Understanding and maintaining tactics graphically OR how we are learning that a diagram can be worth more than 10K LoC

Understanding and maintaining tactics graphically OR how we are learning that a diagram can be worth more than 10K LoC

Yuhui Lin, Gudmund Grov
Heriot-Watt University, UK
and
Rob Arthan
Lemma1, UK

The use of a functional language to implement proof strategies as *proof tactics* in interactive theorem provers, often provides short, concise and elegant implementations. Whilst being elegant, the use of higher order features and combinator languages often results in a very procedural view of a strategy, which may deviate significantly from the high-level ideas behind it. This can make a tactic hard to understand and hence difficult to to debug and maintain for experts and non-experts alike: one often has to tear apart complex combinations of lower level tactics manually in order to analyse a failure in the overall strategy.

In an industrial technology transfer project, we have been working on porting a very large and complex proof tactic into PSGraph, a graphical language for representing proof strategies, supported by the Tinker tool. The goal of this work is to improve understandability and maintainability of tactics. Motivated by some initial successes with this, we here extend PSGraph with additional features for development and debugging. Through the re-implementation and refactoring of several existing tactics, we demonstrates the advantages of PSGraph compared with a typical linear (term-based) tactic language with respect to debugging, readability and maintenance. In order to act as guidance for others, we give a fairly detailed comparison of the user experience with the two approaches. The paper is supported by a web page providing further details about the implementation as well as interactive illustrations of the examples.

1. INTRODUCTION

Proof tactics have played an important role in reducing user interaction and proof development time for interactive theorem provers. However, tactics tend to be difficult to debug and maintain: (1) they may not fail outright and instead generate undesirable subgoals; (2) each layer of a (reasonably) large and powerful (hierarchically composed) tactic may involve search which makes it hard to identify *why* it failed and *where* the culprit for the failure resides.

We will illustrate the maintenance issue with an industrial example: D-RisQ

This work has been predominantly supported by EPSRC platform grants EP/J001058 and EP/N014758 and IAA grant EP/K503915. The second author is supported by a SICSA industrial fellowship and the first author by EPSRC grant EP/M018407. The development of PSGraph was started in the AI4FM EPSRC grant (EP/H023852 and EP/H024204). We would also like to thank D-RisQ, in particular Colin O'Halloran and Priiya G, for excellent discussions. We would also like to thank the anonymous reviewers for the suggested improvements to the paper.

Software Systems[1] deploys a very powerful tactic to automate formal proofs of correctness of code auto-generated from Simulink models [47]. This tactic has been developed over a number of years and now constitutes around $10K$ lines of dense ML code ($50K$ LoC if scripts to prove supporting lemmas are included). Both a high degree of automation and *ease of maintenance* is crucial for D-RisQ's business model: when a conjecture fails a developer must have an efficient way of finding and fixing the problem. The tactic must be intuitive to use and understand so that, as personnel move on, new developers can take over maintenance and further development. Proofs of low-level properties of automatically generated code are not interesting in themselves: what is important is the ability to produce proofs automatically as evidence that the code generator has not introduced bugs. To support this, the tactic developers will want to exploit insights from one failed and patched proof to increase the level of automation on other conjectures.

Crucial to such debugging and refactoring of tactics is a suitable *tactic representation*. If one think of tactics as *flow networks*, where subgoals flow between tactics, then there is evidence that the human brain finds it more natural to understand such networks diagrammatically compared with linear (term-based) representations [39]. Our *PSGraph* [26] language was designed to make proof strategies more intuitive to understand and easier to debug and change than is the case with existing tactic languages. In PSGraph, the 'flow graph' view is followed literally and tactics are represented as directed, typed and hierarchical graphs. Boxes are labelled by (smaller) tactics and wire labels are used to direct subgoals as they 'flow' through the graph. This flow can be inspected step-by-step when debugging a graph. The PSGraph language is implemented in the *Tinker* tool [27, 44], which includes a graphical user interface to support the development and analysis of PSGraphs. The tool can support a range of theorem provers and has currently been instantiated for Isabelle [27], Rodin [40] and ProofPower [4]. In this paper, we concentrate on ProofPower, a system which is comparable with other provers in the HOL family such as HOL4 [59], HOL Light [32] and Isabelle/HOL [45].

Motivated by work with D-RisQ on their tactic in PSGraph [43], our main hypothesis of this research is that

> *understanding, debugging and maintaining*[2] *proof strategies is easier with PSGraph than with traditional linear tactic languages.*

The hypothesis relates to our overall research vision. We have already reported some evidence for it in an industrial setting [43]. This paper has an *exploratory* objective, where the goal is to study PSGraph's relative strengths and weaknesses with respect to our given hypothesis. The contributions of this paper are three-fold.

The first contribution, discussed in §3, comprises recent extensions to PSGraph and the Tinker tool, with new features to improve development and debugging. The most notable extension is the introduction of a new language for specifying wire labels.

[1] http://www.drisq.com

[2] Software *maintainability* refers to the ease in which a software system or component can be modified or adapted to a changed environment [1]. This will naturally include both *understanding* and *debugging* software. However, as the main loci is on these two aspects of maintainability, the hypothesis is explicit about them.

The second and, in our view, main contribution is found in §4, where we address our hypothesis by means of three case studies, each with a distinctive flavour and level of complexity. Evaluation based upon case-studies is motivated by the work being *exploratory* and *improvement-driven* [57]; the aim is to identify actionable limitations of PSGraph with respect to the hypothesis – and to provide the necessary armoury to address D-RisQ's tactic in full. We reflect on alternative evaluation approaches in §5.

To expose the differences between the user experience with the traditional linear representation of proofs and proof procedures and the user experience with our graphical representation, we work through several simple but instructive aspects of the case studies in some detail. Our aim is to give a good understanding of what goes on in the two approaches and to guide future work on and with PSGraph by ourselves and others. This leads us to the third contribution of this paper: to provide a tutorial-like introduction to PSGraph and how to go about connecting Tinker to a theorem prover. To support this, we therefore provide a fairly detailed background on ProofPower and PSGraph/Tinker in §2. Furthermore, §2.2.2 and §2.2.3 are fully devoted to prover integration, while parts of §2.1 and §3.1 discuss such integration. These parts can be skipped if desired. Some aspects of the case studies are very detailed for the same reasons. However, space does not permit a discussion of every detail and so particularly in the second case study, we have tried to give the flavour of the bigger picture supported by enough information to help interested readers find their way around the original source material.

After the description of each case study, we reflect and analyse our approach and provide recommendations which we hope can be used as a template for other developments. Crucially, while two of the authors (Lin and Grov) are developers of PSGraph, the third author (Arthan) had never used PSGraph before we started this work. Arthan was the developer of the original version of the case studies.

Our three case studies consider already extant proofs and proof procedures implemented in ProofPower: they comprise *(i)* a proof procedure for tautologies supplied as part of the standard proof infrastructure, *(ii)* some application-specific tactics used to finesse a tricky lemma forming part of the proof of security properties of a database system and *(iii)* a decision procedure taken from a collection of case studies on pure mathematics in ProofPower that automates problems such as proving the continuity of real-valued functions.

In §5 we discuss related work, and we conclude and discuss further work in §6. Additional supporting materials, including animations of the examples and detailed instructions are available from a dedicated webpage [42].

2. BACKGROUND

2.1 ProofPower

ProofPower [4] is a suite of tools supporting specification and proof. At its heart is an implementation in Standard ML of Mike Gordon's HOL logic (the same logic as is implemented in HOL4 [59] and HOL Light [32] and the core of the logic implemented in Isabelle/HOL [45]). It is implemented in the well-known LCF style [24, 25]. In this paper, we assume some familiarity with basic LCF concepts, but we will briefly review how these concepts are realised in ProofPower. The purpose of this section

is to provide a miniature primer on ProofPower and how it is implemented. This is intended to give a feel for the user experience using and programming an LCF style system in the traditional way for comparison with the Tinker approach. This section also provides technical background for those interested in the details of how Tinker is connected to the theorem prover. Readers familiar with some member of the LCF family are invited to skip or skim this section on a first reading.

Recall that in LCF terminology, the programming language used to implement the system is referred to as the metalanguage (or ML) while the language of the logic implemented by the system is referred to as the object language. ProofPower is implemented as a large library of functions that are invoked from its metalanguage ProofPower-ML, which is the interactive functional programming language Standard ML with extensions to support convenient entry of object language constructs. Object language terms are represented by an abstract data type *TERM* with a constructor for each syntactic category in the object language. Values of type *TERM* can be entered using object language concrete syntax enclosed in Quine corners, '⌜' and '⌝'. So, for example, the following ML command line:

SML

$$val\ tm1 = ⌜(\lambda x \bullet \lambda f \bullet\ f\ x)\ 1⌝;$$

causes the string of symbols between the Quine corners to be parsed and type-checked resulting in a value of type *TERM* that is bound to the ML variable *tm1*. (For customer-oriented reasons, the design of the concrete syntax for HOL in ProofPower was heavily influenced by the Z notation [60], hence the rather heavyweight bullets in the λ-abstraction.)

In the HOL logic, a theorem is a sequent $\phi_1, \ldots, \phi_n \vdash \phi$, asserting that, if the hypotheses $\phi_1, \ldots, \phi_n$ hold, then so does the conclusion ϕ (here ϕ and the ϕ_i are propositions, i.e., terms of type *BOOL*). Theorems are implemented as an abstract data type *THM* with a constructor for each primitive inference rule schema of the logic, parametrised by the antecedents of the rule schema and any other information needed to instantiate the schema. For example, one primitive inference rule schema is an axiom schema asserting that any β-redex is equal to its β-reduct. It is implemented by a constructor *simple_β_conv* with a single parameter that identifies the β-reduct. If (after executing the command above), we execute:

SML

$$val\ thm1\ =\ simple\_\beta\_conv\ tm1;$$

the system responds with:

ProofPower Output

$$val\ thm1\ = \vdash\ (\lambda\ x\ f \bullet\ f\ x)\ 1\ =\ (\lambda\ f \bullet\ f\ 1)\colon THM$$

indicating that a value of type *THM* with the appropriate instance of the β-reduction axiom as its conclusion has been bound to the ML variable *thm*. (Note that the pretty-printer has used a short-hand form for the nested λ-abstraction, as shall we in future examples.)

