Emacs Lisp would be required.

6.3 Conclusion

We have developed a generic tracing facility for Isabelle tactics which strives to replace the old simplifier trace. That new facility is interactive, highly configurable and reasonably intuitive to operate. The impact on the rest of the system turned out to be rather small. Nonetheless, it became possible to provide more insights for the user into the simplification process.

The design goal that the amount of interaction with the user should be kept low has been achieved. Various sophisticated filtering and memoization techniques help maintaining a good trade-off between flexibility and opacity of the system. The user does not have to learn how these work, though, because in most cases they behave just as expected.

A User manual

In this section, we will explain how to set up the Isabelle/jEdit IDE to display the Supertrace panel. Then, the configuration attributes are introduced. Finally, the remaining section describes the interaction with the system.

A.1 Fundamentals

The interaction happens inside of a panel which can be activated as depicted in figure 6. The panel consists of two sets of controls and a message text area. The upper set of controls (figure 7, (1)–(4)) is used to influence the handling of the messages by the IDE. The text area shows zero or more active questions, i. e. interactive messages from the simplifier. At the moment, there will be at most one question displayed simultaneously, and the simplifier cannot advance without user intervention if there is an active question.

A.2 Configuration

There are multiple configuration axes: *interactivity, verbosity,* and maximal *depth.* Except for the last dimension, those settings can be changed during a simplification run. Furthermore, the user can set *breakpoints* for terms and theorems.

By default, the trace is disabled and the panel will not show any contents. To enable, one has to declare the `supertrace` attribute.

The full syntax to enable the trace is as follows:

```
declare [[supertrace mode=normal interactive depth=10]]
```

Usually, one would want to just specify `interactive` and skip the other fields.

If interactivity is disabled (which is the default), the progress of the simplifier can be observed by opening the trace window which shows the steps the simplifier made (figure 8 and section A.3). Interactivity can be switched on by specifying the `interactive` attribute. Questions then become visible as can be seen in figure 7.

(Verbosity) Modes

For interoperability reasons the tracer offers a legacy mode. In legacy mode, the output is almost identical to the "old" simplifier trace. It can be enabled by declaring

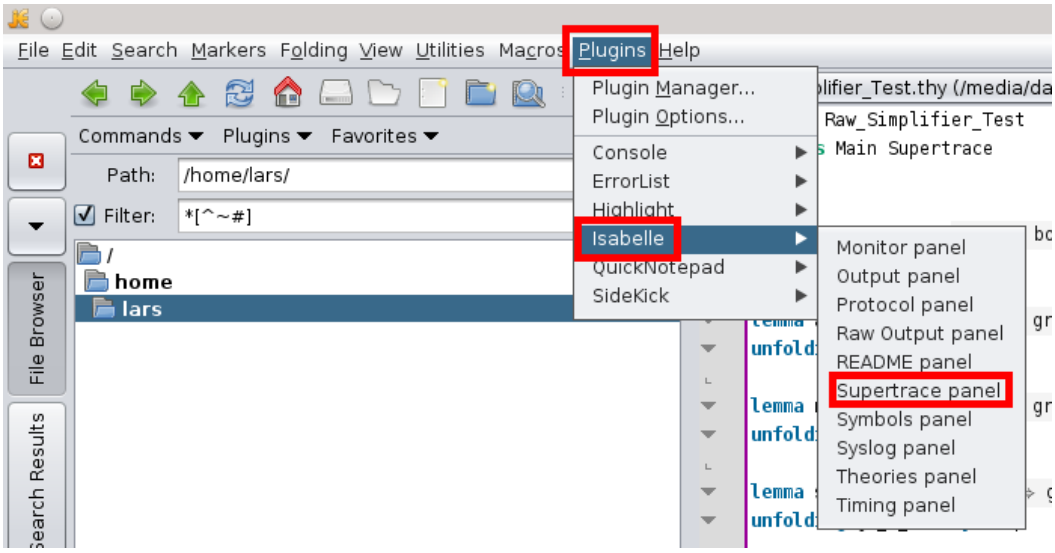
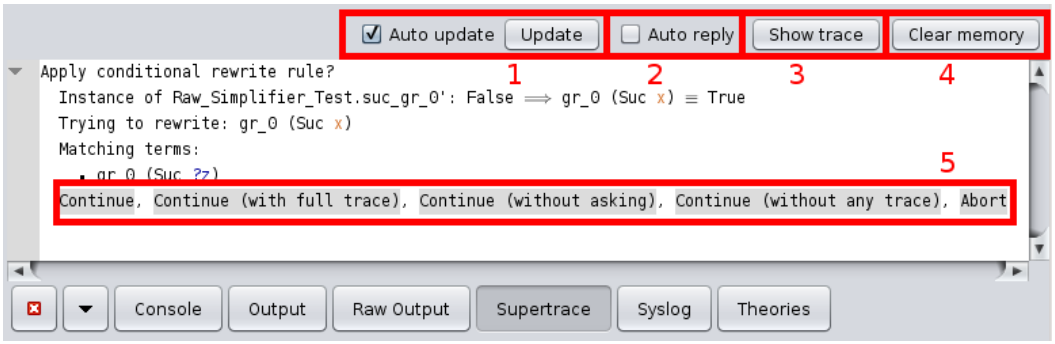


Figure 6: Menu: Open SUPERTRACE panel



1. if enabled, panel contents follow cursor position
2. if enabled, automatically reply with a default response
3. show accumulated trace information in new window
4. clears the response cache
5. list of possible responses for the active question

Figure 7: SUPERTRACE panel

```
declare [[supertrace legacy]]
```

which requires that the `interactive` flag is not set.

Apart from that, the three "regular" modes of operation are:

disabled does not produce any trace messages at all

normal produces messages, but displays them only if their contents are triggering a breakpoint

full produces messages and displays all of them

In most cases, it is reasonable to stay away from the *full* mode: even for seemingly small terms, the potential amount of applied rewrite rules can get quite high. While the system has no problem producing and transmitting these messages, displaying them might take a while.

Depth

This setting works as expected from the legacy trace: it merely specifies the maximum depth until which trace messages are being produced. The default is 10.[15]

It can occasionally be useful to increase the default to a higher number, and in *normal* mode it rarely makes a difference, since most of the messages get filtered out anyway in this mode. In *full* mode, it is strongly advised against increasing the limit.

Breakpoints

Breakpoints can be set with

```
declare [[break_term "?x > 0"]]
declare conjI[break_thm]
```

Because term breakpoints might want to capture locally fixed variables, it is possible to declare them "on the fly":

[15] As indicated earlier, the maximum depth setting cannot be changed while a simplification run is in progress. The problem is – once again – designing an appropriate user interface. The GUI component used to display the questions supports only showing text, some of which might be clickable, but not entering text.

Reply	Verbosity	Interactivity
(default)	set to *normal*	left unchanged
`with full trace`	set to *full*	left unchanged
`without asking`	left unchanged	disabled
`without any trace`	set to *disabled*	disabled

Table 3: Settings change

```
proof
  fix x
  have "..."
    using [[break_term "?z > x"]] ...
  ...
qed
```