The constructor *asm_rule* implements the axiom schema containing all theorems of the form $\phi \vdash \phi$ and the constructor *eq_trans_rule* implements the rule schema for transitivity of equality. Putting these together, if we execute:

SML

```
val thm2 = eq_trans_rule (asm_rule ⌜H = (λx f•f x) 1⌝) thm1;
```

the system responds with

ProofPower Output

```
val thm2 = H = (λ x f• f x) 1 ⊢ H = (λ f• f 1): THM
```

Typically, ProofPower users do not use the primitive inference rules directly, but instead use derived proof procedures that operate at a higher-level. One widely used abstraction supporting equational reasoning is the notion of conversion [51]. A conversion is a function of type $TERM -> THM$, which, by convention, when passed a term t, returns a theorem with conclusion of the form $t = t'$. The primitive inference rule *simple_β_conv* discussed above is an example of a conversion, which proves all theorems of the form $\vdash (\lambda x \bullet t)u = t[u/x]$. Conversions are often used to package various kinds of normal form. For example, the conversion *anf_conv* implements a normal form for natural number arithmetic expressions. If we execute:

SML

```
val thm3 = anf_conv⌜7*(11 + x)*(13 + y)⌝;
```

the system responds with:

ProofPower Output

```
val thm3 = ⊢ 7 * (11 + x) * (13 + y) =
        1001 + 91 * x + 77 * y + 7 * x * y: THM
```

There is a heavily used family of conversions that work by rewriting with equational theorems such as the definitions of library functions:

SML

```
val thm4 = rewrite_conv[map_def, pair_ops_def]
                        ⌜Map Fst [(1, 2); (3, 4); (5, 6)]⌝;
```

Here *map_def* and *pair_ops_def* refer to the theorems representing the definitions of the *Map* combinator and the constructor $(,)$ and destructors *Fst* and *Snd* for pairs. (The ProofPower syntax follows the old HOL tradition of using semi-colons to separate the elements of lists.) This results in:

ProofPower Output

```
val thm4 = ⊢ Map Fst [(1, 2); (3, 4); (5, 6)] = [1; 3; 5]: THM
```

Many of the standard proof procedures provided in ProofPower are parametrised by what is called a *proof context*: a named collection of standard transformations to apply to a problem. The proof context allows the standard proof procedures to be tailored to particular problem domains and proof techniques. Higher-order functions are provided to allow proof procedures of various types to be executed in a specified proof context. For example, the function *PC_C1* executes a function returning a conversion in a specified proof context. Thus the following command performs a single step of rewriting in the proof context *sets_ext* designed for reasoning about sets using extensionality:

SML
```
val thm5 = PC_C1 "sets_ext" once_rewrite_conv[]
                        ⌜{x | x < 20} = {x | x ≤ 19}⌝;
```

```
val thm5 = ⊢ {x|x < 20} = {x|x ≤ 19} ⇔
              (∀ x• x ∈ {x|x < 20} ⇔ x ∈ {x|x ≤ 19}): THM
```

The following command rewrites in the proof context *sets_ext* until no more rewriting is possible:

SML
```
val thm5 = PC_C1 "sets_ext" rewrite_conv[] ⌜{x | x < 20} = {x | x ≤ 19}⌝;
```

which reduces the problem to pure arithmetic:

ProofPower Output
```
val thm5 = ⊢ {x|x < 20} = {x|x ≤ 19} ⇔ (∀ x• x < 20 ⇔ x ≤ 19): THM
```

Conversions can be composed using the infix combinator *THEN_C*. If we compose rewriting in the proof context *sets_ext* with rewriting in a proof context designed to deal with linear natural number arithmetic the problem reduces to truth and we can derive a proof of our original equation:

SML
```
val thm7 = (PC_C1 "sets_ext" rewrite_conv[] THEN_C
            PC_C1 "lin_arith" rewrite_conv[])
                    ⌜{x | x < 20} = {x | x ≤ 19}⌝;
val thm8 = ⇔_t_elim thm7;
```

yielding:

ProofPower Output
```
val thm7 = ⊢ {x|x < 20} = {x|x ≤ 19} ⇔ T: THM
val thm8 = ⊢ {x|x < 20} = {x|x ≤ 19}: THM
```

While forward proof using conversions and other inference rules gives a powerful approach to programming proof procedures, a more natural and productive approach to finding proofs interactively is a goal-directed search, starting with the assertion you wish to prove as the initial goal and transforming each goal into subgoals that entail that the goal and are (hopefully) easier to prove. The transformations are effected by what Milner christened *tactics*: ML functions that map a goal to a pair comprising *(i)* the list of subgoals and *(ii)* a proof, i.e., a function that will prove the goal given theorems validating the subgoals. As in other HOL systems, goals in ProofPower comprise a list of assumptions and a conclusion, so this simple but powerful idea is captured in the following type declarations:

```
type GOAL   = TERM list * TERM;
type PROOF  = THM list -> THM;
type TACTIC = GOAL -> GOAL list * PROOF;
```

For example, consider the goal: ([],⌜1 < 2 ∧ 2 < 3 ⌝)(with an empty list of assumptions). A tactic (namely ∧_tac) might reduce this goal to:

```
([([], ⌜1 < 2⌝) , ([], ⌜2 < 3⌝)],
        fn [th1, th2] => ∧_intro th1 th2) : GOAL list * PROOF
```

I.e., it gives us two subgoals with conclusions $1 < 2$ and $2 < 3$ respectively, together with a function, which, given a list comprising two theorems that validate these subgoals, will use $\wedge$-introduction to return a theorem validating our original goal.

This approach to interactive proof was supported from the earliest days of LCF (see [25] for the history) and came into its own when Paulson implemented the first interactive package for managing the subgoal state during the user's search for a combination of tactics that will prove their goal.

In the ProofPower subgoal package, the goal with assumptions $t_1, \ldots, t_n$ and conclusion t is internally represented as a single term that is logically equivalent to the universal closure of $t_1 \wedge \ldots \wedge t_n \Rightarrow t$. The logical state of the proof search is captured in a theorem whose conclusion represents the original goal and whose assumptions represent the outstanding subgoals. When a tactic is applied to a goal, the corresponding assumption is replaced by terms representing the list of subgoals returned by the tactic. Assumptions are labelled by dot-separated lists of natural numbers, representing a position in a tree whose root corresponds to the original goal and whose nodes correspond to tactic applications which result in more than one subgoal. In any state there is a current subgoal that tactics are applied to. A function *set_labelled_goal* is provided to allow the user to navigate around the outstanding subgoals.

A session with the subgoal package may be initiated with the *set_goal* command:

SML

$$set\_goal([], \ulcorner \forall x \bullet (1, x) \in \{(a, b) \mid a = 1 \vee b \geq 2\} \wedge (x \geq 3 \vee x \leq 4) \urcorner);$$

The system responds by printing out the state of the proof search. There is 1 goal and its label is the empty string:

ProofPower Output

Now 1 goal on the main goal stack

(* *** *Goal* "" *** *)

(* ?⊢ *)$\ulcorner \forall\ x \bullet (1, x) \in \{(a, b) | a = 1 \vee b \geq 2\} \wedge (x \geq 3 \vee x \leq 4) \urcorner$

Here the symbol ?⊢ is used in the display of a goal as indicative of a sequent that has not yet been proved. It is included as an ML comment to facilitate copying and pasting the output as executable code in an ML script.

At this point, we have several choices about the tactic to apply. If we want to take a fine-grained approach, we could apply tactics that exactly match the outer two layers of the logical structure:

SML

$$a(\forall\_tac\ THEN\ \wedge\_tac);$$

Here $THEN$ is a *tactical*, i.e., an operator that constructs new tactics from old, in this case by a form of sequential composition. This results in:

ProofPower Output

Tactic produced 2 subgoals:

(* *** *Goal* "*2*" *** *)

(* ?⊢ *)⌜$x \geq 3 \vee x \leq 4$⌝

(* *** *Goal* "*1*" *** *)

(* ?⊢ *)⌜$(1, x) \in \{(a, b) | a = 1 \vee b \geq 2\}$⌝

Alternatively, we could repeatedly apply the general purpose tactic *strip_tac* which applies a standard simplification to logical connectives if possible and proof context-dependent transformations to atomic formulas. The following command does this after undoing what we have just done:

SML

undo 1; *a*(*REPEAT strip_tac*);

ProofPower Output

Tactic produced 2 subgoals:

(* *** *Goal* "*2*" *** *)

(* *1* *)⌜$\neg\ x \geq 3$⌝

(* ?⊢ *)⌜$x \leq 4$⌝

(* *** *Goal* "*1*" *** *)

(* ?⊢ *)⌜$(1, x) \in \{(a, b) | a = 1 \vee b \geq 2\}$⌝

Note how in goal 2, the disjunction has been dealt with by asking us to prove its right-hand side on the assumption that its left-hand side is false. (The assumptions are displayed above the conclusion of the goals with numbers in ML comments to identify them. In this case there is just one assumption.) Let us assume that this is not quite what we wanted; so we undo it and try again but only stripping off two layers of connective:

SML

undo 1; *a* (*strip_tac THEN strip_tac*);

This gives us:

ProofPower Output

Tactic produced 2 subgoals:

(* *** *Goal* "*2*" *** *)

(* ?⊢ *)⌜$x \geq 3 \vee x \leq 4$⌝

(* *** *Goal* "*1*" *** *)

(* ?⊢ *)⌜$(1, x) \in \{(a, b) | a = 1 \vee b \geq 2\}$⌝

Looking at goal 1, we see it should become trivial once the set notation has been eliminated using standard properties of set comprehensions and pairs. So iterating *strip_tac* should do just what we want in the *sets_ext* proof context. To do this we use the *tactical PC_T1*, which does for tactics what *PC_C1*, discussed above, does for conversions:

SML

a(*PC_T1* "*sets_ext*" *REPEAT strip_tac*);

which results in:

ProofPower Output

Tactic produced 0 subgoals:
Current goal achieved, *next goal is*:

(* *** *Goal* "*2*" *** *)

(* ?⊢ *)⌜$x \geq 3 \vee x \leq 4$⌝

We recognise that this problem is entirely in the domain of linear natural number arithmetic. The proof context *lin_arith* for this domain includes a decision procedure that we can access via a generic tactic *prove_tac*:

SML

a(*PC_T1* "*lin_arith*" *prove_tac*[]);

This completes the proof search:

ProofPower Output

Tactic produced 0 subgoals:
Current and main goal achieved

We can now extract our theorem from the subgoal package:

SML

val thm9 = *pop_thm*();

ProofPower Output

Now 0 goals on the main goal stack
val thm9 = ⊢ ∀ x• $(1, x) \in \{(a, b) | a = 1 \vee b \geq 2\} \wedge (x \geq 3 \vee x \leq 4)$: *THM*

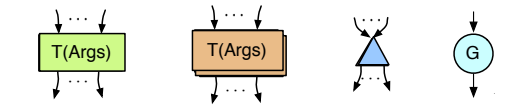


Fig. 1. Types of graph nodes for PSGraph

2.2 PSGraph

In PSGraph, tactics are represented as directed, typed, hierarchical and open graphs. A graph consists of *boxes*, representing "processes" and typed (labelled) *wires* that connect them together. A process box is labelled by a tactic, which can either be an existing ProofPower tactic or another graph, where the latter introduces hierarchies. The graphs are *open* in that wires need not be connected to a box at both ends, but can be left open to represent graph inputs and graph outputs. Evaluation is achieved by adding input goals to a graph input wire. The goals will then flow through the graph; each step will apply a tactic to an incoming goal, consume the goal, and add the resulting subgoals to its output wires. This process will continue until all subgoals appear on the graph output wires. These subgoals will then be returned. All wires are labelled by *goal types*, which are predicates on a goal that are used to direct goals to the correct tactic.