A.3 User Interaction

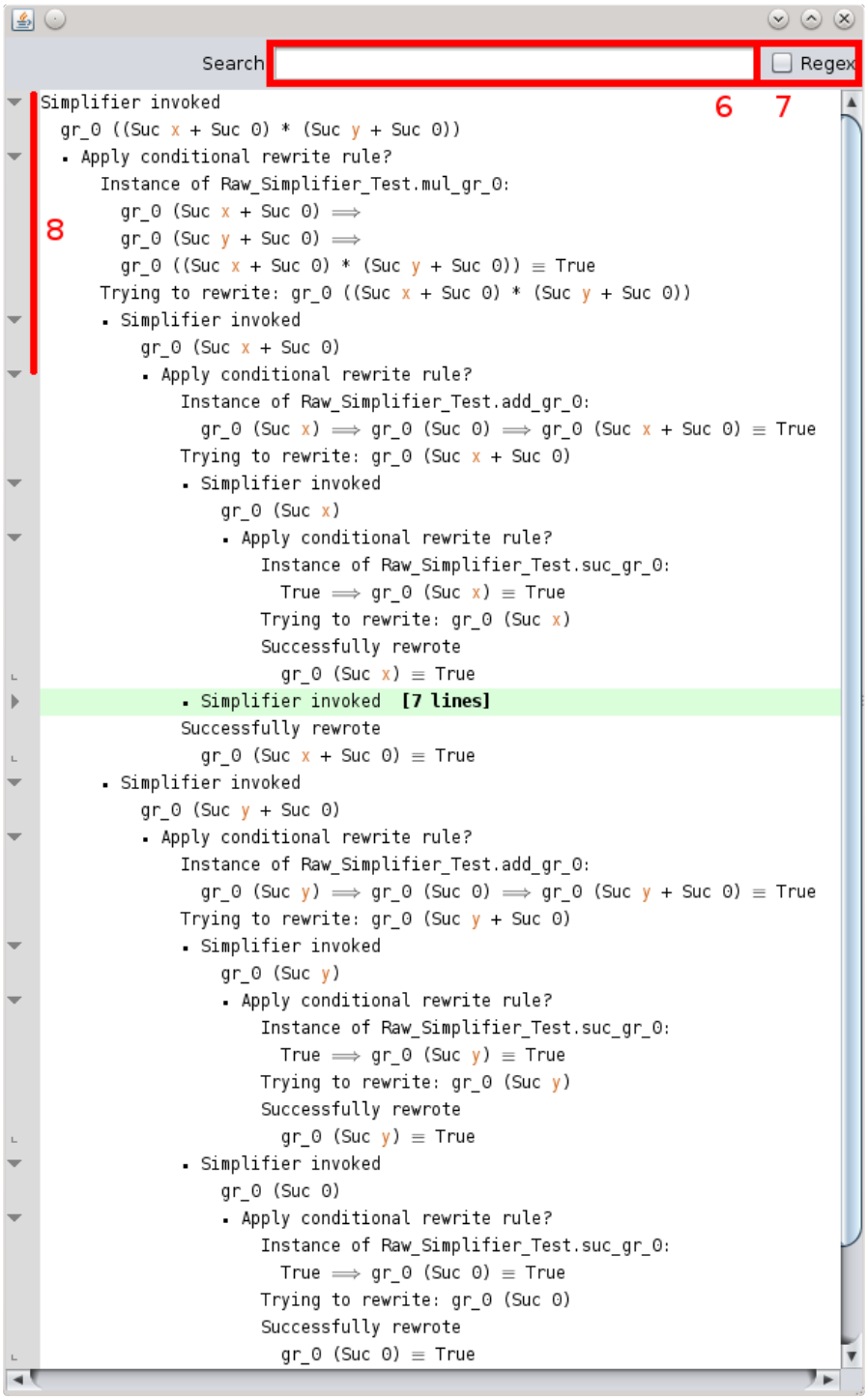
Questions behave just like regular output: they are attached to a command. That means that in order to see and react to a message, the cursor must be placed on the originating command. (Usually, that is an invocation of a tactic involving the simplifier, such as `simp` or `auto`.) Fortunately, the IDE highlights active commands.

Every time a command is selected, its active questions are displayed immediately in the text area of the SUPERTRACE panel (figure 7). Similarly to the output panel, this behaviour can be overridden: if the checkbox `Auto update` (1) is deselected, the panel will keep its focus, even if the cursor moves to another command. Explicit updating can be done by clicking on the `Update` button.

The text area offers the usual markup as provided in other places in the IDE. For example, clicking on a symbol in a term opens the corresponding definition in the editing area.

A.3.1 Answering Questions

This is a trivial undertaking: questions are shown in the text area of the panel (figure 7), together with the possible answers (5). An example for each question type is shown in listing 7.



(6) search box
(7) if enabled, search string is interpreted as a regular expression
(8) collapse/expand trace items

Figure 8: SUPERTRACE window

```
Apply conditional rewrite rule?
  Instance of thm:
    pre1 ⟹ pre2 ⟹ lhs ≡ rhs
  Trying to rewrite: f lhs

  Continue,
    Continue (with full trace),
    Continue (without asking),
    Continue (without any trace),
    Skip
```

(a) Applying a (conditional) rule

```
Step failed
  In an instance of thm:
    pre1 ⟹ pre2 ⟹ lhs ≡ rhs
  Was trying to rewrite: f lhs

  Redo,
    Exit
```

(b) Rewrite step failed

Listing 7: Question types

The user can freely choose and click on an answer, upon which the question disappears. Automatic choosing of the default answer can be enabled by checking `Auto reply` (2).

Note that for the question about applying a rewrite rule, the settings with which the tracing continues can be influenced. For example, if the user wishes to get more detailed information in the recursive call, they can click on the option `Continue (with full trace)`. The settings changes for each answer are listed in table 3.

A.3.2 Trace Window

At any time, the user can choose to open a window with the accumulated trace information (figure 8) by clicking (3) (in figure 7). In particular, that can also happen before the simplifier has terminated.

The window offers a hierarchical view on the process. Hence, the nodes can be collapsed and expanded ((8), figure 8).

Note that the window contents are not updated, neither automatically nor manually. This allows for convenient side-by-side comparison of old and new traces when the set of rewrite rules has been changed. It also enables displaying a (finite) snapshot of a non-terminating simplifier run, where otherwise the system would get stalled indefinitely while trying to print all trace information.