Fig. 1 shows the types of boxes that can appear in a PSGraph. An *atomic* tactic is an existing ProofPower tactic, possibly parametrised. A parameter could for example be the name of a rewrite rule to apply or a term used to instantiate a variable. Note that if the list of parameters ($Args$) is empty, then we can write T instead of $T()$. A *graph* tactic is labelled by a named graph, which we can look up. For graph tactics, the arguments ($Args$) relate to the scope of the variables of the goal node environment, which we return to in §3.1. An *identity* tactic is used to split and merge multiple wires and, as discussed below, will not have any side-effects on the proof state or goal nodes. The final type of box is a *goal*. This is only used for evaluation and cannot be added to a PSGraph by the user. It contains sufficient information to evaluate and link to the ProofPower proof state, including:

—the name given by ProofPower for the goal;
—the internal representation of the goal in ProofPower; and
—an environment that is used to support variables in the graph, which is discussed in detail in §3.1.

When displaying the goals, we will only show the name (see e.g. Fig. 4). We return to goals when discussing evaluation below.

Finally, the wires are labelled by *goal types*, which are predicates defined on goals. Intuitively, these provide information about some characteristics, such as "shape", of a goal, which are used to influence the path a goal takes as it passes through the graph. We develop a language for expressing these in §3.2, and defer the details to that section.

A simple but complete example of a PSGraph is given in Fig. 2. This is a PSGraph encoding of the proof discussed in §2.1. The input wire is labelled by

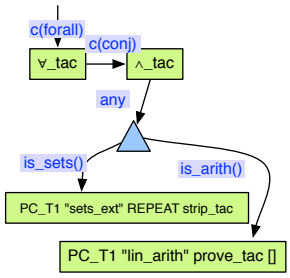


Fig. 2. A PSGraph encoding of the proof discussed in §2.1

$$R_{eval} = \left\{ \frac{\begin{array}{c}\langle defs, atoms\rangle \vdash h_{10} : H_1 \cdots \langle defs, atoms\rangle \vdash h_{1m} : H_1 \\ \vdots \\ \langle defs, atoms\rangle \vdash h_{n0} : H_n \cdots \langle defs, atoms\rangle \vdash h_{nk} : H_n \\ ([h_{10}, \cdots, h_{n0}, \cdots, h_{n0}, \cdots, h_{nk}], \_) = T(Args)\ g\end{array}}{\begin{array}{c}\langle defs, atoms\rangle \vdash g : G\end{array}} \right\}$$

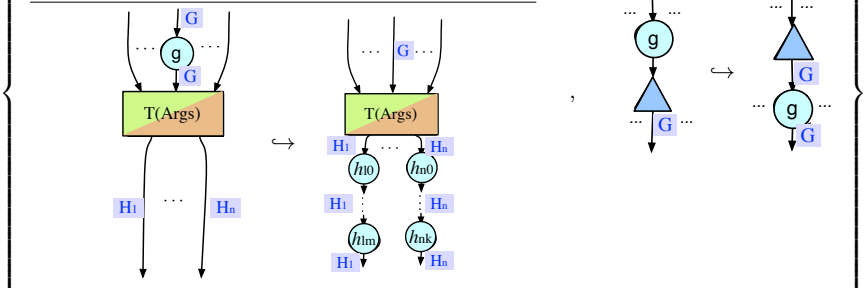


Fig. 3. Evaluation of PSGraph.

$c(forall)$ which means that the conclusion (c) must be a universal quantifier. The $\forall\_tac$ tactic is applied to it followed by the $\wedge\_tac$ tactic, as long as the conclusion of the goal is a conjunction. The *any* goal type always succeeds. The *identity* tactic is then used to separate the goals that are arithmetic (is_arith) from those that are set theoretic (is_sets), where the suitable tactic is applied in both instances.

2.2.1 *Evaluation.* In order to initialise the evaluation (i.e. proof of) of a subgoal by a PSGraph, a ProofPower goal must be provided. This is achieved by calling *top_goal()*, which will provide the first subgoal from the ProofPower subgoal package. From this subgoal, a goal node with an empty environment is created, and added to a graph input wire where the goal type is satisfied. A step of the proof will apply a tactic box to a goal that is on an input wire. Each generated subgoal (if any) will be added to an output wire where the goal type is satisfied. This process will continue until *termination*:

Definition (Termination). *A graph has terminated, if for all goals g of the graph, g is either on a graph output wire or it is wired to another goal.*

It follows by induction over the number of goals present that all goals are (directly or indirectly) on graph output wires.

It is worth noting that (as with other tactic languages) termination is not guaranteed. One example of non-termination is that an atomic tactic from the theorem prover may not terminate; another example is a PSGraph that contains a non-terminating loop, e.g. as a result of rewriting in presence of commutative operators.

As the proof state is handled by ProofPower's subgoal package, evaluation is only concerned with how the subgoals "flow" from the graph input to the graph output wires. Two properties are crucial for a *successful* evaluation step:

—No subgoals are lost, that is if a tactic produces a subgoal then it will appear on the graph.

—No subgoals are duplicated in the graph.

At the graph level, evaluation is achieved by graphical rewriting, where $l \hookrightarrow r$ is a rule that rewrites l to r[3]. Fig. 3 gives the set R_{eval} of rewrite rules used to evaluate a PSGraph. We use a notation where side-conditions are above the line and the rewrite is below the line. Each rule is non-deterministic in the sense that there may be several ways to apply it in a given situation. Evaluation of a graph is achieved by applying rules from R_{eval} repeatedly until no rules are applicable. At this point evaluation have either failed or successfully terminated.

The simplest case is the *identity* box, shown rightmost in Fig. 3. Here, the input goal and the output goal is the same as the node is essentially used to fork the goal to the correct target box. $\langle defs, atoms \rangle \vdash g : G$ is a predicate that holds if goal g satisfies goal type G. The ellipses illustrate that there could be other input and output wires. Note that if there are more than one output wire where g satisfies the goal type then there will be multiple rewrites. Each of these rewrites will be a separate branch of the search space. We will return to the goal type predicate in §3.2.

The leftmost rule of Fig. 3 shows the evaluation of an atomic or graph tactic. For an atomic tactic, $T(Args)\ g$ is the result of applying the tactic (in ProofPower). This is described in §2.2.3. For a graph tactic, this is the result of evaluating the nested graph as discussed below. For these tactics, there will be a side-effect on the proof state, which we return to in §2.2.3. The case when $T(Args)$ is an *atomic* tactic can be summarised as follows:

(1) Apply tactic $T(Args)$ to obtain a list of subgoal nodes.
(2) Consume g from the graph.
(3) Add all valid combination of the resulting subgoals to output wires.

Fig. 4 illustrates some of the steps of the flow through the proof strategy of Fig. 2 applied to the example of §2.1. In the left-most graph, g holds the initial goal:

$$(* \vdash ?*)^{\ulcorner} \forall\ x \bullet\ (1,\ x) \in \{(a,\ b) | a = 1 \lor b \geq 2\} \land (x \geq 3 \lor x \leq 4)^{\urcorner}$$

It then applies R_{eval} twice, which will first apply universal introduction followed by conjunction introduction, introducing two new subgoals, $h1$:

$$(* \vdash ?*)^{\ulcorner} (1,\ x) \in \{(a,\ b) | a = 1 \lor b \geq 2\}^{\urcorner}$$

[3]When applied to a graph, this rule will match l with a subgraph and replace this with r. For more details see [17, 18].

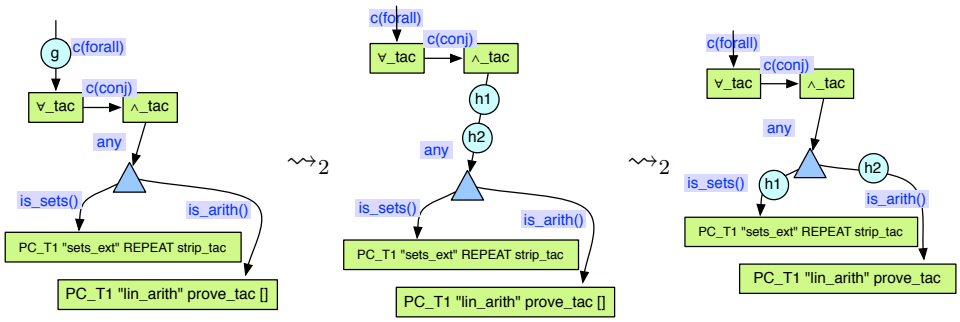


Fig. 4. Example evaluation of PSGraph.

and $h2$:

$(* \ ?\vdash \ *)\ulcorner x \geq 3 \vee x \leq 4 \urcorner$

Next it applies the identity tactic to $h2$ and then $h1$, with the result shown in the right-most graph. This is used to route the goals to the correct tactic to complete the proof. $h1$ is set-theoretic and thus goes down the left branch while $h2$ is arithmetic and follows the right branch. Note that when there are multiple goals, as in the right most graph, the order in which goals are evaluated will have no impact on the end result[4].

When T in the leftmost rule of Fig. 3 is a *graph* tactic, the arguments $Args$ of T are used to introduce *local scoping*: any variable not in $Args$ is "fresh" in the nested scope and will not have global effect. The evaluation can be summarised as:

(1) Consume g from the graph.
(2) Lookup the graph G which T points to.
(3) Constrain the environment of g to variables in $Args$ and add this to an input wire of G (such that the goal type is satisfied). If there are multiple satisfying input wires, then one branch will be generated for each.
(4) Evaluate G until termination.
(5) Add all valid combination of the goals on the output wires of G to the output wires of the graph tactic T.

If any steps fail then evaluation of this node fails. Note that when adding a resulting subgoal to the output of T in the last step, the subgoal will be given the environment of g, with values of $Args$ replaced by those in the resulting subgoal. We will return to how this works in §3.1, after introducing environments more formally.

To illustrate how a graph tactic is evaluated, consider the PSGraph in Fig. 5. It contains a graph tactic *conj_imp*, with two input wires and one output wire. The input wires require that the conclusion of the a goal is either an implication ($c(implies)$) or a conjunction ($c(conj)$). The output must have an hypothesis that is the same as the conclusion. It is then proven by assumption by *concl_in_asms_tac*.

[4]This would not have been the case in presence of shared meta-variables between goals, a feature that is not currently supported in either ProofPower's subgoal package or PSGraph.

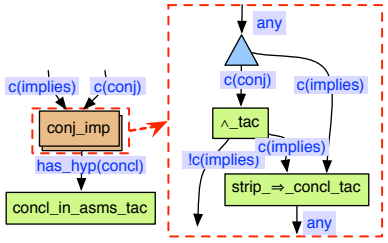


Fig. 5. Example PSGraph with graph tactic.

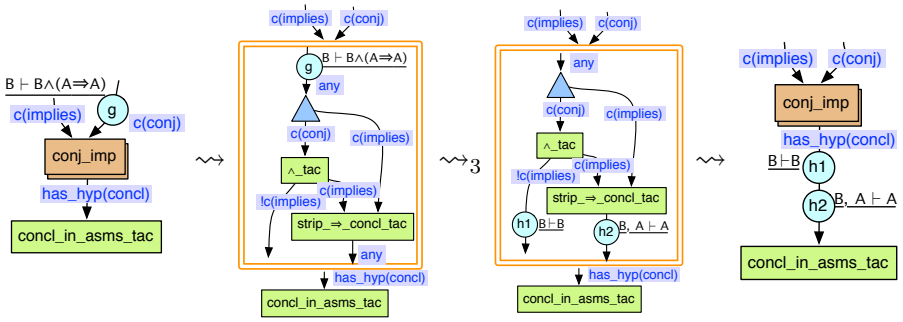


Fig. 6. Example evaluation of graph tactic in PSGraph.

The graph nested by *conj_imp* is shown in the red stippled box of Fig. 5 (right). Depending on the input goal, it will either break up the conjunction or the disjunction, and in the former case, it may also be followed by breaking up a disjunction. In order to illustrate several aspects of evaluation, the number of input/output wires of the nested graph, and their goal types, deviates from the parent *conj_imp* graph tactic.

Consider Fig. 6, which shows the key evaluation steps for a goal g:

$(*\ \ 1\ *)\ulcorner B \urcorner$
$(*\ ?\vdash *)\ulcorner B \wedge (A \Rightarrow A)\urcorner$

applied to one of the input wires of the graph of Fig. 5.

In the first step, this goal is consumed from the parent graph and added to one of the input wires of the nested graph. Note that for evaluation there is no correspondence between the input wires of the parent box (in this case labelled by $c(conj)$), and the input wire of the nested graph (here *any*). The goal g is simply added to any input wire of the nested graph where the goal type holds (with a separate branch in the search space for each such wire). In this case there is only one possibility. It will then go through three steps of evaluation of the nested graph, and at the end there are two goals, $h1$ and $h2$, on the output wires of the nested graph. According to the definition of termination, the graph tactic has now terminated. The (nested) graph will then "return" the list of goals $[h1, h2]$[5]. These are then added to the output wires where the goal type is satisfied, as the case

[5]The order of the goals returned is irrelevant.

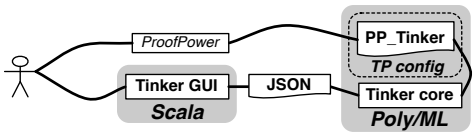


Fig. 7. Tinker architecture.

is for an atomic tactic. In this case, both matches the goal type of *conj_imp*'s output wire (*has_hyp*(*concl*)) since the conclusion is found in the list of hypothesis. Again, note that there is no relationship between the goal types of the output wires of the nested graph (!*c*(*implies* and *any*), and those of the nesting graph tactic (*has_hyp*(*concl*)). The *concl_in_assms_tac* tactic will then discharge both $h1$ and $h2$ by assumption.

2.2.2 *Architecture & GUI of the Tinker tool.* The Tinker tool [27, 44] implements PSGraph with support for the *Isabelle*, *Rodin* and *ProofPower* theorem provers. Here, we will focus on the ProofPower version only. Tinker consists of two parts: the *CORE* and the *GUI*. These are shaded in separate boxes in Fig. 7. The core implements the main functions of Tinker. Most of the functions are implemented using ML functors to achieve theorem prover independence. In order to connect a theorem prover to Tinker, and use its GUI and basic functionality, a ML structure that implements a provided ML signature called *PROVER* has to be provided. This will enable basic usage of Tinker and the GUI. In Fig. 7 the structure implementing this signature is called *PP_Tinker*.

Note that our ambition is not to replace the existing tactic language, but to offer a different view of tactics. Tinker is designed to support a dynamic interplay between a PSGraph and the existing tactic language, where the level of atomicity of the atomic tactics used in PSGraph is flexible. This enables developers to decide themselves which parts are best to express in PSGraph and which are not.

The remaining of this subsection, as well as the next subsection, are intended for readers interested in the details of how Tinker connects to theorem provers. Other re.

The *PROVER* signature includes both the types and functions required. It has to know how types, terms, theorems and contexts are represented:

type typ
type term
type thm
type context

To illustrate, in ProofPower these are instantiated to:

type typ = *TYPE*
type term = *TERM*
type thm = *THM*
type context = *string list* ∗ *string*

The CORE communicates with the GUI (written in Scala) over a JSON socket protocol, which requires serialisation functions for some of this types (via strings), e.g.:

val trm_of_string : context -> string -> term
val string_of_trm : context -> term -> string

The GUI allows users to develop proof strategies in a mostly graphical approach, and to debug proof strategies with controlled interactive inspections.

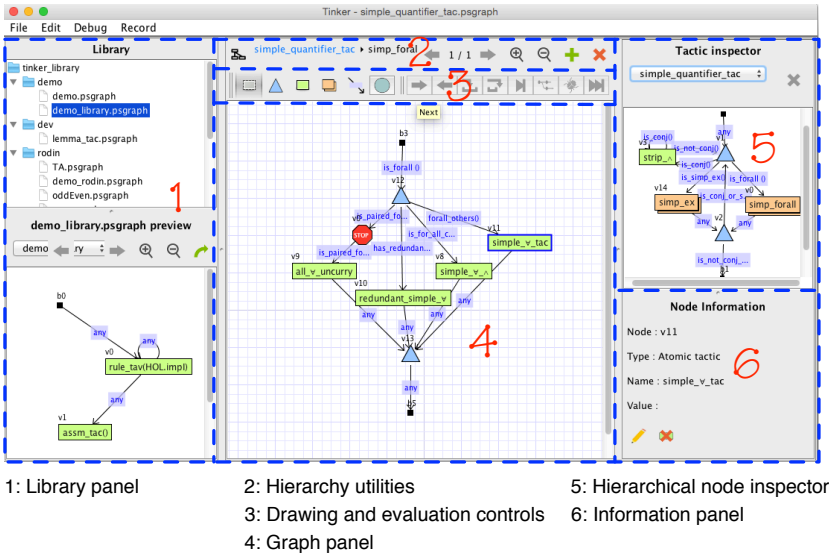


Fig. 8. The Tinker GUI [44].

Fig. 8 shows the components of the GUI. The *graph panel* is the main area where users view and edit the graph of current proof strategies. With the interactive options in the *drawing and evaluation control panel*, users can develop graphs in a click and drag style, and step through evaluation with controls such as step over a graph tactic. When a user selects a node or edge in the graph panel, the detail information of the node or edge will be showed in the *information panel*. To facilitate developing hierarchical graphs, the *hierarchical node inspector* panel allows users to preview the sub-graphs of a graph tactic; and the *hierarchy utility* panel shows the depth and path of the graph in the graph panel. For reusing existing proof strategies, there is also a *library panel* to preview existing PSGraphs, and import them to the graph panel. The core is implemented on top of the Quantomatic graphical rewrite system [36]. Videos of interaction with the different features of the GUI are available from [42].

Tinker also needs to know about tactics and their execution, also provided via implementation of the *PROVER* signature:

type tactic

Tinker uses the underlying prover's proof state and goal representation augmented with some additional book-keeping information represented in the following ML types:

type pplan
type pnode

The proof state (type *pplan*) is mainly used to link with the proof state of the theorem prover and keep track of the goals that are "active" in the PSGraph. To illustrate, in ProofPower this is a record where the main fields are the underlying goal state of ProofPower and the goals that the PSGraph are allowed to work on:

type pplan = {*gstate*: *GOAL_STATE*, *opengs* : *pnode table*, ...}

The type *pnode table* is a map from a *string* to a *pnode*. The key fields of the goal representation (*pnode*) are the name of the goal, its internal representation and an environment:

type pnode = {*pname* : *string*, *g*: *GOAL*, *env* : *env*, ...}

The Tinker representation of a tactic uses these types, and therefore has type:

type appf = *pnode* ∗ *pplan* −> (*pnode list* ∗ *pplan*) *Seq.seq*

To be used in Tinker, the underlying prover's tactics (i.e. functions of type *tactic*) have to be "lifted" to *application functions* with the above type *appf*.

2.2.3 *Tactic "lifting".* An atomic tactic $T(Args)$ encapsulates a tactic of the underlying theorem prover, i.e., ProofPower for the purposes of the present paper. Recall from §2.1 that a ProofPower tactic maps a goal to a pair of new subgoals and a validation function. PSGraph uses ProofPower's existing package for handling the subgoal state, meaning we can ignore the validation function at this level. We write

$$(gs, \_) = T(Args)\ g$$

to denote that $T(Args)$ returns the list of subgoals gs when applied to g.