However, it is possible to filter the contents by typing any string into (6) and pressing the Enter key. Only the items where that string occurs in the body (not in the title) and their parents will remain. The search supports (Java) regular expressions. Finally, it is

possible to search inside of terms, because all formatting is stripped beforehand. Input for special symbols is not available yet.

Acknowledgements

I would like to thank my advisor, Lars Noschinski, for fruitful discussions about how the tracing should behave and helpful remarks about the structure of my writing. Furthermore, Makarius Wenzel, who patiently explained the inner workings of the various Isabelle layers to me, as well as Stefan Berghofer and Holger Gast who provided insights into the implementation of the simplifier. Finally, my supervisor, Tobias Nipkow, for supporting me and this topic.

I am very grateful to Cornelius Diekmann and Lukas Erlacher for their great review and comments about this thesis.

References

[1] Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518, 2009. URL `http://heim.ifi.uio.no/msteffen/download/07/futures.pdf`.

[2] María Alpuente, Demis Ballis, Francisco Frechina, and Julia Sapiña. Slicing-Based Trace Analysis of Rewriting Logic Specifications with iJulienne. In *ESOP*, pages 121–124, 2013.

[3] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43. Springer, 2000. URL `http://homepages.inf.ed.ac.uk/da/papers/pgoutline/pgoutline.pdf`.

[4] Eugene Burmako. Scala Macros: Let Our Powers Combine! In *Proceedings of the 4th Annual Scala Workshop*, 2013. URL `http://infoscience.epfl.ch/record/186844/files/2013-04-22-LetOurPowersCombine.pdf`.

[5] Eugene Charniak, Christopher K Riesbeck, Drew V McDermott, and James R Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1987.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.6).

[7] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. URL `http://maude.cs.uiuc.edu/papers/postscript/tcs4006.ps.gz`.

[8] William F Clocksin and Christopher S Mellish. *Programming in Prolog: Using the ISO standard*. Springer, 2003.

[9] Mireille Ducassé and Jacques Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19–20, Supplement 1:351–384, 1994. ISSN 0743-1066. URL `http://www.sciencedirect.com/science/article/pii/0743106694900302`.

[10] Thom Fruehwirth, Jan Wielemaker, and Leslie De Koninck. *SWI Prolog Reference Manual 6.2.2*. Books on Demand, 2012. ISBN 9783848226177. URL `http://www.swi-prolog.org/pldoc/refman/`.

[11] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009. ISSN 0304-3975. URL http://www.sciencedirect.com/science/article/pii/S0304397508006695.

[12] Philipp Haller and Frank Sommers. *Actors in Scala*. Artima Incorporation, 2012.

[13] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises, 2012. URL http://docs.scala-lang.org/overviews/core/futures.html.

[14] Vojin Jovanovic and Philipp Haller. The Scala Actors Migration Guide, 2012. URL http://docs.scala-lang.org/overviews/core/actors-migration-guide.html.

[15] David CJ Matthews and Makarius Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 53–62. ACM, 2010. URL http://www4.in.tum.de/~wenzelm/papers/parallel-ml.pdf.

[16] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002. URL http://isabelle.in.tum.de/doc/tutorial.pdf.

[17] Lawrence C Paulson. *Isabelle: A Generic Theorem Prover*, volume 828. Springer, 1994.

[18] Leon Sterling and Ehud Y Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press Cambridge, 1994.

[19] Makarius Wenzel. Parallel proof checking in Isabelle/Isar. In *Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 13–29. ACM, 2009. URL http://www4.in.tum.de/~wenzelm/papers/parallel-isabelle.pdf.

[20] Makarius Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. *Electronic Notes in Theoretical Computer Science*, 285:101–114, 2012. URL http://www4.in.tum.de/~wenzelm/papers/async-isabelle-scala.pdf.

[21] Makarius Wenzel. Isabelle/jEdit – a Prover IDE within the PIDE framework. In *Intelligent Computer Mathematics*, pages 468–471. Springer, 2012. URL http://arxiv.org/pdf/1207.3441.

[22] Makarius Wenzel. The Isabelle/Isar Implementation, 2013. URL http://isabelle.in.tum.de/dist/Isabelle2013/doc/implementation.pdf.

[23] Makarius Wenzel and Burkhart Wolff. Isabelle/PIDE as Platform for Educational Tools. In *THedu'11*, pages 143–153, 2011. URL http://arxiv.org/pdf/1202.4835v1.

[24] Brent Yorgey. The Typeclassopedia. *The Monad.Reader*, 13:17–68, 2009.