We need to be able to connect ProofPower tactics, which works on goals, to our atomic tactic boxes, which works on goal nodes (i.e. type *pnode*). The simplest case is when there are no arguments, which we can just write T. All the atomic tactics of Fig. 2 are examples of this case. Here, the label of the atomic box, e.g. *PC_T1 "lin_arith" prove_tac[]*, is wrapped into a function that will take a goal node and produce a list of new goal nodes as follows. It will extract the goal name from the input goal node and set this to be the top goal in ProofPower's goal stack, using a ProofPower function called *set_labelled_goal*. It will then apply the wrapped tactic (to the goal at the top of the stack), and finally it will put each goal name and goal into a goal node, together with the environment of the input node. We call this process "lifting" of the ProofPower tactic to PSGraph.

As there are no arguments, the string of the atomic tactic box is simply parsed as ML code and applied. Such parsing has to be provided in the *PROVER* signature:

val exec_str : *string* −> *unit*

This is a generic parser interface, which is also used for other parsing tasks. Internally, Tinker provides functionality to cast it to the correct type after some minor user configurations. For tactics, it will cast it to the type *tactic*. The "lifting" into the *appf* type is trivial.

When there are arguments, then we need to manually provide some ML code for this[6]. To illustrate, consider the ProofPower tactic *prove_tac* discussed above. We would like to parameterise over the proof context, which we can do by providing the name of a proof context as a parameter:

```
fun prove_with_ctxt0 ctxt = PC_T1 ctxt prove_tac [];
```

However, this will not work in PSGraph. In order to use such parametrised tactics, the function needs to have a different type. Internally, the arguments of an atomic tactic are represented using a deep embedding, i.e. as a list of an inductive datatype with a constructor for each type:

```
datatype arg_data = A_Trms of term list | A_Var of string | A_Str of string | ···
```

Arguments must be passed as a list of *arg_data*, and such arguments have to be reflected in the tactic type of *PROVER*; for ProofPower it is[7]:

```
type tactic = arg_data list -> TACTIC
```

With this type, the signature has to be provided an interpretation of tactics in terms of the defined application function (type *apps*):

```
val apply_tactic : arg_data list -> tactic -> appf
```

Returning to our example, we represent the context as a string, so we provide the following "lifting" function:

```
fun prove_with_ctxt [A_Str ctxt] = PC_T1 ctxt prove_tac []
  | prove_with_ctxt _ = fail_tac;
```

As a result, *prove_with_ctxt(A_Str lin_arith)* will apply this function, with *lin_arith* parsed as a string.

We will introduce a new type of tactic in §3.1, while §4 contains many examples of tactics with and without arguments.

3. A (MOSTLY) GRAPHICAL DEVELOPMENT & DEBUGGING FRAMEWORK

In §4 we will showcase PSGraph and the Tinker tool by developing, debugging and refactoring several case studies adapted from the existing ProofPower developments. To support this we first extend PSGraph and Tinker with new features.

In §3.1 and §3.2 we add features that are mainly beneficial for development: in §3.1 we introduce a new family of tactics used to exchange information and constraints between tactics and goal types; while in §3.2 we develop a goal type that allows us to hide low-level details in the graphs to improve readability.

In §3.3 and §3.4 we develop support for debugging: §3.3 introduces breakpoints to PSGraph, while §3.4 describes a simple, yet useful, logging mechanism for Tinker.

3.1 "Environment" tactics

Recall that the type of a tactic is from a goal to a pair consisting of a list of new subgoals and a validation function. In an atomic tactic box, such tactic is then

[6] We hope to introduce some level of automation for this process in the future.
[7] For example, Isabelle in addition needs the context and the index of the subgoal as arguments.

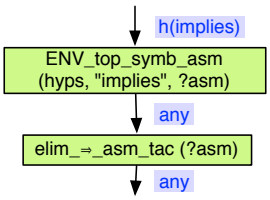


Fig. 9. An example of "environment" tactic

"lifted" to work on the goal nodes that are in the graph. In addition to the actual goal, such a goal node also contains an *environment*, which we introduce here:

DEFINITION (ENVIRONMENT). *An environment* Env *is a function*

$$\mathsf{Var} \to \mathsf{EnvVal}$$

where Var *is a finite set of variables named by strings prefixed with a '?' and* EnvVal *is the disjoint union*

$$T \uplus N \uplus T^* \uplus N^*$$

where T and N denote the set of all terms and names respectively, where a name is an arbitrary uncapitalised string and X^* denotes the set of lists of elements of X[8].

The validity of a name depends on the context. For example, it could be a named lemma, which will only be valid if that lemma exists. Two special names are: *concl*, which refers to the conclusion of a given goal; and *hyps*, which refers to the (list of) hypothesis of the goal.

An environment is used to pass information between tactics and between tactics and goal types. This could be to extract some information at one point in the proof and use it later. For example, consider Fig. 9. Here, *ENV_top_symb_asm* looks for a hypothesis that starts with an implication, and binds it to a variable $?asm$. In the next atomic tactic, *elim_* ⇒ *_asm_tac* will apply implication elimination to the hypothesis that $?asm$ is bound to.

Within graph tactics, local scoping of the environment is achieved by using the arguments *Args* of the box. Recall the evaluation steps of graph tactics, as given in §2.2.1. We are given a goal with an environment $\{?x \mapsto v_1, ?y \mapsto v_2\}$ and a graph tactic $t(?x)$. When evaluating g for the graph t references, the environment $\{?x \mapsto v_1\}$ is provided. Assume that on termination of t that a new goal g' has the environment $\{?x \mapsto v_3, ?z \mapsto v_4\}$. This will return a goal g' with environment $\{?x \mapsto v_3, ?y \mapsto v_2\}$. This illustrates how environments are constrains for "local computation" within graph tactics.

Now, the problem with *ENV_top_symb_asm*, discussed above, is that it binds $?asm$, meaning the result of applying it is a change to the environment, while a "lifted" tactic will change the goal (and proof state) and cannot change the environment. In this case, this issue could be overcome by combining these two

[8]Note that T and N are redundant as they can be represented as singleton sequences of T^* and N^*, respectively. We find it more natural to separate them, as they are often treated differently (e.g. some tactics only work on a single term). This will also simplify static checking, which we plan to add in the future.

atomic tactics into a single ProofPower tactic. However, part of the reason to have them as a separate boxes is to enable users to inspect the flow, and use Tinker's debugging features if a tactic application fails. By combining them into a single tactic the granularity becomes too terse for such analysis. A second problem is that there are more complex examples where there are tactic in between binding and using a variable, which we will see examples of in §4. For these cases the solution of merging boxes will not work.

Instead we introduce a type of atomic tactic that works on the environment, which we call *environment tactics*[9]:

Definition (Environment tactic). *An* environment tactic*, is an atomic tactic, with a name prefixed by 'ENV_', whose underlying function is a function*

$$\mathsf{Env} \rightarrow \mathsf{Env}^*$$

The rest of this section focuses on implementation issues of environments and environment tactics. This material is intended for readers interested in the technical details of the connection between Tinker and a theorem prover. It can can be skipped if desired. To support environment tactics, the *PROVER* signature is augmented with new types for an environment and an environment tactic:

```
type env = env_data table
type env_tac
```

The type *env_data table* is a map from a string to the type *env_data*, which holds the types an environment may contain

```
datatype env_data = E_Str of string | E_Trm of term  | ···
```

As environment tactics may also have arguments (*Args*), they have a dual type and application function to ProofPower tactics:

```
type env_tac = arg_data list -> env -> env list
val apply_env_tactic :  arg_data list  -> env_tac -> appf
```

From the types one can see that an environment tactic will not change the underlying proof state; it will only change the environment of a goal node (type *pnode*). However, they may still require features of the provers, such as matching of terms.

ENV_top_symb_asm is an example of an environment tactic. As we can see from Fig. 9, it takes three arguments: the first (*hyps*) is a list of terms, the second (*"implies"*) is a string, and the third (*?asm*) is a variable. The underlying function that has to be provided by the user will look something like:

```
fun ENV_top_symb_asm [A_Trms ts, A_Str s, A_Var var] env = ···
```

Note that Tinker will automatically lookup *hyps*, which is the list of hypothesis, before calling this function. We will see many examples of environment tactics in §4.

[9] An alternative to environment tactics, is to bind variables during matching in goal types, which is explained in §3.2. We have made a design decision to treat goal types as predicates, and therefore to not support this. This is a deliberately compromise made in order to cleanly separate concerns and to simplify the semantics of PSGraph composition (see [26]).

3.2 A constraint language to express goal types

Goal types are crucial in order to achieve maintainable proof strategies and to reduce the search space. These are represented as *constraints* on the goal, and to represent these, we develop goal types as a Prolog inspired *constraint language*. Prolog is a natural starting point as constraints can be combined in an elegant and declarative manner[10], and enables support for machine learning goal types using a technique based upon logic-based learning [19]. Analogous to how graphs are used to compose tactics, this language acts as a way of combining and re-using low-level atomic constraints, which the underlying prover needs to provide. By supporting recursive definitions, expressive constraints can be encoded, and lower-level details can be hidden in the goal types appearing on the graphs.

A relation may have (goal type) variables that are instantiated. This is discussed below. However, a goal type appearing in a graph cannot have any such variables. We therefore distinguish *goal type schemas*, which may have goal type variables, from *goal types*, which does not allow the use of such variables:

Definition (Goal type & goal type schema). *A* goal type schema *(GTS) is a predicate on a goal, defined by the following BNF:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} GTS & ::= & C, GTS \mid C. & Oa & ::= & As \mid \epsilon \\ C & ::= & L \mid !L & As & ::= & A \mid A, As \\ L & ::= & F(Oa) & A & ::= & T \mid N \mid \mathsf{GVar} \end{array}$$

Following from Definition 2 (Environment): T denotes a term; F denotes a named fact and N denotes a name, which in either case is an arbitrary uncapitalised string; and GVar *denotes a goal type variable, which is an arbitrary capitalised string. A* goal type *is a goal type schema without any goal type variables, i.e. with $A ::= T \mid N$*

To illustrate, the goal type schema

$$top\_symbol(T, Y).$$

expresses that the top level symbol of (goal type variable) T has to be (goal type variable) Y. An example goal type using this schema:

$$top\_symbol(concl, \wedge), has\_top\_symbol(hyps, \wedge).$$

It states that the conclusion and one hypothesis of the goal has to be a conjunction; as we shall see later, *has_top_symbol* can be defined in terms of *top_symbol*.

In practice, we have found that most goal types, such as $top\_symbol(T, Y)$, are constraints over terms. These may have to be provided by the underlying theorem prover, and we call them *atomic goal types*. One example atomic goal type is $top\_symbol(T, Y)$, while $any()$, which we can also write any, is provided by default. The *any* predicate will always succeed.

As will be seen in case studies, many goal types combines such atomic goal types. To achieve readable and intuitive proof strategies, low-level implementation details needs to be hidden to highlight the high-level concepts of the graph. To support that, a user can define a new goal type schemas, which can be used by a goal type:

[10] Compared with for example using the underlying implementation language of Tinker (ML).

Definition (Goal type schema definition). *A* goal type schema definition *(GTSD) is a* rule *defined by:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} GTSD & ::= & N(Ov) \leftarrow GTS & Ov & ::= & Vs \mid \epsilon \\ & \mid & N(Ov) \leftarrow GTS \;\; GTSD & Vs & ::= & \mathsf{GVar} \mid \mathsf{GVar}, Vs \end{array}$$

We often call the left hand side of '$\leftarrow$' the *head* and the right hand side the *body*.

To illustrate, the goal type scheme definition:

$$g\_h(X, Y) \leftarrow top\_symbol(concl, X), has\_top\_symbol(hyps, Y).$$

requires that the first argument is the top symbols of the conclusion and that there exists a hypothesis that has the top symbol of the second argument. This is used in the following definition:

$$g(X) \leftarrow g\_h(X, \wedge).$$
$$g(X) \leftarrow g\_h(X, \vee).$$

Here, $g(X)$ is defined to be a goal type scheme where the conclusion has the top symbol given by the argument X, and there is a hypothesis with either $\wedge$ or $\vee$ as top symbol. $g(\wedge)$ is an example of a a valid *goal type* using this schema.

One can also use variables in the body not present in the head. This is used to pass arguments between literals of the body. For example,

$$concl\_top\_in\_hyp() \leftarrow top\_symbol(concl, X), has\_top\_symbol(hyps, X).$$

expresses that there is a hypothesis with the same top symbol as the conclusion.

Recall from Fig. 3 (§2.2.1) that in an evaluation step of a PSGraph, we need to check if a goal g satisfies a goal type G. This depends on the provided atomic goal types *atoms* and goal type definitions *defs*. We write

$$\langle defs, atoms \rangle \vdash g : G$$

to express that such relation holds. In order to determine this, information about the values of variables has to be passed between the clauses. For example, consider *concl_top_in_hyp()*. Here, the X of both the clauses in the body has to be the same; in other words, $has\_top\_symbol(hyps, X)$ needs to know the value of X from $top\_symbol(concl, X)$. To achieve this, an environment is passed between the literals. This environment is different from the environment in the goal node in that goal type variables are bound, and we call it a *goal type environment*:

Definition (Goal type environment). *A goal type environment (GTEnv) is a function:*

$$\mathsf{GVar} \rightarrow \mathsf{EnvVal}.$$

For $concl\_top\_in\_hyp()$, the result of applying the first *top_symbol* will be an environment with X bound, which is then used in the second application of *top_symbol*.

In order to specify $\langle defs, atoms \rangle \vdash g : G$, we introduce a relation that generates a goal type environment:

$$\langle defs, atoms \rangle \vdash \langle g, G \rangle \Downarrow gtenv$$

This should be read as: given a *context* consisting of a pair of the atomic goal types *atoms* and goal type definitions *defs*, and an *input* consisting of a pair of a goal g

and a goal type G, a goal type environment *gtenv* is produced. $\langle$*defs,atoms*$\rangle \vdash g : G$ can then be defined in terms of the existence of such a goal type environment:

$$(\langle \textit{defs,atoms} \rangle \vdash g : G) \Leftrightarrow (\exists\ \textit{gtenv}.\ \langle \textit{defs,atoms} \rangle \vdash \langle g, G \rangle \Downarrow \textit{gtenv})$$

The semantics of the $\Downarrow$ relation is inspired by Prolog with some key differences. Firstly, due to the "lifted" nature over atomic goal types, most of the unification work is provided by the underlying prover. Secondly, we have to work with two distinct environments. Thirdly, we have to communicate with the underlying theorem prover. We can therefore not use Prolog directly in a natural way, and decided to develop a domain specific version, which we provide big-step operational semantics for next.

The semantics of $\Downarrow$ are non-deterministic, in that it can generate multiple valid goal type environments (*gtenv*). As $\langle$*defs,atoms*$\rangle \vdash g : G$ is only concerned with the existence of a valid goal type environment, it does not matter which of the valid ones is found. The definition of the semantics uses auxiliary relations $\Downarrow_b$ and $\Downarrow_c$.

The evaluation of a goal type is a special case of the evaluation of the body of a goal type schema, the difference being that the relation is over a goal type environment, which is updated. This is evaluated by the relation $\Downarrow_b$, where b stands for 'body'. This is evaluated over an environment and a clause body. As there are no side-effects on the goal, the goal and its environment are moved to the context:

$$\frac{\langle env(g), defs, atoms, g \rangle \vdash \langle \{\}, G \rangle \Downarrow_b \textit{gtenv}}{\langle \textit{defs,atoms} \rangle \vdash \langle g, G \rangle \Downarrow \textit{gtenv}}$$

The body is either a single clause 'C.', or a clause followed by more clauses 'C, GTS'. As a result there are four cases: 'C, GTS', 'C.', atomic goal types and negated literals.

For the second case, the clause is evaluated. We write this as $f\ vs$ where vs is a list of the arguments. To evaluate this we try to find a definition of f in all the definitions. This is achieved by the $\Downarrow_c$ relation, where c stands for 'clause':

$$\frac{\langle env, defs, atoms, g \rangle \vdash \langle \textit{gtenv}, f\ vs, \textit{defs} \rangle \Downarrow_c \textit{gtenv}'}{\langle env, defs, atoms, g \rangle \vdash \langle \textit{gtenv}, f\ vs. \rangle \Downarrow_b \textit{gtenv}'}$$

Each definition is terminated by '.'; we therefore ensure that all clauses are evaluated:

$$\frac{\langle env, defs, atoms, g \rangle \vdash \langle \textit{gtenv}, f\ vs, \textit{clauses} \rangle \Downarrow_c \textit{gtenv}'}{\langle env, defs, atoms, g \rangle \vdash \langle \textit{gtenv}, f\ vs, \textit{clause. clauses} \rangle \Downarrow_c \textit{gtenv}'}$$

The main work happens in the case where we find a definition with a head $f\ Vs$ of the same name. The formal parameter list Vs is a list of variable names of the same length as vs. To illustrate, assume we are evaluating the underlined f in

$$\cdots \leftarrow h(X), \underline{f(\textit{concl}, X, Y, ?x)}, \cdots .$$

As a result of evaluating h, X may be bound to some value e_1 in the goal type environment. Furthermore, assume that *concl* is e_2 and $?x$ is bound to e_3 in the environment. When a definition of the same name, such as

$$f(A, B, C, D) \leftarrow \textit{body}.$$

is found, then the formal parameters must be instantiated. In this case it should generate an environment $\{A \mapsto e_2, B \mapsto e_1, D \mapsto e_3\}$. Note that these are *constraints* of the variables; as Y is not bound it is unconstrained, and is therefore not included. To achieve this instantiation we first introduce the partial function *lookup*:

$$lookup(goal, env, gtenv, v) = \begin{cases} get\_name(goal, v) & \textbf{if } v \in name \\ env(v) & \textbf{if } v \in \textbf{dom}(env) \\ gtenv(v) & \textbf{if } v \in \textbf{dom}(gtenv) \end{cases}$$

This function looks up values of names or variables when present in one of the environments. Note that *concl* and *hyps* are examples of names, and *get_name* will in those cases return the underlying conclusion (a term) or list of hypothesis (list of terms), respectively. We then define a function that is used to apply the instantiations:

$$\begin{array}{l} inst\_gtenv(goal, env, gtenv, [V_1, \cdots, V_n], [v_i, \cdots, v_n]) := \\ \quad \{V_i \mapsto lookup(goal, env, gtenv, v_i) | (goal, env, gtenv, v_i) \in dom(lookup)\} \end{array}$$

Returning to our example, the *body* of f is then evaluated, starting with the initial environment $\{A \mapsto e_2, B \mapsto e_1, D \mapsto e_3\}$ generated. A result of this evaluation is a new environment. Let's assume that this binds C and a new variable F: $\{A \mapsto e_2, B \mapsto e_1, D \mapsto e_3, C \mapsto e_4, F \mapsto e_5\}$. The clause should return an environment with only the variables in the actual parameters bound. In this case, the call made was $f(concl, X, Y, ?x)$, meaning only X and Y should be in the domain – which corresponds to variables B and C: $\{X \mapsto e_1, Y \mapsto e_4\}$. This functionality is handled by the *res_gtenv* function:

$$\begin{array}{c} res\_gtenv(gtenv, [V_1, \cdots, V_n], [v_i, \cdots, v_n]) := \\ \{v_i \mapsto gtenv(V_i) | v_i \in var \wedge V_i \in dom(gtenv)\}. \end{array}$$

As a result, the derivation rule for evaluating a single goal type schema becomes:

$$\frac{\begin{array}{c} length(vs) = length(Vs) \qquad gtenv_0 = inst\_gtenv(g, env, gtenv, Vs, vs) \\ \langle env, defs, atoms, g\rangle \vdash \langle gtenv_0, body\rangle \Downarrow_b gtenv_1 \\ gtenv' = res\_gtenv(g, gtenv_1, Vs, vs) \end{array}}{\langle env, defs, atoms, g\rangle \vdash \langle gtenv, f\ vs, f\ \ Vs \leftarrow body.\ clauses\rangle \Downarrow_c gtenv'}$$

with a special case when f is the last definition:

$$\frac{\begin{array}{c} length(vs) = length(Vs) \qquad gtenv_0 = inst\_gtenv(g, env, gtenv, Vs, vs) \\ \langle env, defs, atoms, g\rangle \vdash \langle gtenv_0, body\rangle \Downarrow_b gtenv_1 \\ gtenv' = res\_gtenv(g, gtenv_1, Vs, vs) \end{array}}{\langle env, defs, atoms, g\rangle \vdash \langle gtenv, f\ vs, f\ \ Vs \leftarrow body.\rangle \Downarrow_c gtenv'}$$

Another option is that f is an atomic goal type, which is then applied to generate the new environment:

$$\frac{f \in dom(atoms) \qquad gtenv' \in atoms(f)\ gtenv\ (v_1, \cdots, v_n)\ g}{\langle env, defs, atoms, g\rangle \vdash \langle gtenv, f\ (v_1, \cdots, v_n)\rangle \Downarrow_b gtenv'}$$

We have already mentioned *any* and $top\_symbol(X, Y)$ which are both atomic goal types. Other generic atomic goal types include:

—$trm\_var(X)$ holds if term X is a variable.
—$member(XS, X)$ holds if X is a member of list XS.
—$eq\_trm(X, Y)$ holds if the terms X and Y are syntactically equal (α-equivalence).

These can for example be used in a schema to check if a given term is the same as a hypothesis or the conclusion:

$$\begin{aligned} is\_goal(X) &\leftarrow eq\_trm(concl, X). \\ has\_hyp(X) &\leftarrow member(hyps, Y), eq\_trm(X, Y). \end{aligned}$$

We can also use *member* to define the above $has\_top\_symbol(X, Y)$:

$$has\_top\_symbol(X, Y) \leftarrow member(X, Z), top\_symbol(Z, Y).$$

A third case is negation. This case will behave as an identity function on the environment, if the non-negated version fails:

$$\frac{\neg\big(\exists\ gtenv'.\langle env, defs, atoms, g\rangle \vdash \langle gtenv, f\ (v_1, \cdots, v_n)\rangle \Downarrow_b gtenv'\big)}{\langle env, defs, atoms, g\rangle \vdash \langle gtenv, !f\ (v_1, \cdots, v_n)\rangle \Downarrow_b gtenv}$$

The final case is the evaluation of the body of a goal type of multiple clauses, i.e. the case: 'C, GTS'. Here, evaluation is sequential: first C is evaluated by $\Downarrow_c$, then the rest GTS is evaluated recursively by $\Downarrow_b$. However, the goal type environments cannot just be passed sequentially as the following example illustrates. Consider:

$$p(X, Y, Z) \leftarrow a(X, Z), b(Y), c(Z).$$

Assume we have the following application: $p(t_1, t_2, A)$. Here, the initial environment will be

$$\{X \mapsto t_1, Y \mapsto t_2\}$$

However, when applying a this is restricted to X and Z, so a will only return an environment with X and Z in. If we use this directly the Y binding is lost.

Instead, the environment is *updated* with the new values. Note that the constraints are checked in each element – thus it is safe to override. Finally, the two environments are combined, where the latter overrides[11] the former:

$$\frac{\begin{array}{c}\langle env, defs, atoms, g\rangle \vdash \langle gtenv, c_1, defs\rangle \Downarrow_c gtenv'' \\ \langle env, defs, atoms, g\rangle \vdash \langle gtenv\ \dagger\ gtenv'', (c_1, c_2)\rangle \Downarrow_b gtenv''' \\ gtenv' = (gtenv\ \dagger\ gtenv'')\ \dagger\ gtenv'''\end{array}}{\langle env, defs, atoms, g\rangle \vdash \langle gtenv, (c_1, c_2)\rangle \Downarrow_b gtenv'}$$

As the example above illustrates, the environment following b only contains Y. Thus X and Z are added, whilst X and Z have to be added after evaluating c. This completes the evaluation semantics of goal types.

To illustrate more complex usage of goal types, we provide an example from ongoing work by Farquhar and others on machine learning PSGraphs and goal types using a technique called *meta interpretive learning* [19]. Here, we provide low-level operations on terms and, using a small set of examples, learn suitable goal type definitions from them.

[11] $A\ \dagger B$ denotes that B overrides A

To achieve this, we introduce an atomic goal type

$$dest\_trm(X, L, R),$$

which holds if term X is "destructed" into its left L and right R sub-terms, when such exists[12] For example $dest\_trm(\ulcorner f\ x\ y \urcorner, \ulcorner f\ x \urcorner, \ulcorner y \urcorner)$ holds, meaning that $\ulcorner f\ x \urcorner$ is the left sub-term and $\ulcorner y \urcorner$ is right sub-term of $\ulcorner f\ x\ y \urcorner$. By using *dest_trm*, we can define *left* and *right* as:

$$\begin{aligned} left(X, L) &\leftarrow dest\_trm(X, L, \_). \\ right(X, R) &\leftarrow dest\_trm(X, \_, R). \end{aligned}$$

Next, we introduce another atomic goal type

$$const(X, C)$$

which holds if and only if term X is a the constant (name) C. For example, $const(\ulcorner f \urcorner, f)$ holds. Instead of treating *top_symbol(T,Y)* as an atomic goal type we can define it using these more primitive atomic goal types and recursion:

$$\begin{aligned} top\_symbol(T, Y) &\leftarrow const(T, Y). \\ top\_symbol(T, Y) &\leftarrow left(T, Z), top\_symbol(Z, Y). \end{aligned}$$

This goal type will traverse the left side of the term until the end is reached and check if this is the correct constant (or bind it to variable Y). Note that it will not work if the top-level function is higher-order (i.e. a lambda abstraction). We can also define a function that checks both the left and right side of an application. This amounts to checking if a symbol is present at any place of the term[13]:

$$\begin{aligned} has\_symbol(T, Y) &\leftarrow const(T, Y). \\ has\_symbol(T, Y) &\leftarrow right(T, Z), has\_symbol(Z, Y). \\ has\_symbol(T, Y) &\leftarrow left(T, Z), has\_symbol(Z, Y). \end{aligned}$$

These examples show that, as with PSGraph, we can work with the goal types at different levels of abstraction. This is illustrated by treating *top_symbol* as an atomic goal type or by defining it in the language in terms of lower-level more primitive atomic goal types.

3.3 Graphical breakpoints

There are two ways to apply a PSGraph to a goal in Tinker: (1) in the *automatic mode* it is applied as a black box and all you see is the final subgoals on the output wires; (2) in the *interactive mode* the user can step through and guide the proof of the goal. When debugging a large proof, such as our current work with D-RisQ's tactic [43], one often wants to combine these modes: one would like to use an automatic/black-box execution until the problematic part of the proof strategy is reached, and at that point enter an interactive mode where the user can step through the proof.

[12] Applications may also instantiate variables. For example, $dest\_trm(\ulcorner f\ x\ y \urcorner, V_1, V_2)$ will instantiate V_1 to $\ulcorner f\ x \urcorner$ and V_2 to $\ulcorner y \urcorner$, if they are not bound in the goal type environment.

[13] For simplicity, this definition does not work in presence of binders (lambda abstractions), but can easily be extended to support this with a new atomic goal type.

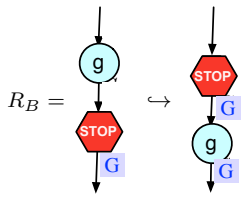


Fig. 10. Breakpoint rule

This is essentially how *breakpoints* of modern IDEs work: the user inserts a breakpoint in the program text, and the debugger will execute the code until the breakpoint is reached. At that point the user can manually step through the code. Inspired by this idea for debugging programs, we extend PSGraph with a new special *breakpoint node*, which can be seen in Fig. 10.

We also introduce a third mode called *debug mode*. The intuition behind this mode is to achieve exactly the requirement above: the graph is executed as in automatic mode until it cannot execute any further, either because it has successfully terminated or because the goals are followed by a debug node. In order to keep the semantics of PSGraph, we only need to update the termination condition for the debug mode:

Definition (Termination in debug mode). *A graph has terminated in* debug mode*, if for all goals g of the graph, g is either on a graph output wire or it is wired to another goal, or is wired to a debug node.*

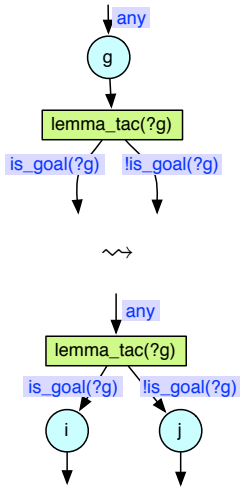
If the graph has successfully terminated in debug mode, it will enter interactive mode and the user can step through the graph manually. In this case, goals needs to be able to "step over" breakpoints, which is achieved by adding the rule R_B from Fig. 10 to the ruleset R_{eval} (Fig. 3) when we are in *interactive* and *automatic* modes, whilst omitting it from R_{eval} in *debug* mode.

This very small extension turns out to be a very powerful aid for debugging PSGraphs. We will see it in action in the case studies in §4.

3.4 A logging mechanism

Another recent extension, which as we will show later has been very useful in our case studies, is a *logging* mechanism. Introducing such a mechanism is an engineering problem rather than a scientific one, but as logging forms part of the user experience, it merits a brief discussion here.

To illustrate logging, consider the example in Fig. 11 (left), where *lemma_tac* applies the cut rule with the term bound in $?g$. The goal types $is\_goal(?g)$ and $is\_not\_goal(?g)$ check if the goal is or is not the same as $?g$ respectively. The logging mechanism will then print the logging messages as shown in Fig. 11 (right). First, it prints the information about the environment of goal g, which says it has a variable $?g$ bound. The next two lines shows the open goals afterwards, which are i and j. It then displays the results from evaluating the goal types: First we see that i success for the wire labelled by $is\_goal(?g)$. As all possible combinations will be generated, it also checks if i succeeds for the other wire, which fails. It then does the same for j, which will only succeeds for $is\_not\_goal(?g)$. The final line states that one branch was generated, with i and j on separate wires.




```
> ENV_DATA :
   g: E_Trm (...)
> GOAL : Open goals
   [Goal i] ...
   [Goal j] ...
> GOALTYPE : evaluating is_goal(?g) with
 pnode i env g => E_Trm(...)
> SUCCESS
> GOALTYPE : evaluating !is_goal(?g) with
 pnode i env g => E_Trm(...)
> FAILURE
> GOALTYPE : evaluating is_goal(?g) with
 pnode j env g => E_Trm(...)
> FAILURE
> GOALTYPE : evaluating !is_goal(?g) with
 pnode j env g => E_Trm(...)
> SUCCESS
> EVAL : Branch(goals on the output edges):  | i | j |
```

Fig. 11. A logging example

Full logging of a complex strategy with many branches can be very verbose. Our logging mechanism allows the user to use the tags such as `ENV_DATA`, `GOAL` etc. seen in Fig. 11 to filter the types of message that are displayed.

4. CASE STUDIES

This section will address our hypothesis through three case studies. The first example re-engineers a tautology-proving tactic into PSGraph. We will express the high-level ideas behind the tactic in an abstract way and then obtain an efficient implementation by a sequence of refactorings adding goal types to direct the proof search. Tinker's debugging capabilities are utilised to find and correct mistakes in the encoding. The second example looks at a set of *ad hoc* domain-specific tactics developed to finesse the proof of a lemma forming part of the proof of security of a database system. We will see how to use PSGraph to represent proof patterns involving tacticals (tactic combinators). The final case study considers a decision procedure for problems such as proving continuity of real-valued functions. We will see how PSGraph can be used to express complex recursive rewriting strategies.

A case-study approach for evaluation was chosen as the work is exploratory and improvement-driven. The three case studies have different, yet relevant, challenges and thus provide us with necessary armoury for larger scale problems found in industrial settings. They also enables analyses of PSGraph from different aspects, which is known as *triangulation* in software engineering [57]. We have deliberately addressed unfamiliar problems, as opposed to types of problems that we know that PSGraph will excel for. For reasons discussed in §5, our analysis is qualitative in nature.

4.1 A tautology tactic for propositional logic

For the purposes of this section, a tautology is defined to be a substitution instance of any formula χ formed from boolean variables and the boolean constants T and

F using the connectives $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \mathit{if\_then\_else\_}$, such that χ evaluates to T under any substitution of the constants T or F for its propositional variables. We will describe the design and implementation of a tactic that takes a goal which we assume (for simplicity) has no assumptions: $? \vdash \phi$ and will prove any such goal where ϕ is a tautology.

The decision procedure underlying the tautology tactic transforms its goal to a set S of subgoals:

$$\Gamma_1 \ ?\vdash \ t_1, \ ..., \ \Gamma_n \ ?\vdash \ t_n$$

where the Γ_i and the t_i comprise only propositional literals, i.e., atoms or negated atoms(but not $\neg T$ or $\neg F$). The transformation ensures that S is is logically equivalent equivalent to the original goal when viewed as a conjunction of implications. The original goal is then a tautology iff each of the subgoals in S has one of the following forms (which we refer to below as *structural tautological forms*):

$$\Gamma, \ t, \ \neg t \ ?\vdash \ u$$
$$\Gamma, \ t \ ?\vdash \ t$$
$$\Gamma, \ F \ ?\vdash \ u$$
$$\Gamma \ ?\vdash \ T.$$

The implementation will realise these transformations as tactics and will apply a tactic that will recognise and discharge structural tautological subgoals as they are created. Realising the decision procedure using tactics in this way converts it from an algorithm that merely recognises tautologies into an algorithm that finds a proof.

The tactic that implements the decision procedure uses two rewrite systems. The rewrite systems are defined by theorems giving universally quantified bi-implications which are instantiated as appropriate and used as left-to-right rewrite rules. The first rewrite system is applied to the conclusions of subgoals:

$$\vdash \forall \ a\bullet \ \neg \ \neg \ a \Leftrightarrow a$$
$$\vdash \forall \ a \ b\bullet \ \neg \ (a \wedge b) \Leftrightarrow \neg \ a \vee \neg \ b$$
$$\vdash \forall \ a \ b\bullet \ \neg \ (a \vee b) \Leftrightarrow \neg \ a \wedge \neg \ b$$
$$\vdash \forall \ a \ b\bullet \ \neg \ (a \Rightarrow b) \Leftrightarrow a \wedge \neg \ b$$
$$\vdash \forall \ a \ b\bullet \ \neg \ (a \Leftrightarrow b) \Leftrightarrow a \wedge \neg \ b \vee b \wedge \neg \ a$$
$$\vdash \neg \ T \Leftrightarrow F$$
$$\vdash \neg \ F \Leftrightarrow T$$
$$\vdash \forall \ a \ b \ c\bullet \ \neg \ (\mathit{if} \ a \ \mathit{then} \ b \ \mathit{else} \ c) \Leftrightarrow (\mathit{if} \ a \ \mathit{then} \ \neg \ b \ \mathit{else} \ \neg \ c)$$
$$\vdash \forall \ a \ b\bullet \ (a \Leftrightarrow b) \Leftrightarrow (a \Rightarrow b) \wedge (b \Rightarrow a)$$
$$\vdash \forall \ a \ t1 \ t2\bullet \ (\mathit{if} \ a \ \mathit{then} \ t1 \ \mathit{else} \ t2) \Leftrightarrow (a \Rightarrow t1) \wedge (\neg \ a \Rightarrow t2)$$
$$\vdash \forall \ a \ b\bullet \ a \vee \neg \ b \Leftrightarrow b \Rightarrow a$$
$$\vdash \forall \ a \ b\bullet \ \neg \ a \vee b \Leftrightarrow a \Rightarrow b$$
$$\vdash \forall \ a \ b\bullet \ a \vee b \Leftrightarrow \neg \ a \Rightarrow b$$

The following ProofPower idiom implements the above rewrite system:

```
val taut_strip_concl_conv : CONV = (
        eqn_cxt_conv(
        map thm_eqn_cxt
        [¬_¬_thm, ¬_∧_thm, ¬_∨_thm, ¬_⇒_thm,
         ¬_⇔_thm, ¬_t_thm, ¬_f_thm, ¬_if_thm,
        ⇔_thm, local_if_thm,
        a_∨_¬b_thm, ¬a_∨_b_thm, a_∨_b_thm]));
```

Here $\neg\_\neg\_thm$, $\neg\_ \wedge \_thm$ etc. name the theorems of the rewrite system in the order given above. Repeated application of these rewrite rules will transform the conclusion of a subgoal into either a propositional literal or a conjunction or an implication. If the conclusion is a propositional literal, the tactic will test whether the subgoal has one of the structural tautological forms, discharging the subgoal if it passes the test and reporting an error if it fails. If the conclusion is a conjunction, the subgoal splits into two subgoals, one for each conjunct. If the conclusion is an implication say $\phi \Rightarrow \psi$, the subgoal will reduce to a set of subgoals obtained from the original subgoal by adding certain literals to its assumptions. These literals are obtained by "stripping" the logical connectives out of the antecedent ϕ while making case splits as appropriate.

The following list of functions captures the processing described above but defers the stripping of new assumptions to a parameter that is a function of type

```
THM_TACTIC = THM -> TACTIC,
```

(Functions of this type are referred to as *theorem continuations* and play an important role in the traditional approach to programming LCF style systems [52].) For a conclusion of the form $\phi \Rightarrow \psi$, the tactical $\Rightarrow\_T$ will carry out the stripping process by passing the theorem $\phi \vdash \phi$ representing the new assumption to the parameter function. For the cases other than implications, this parameter is ignored.

```
val taut_strip_concl_ts : (THM_TACTIC -> TACTIC) list = [
        fn _ => ∧_tac,
        ⇒_T,
        fn _ => t_tac,
        fn _ => conv_tac taut_strip_concl_conv,
        fn _ => concl_in_asms_tac];
```

Stripping the antecedent ϕ of an implication $\phi \Rightarrow \psi$ into the assumptions is dual to the processing of a conclusion. If ϕ is not a conjunction or a disjunction, it is rewritten using a second rewrite system. This second system is like the first but with the rules for disjunctions replaced by the following rule for implications:

```
⊢ ∀ a b• a ⇒ b ⇔ ¬ a ∨ b
```

```
val taut_strip_thm_conv : CONV = (
        eqn_cxt_conv(
        map thm_eqn_cxt
        [¬_¬_thm, ¬_∧_thm, ¬_∨_thm, ¬_⇒_thm,
         ¬_⇔_thm, ¬_t_thm, ¬_f_thm, ¬_if_thm,
        ⇒_thm, ⇔_thm, local_if_thm]));
```

The new subgoals derived by stripping ϕ into the assumptions are then produced by iterating around the following list of functions: if ϕ is a conjunction, $\phi_1 \wedge \phi_2$, we strip ϕ_1 and ϕ_2 into the assumptions separately; if ϕ is disjunction, $\phi_1 \vee \phi_2$, we get two subgoals, one with ϕ_1 stripped into its assumptions and one with ϕ_2 stripped into its assumptions; otherwise we attempt to apply the rewrite rules:

```
val taut_strip_thm_thens : THM_TACTICAL list = [
        ∧_THEN,
        ∨_THEN,
        CONV_THEN taut_strip_thm_conv];
```

Here $\wedge\_THEN$ and $\vee\_THEN$ are operators on theorem continuations that perform one logical transformation and pass the theorems representing the result on to their operands. Operators like this provide a powerful continuation-passing style for programming tactics. This style was introduced and popularised by Paulson [52] and widely adopted by developers of tactics in LCF-style systems.

The following tactic implements a single step in the above process as determined by the principal connective of the conclusion of the goal. The expression beginning *REPEAT_TTCL* is the theorem continuation parameter for $\Rightarrow\_T$ mentioned above. It uses *taut_strip_thm_thens* to strip a new assumption into atoms and then uses the tactic *check_asm_tac* to add these assumptions to the resulting subgoals while checking for and discharging subgoals having one of the structural tautological forms.

```
val taut_strip_tac : TACTIC = (
        FIRST
        (map(fn t => t(REPEAT_TTCL (FIRST_TTCL taut_strip_thm_thens)
                       check_asm_tac))
             taut_strip_concl_ts));
```

Here *REPEAT_TTCL* and *FIRST_TTCL* are combinators on theorem continuations that provide repetition until failure and selection of the first non-failing theorem continuation from a list.

For an example of *taut_strip_tac* in action, let us see how it will work given Peirce's law: $((a \Rightarrow b) \Rightarrow a) \Rightarrow a$. It will actually prove this immediately by stripping the antecedent $(a \Rightarrow b) \Rightarrow a$ into the assumptions of a goal with conclusion a. The new assumption will first be rewritten in the form $\neg(a \Rightarrow b) \vee a$ resulting in a case split into two goals, one with assumption $\neg(a \Rightarrow b)$ and one with assumption a. The assumption $\neg(a \Rightarrow b)$ will be rewritten as $a \wedge \neg b$ which will be stripped into two new assumption literals, a and $\neg b$. In both cases, the resulting subgoal is a structural tautology that will be discharged by *check_asm_tac*.

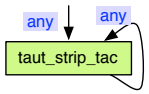


Fig. 12. *simp_taut_tac* version 1

The tautology tactic then simply repeats the single step tactic until there are no subgoals left or until no further progress can be made, in which case it raises an exception.

```
val simple_taut_tac : TACTIC = (fn gl =>
      case REPEAT taut_strip_tac gl of
            done as ([], _) => done
      |     _ => fail "simple_taut_tac" 28121 []);
```

(The number 28121 here is an error code identifying a message reporting that the conclusion of the goal is not a tautology.)

Failure-driven higher-order functional programming using combinators to control iteration and sequencing has proved very successful in programming LCF-style systems. Here it enables us to code a rather complex recursion scheme in a compact way that does reveal the structure of the algorithm to those familiar with the approach. Although we have spent several pages here describing the tautology tactic, the actual source code we have presented is only 32 lines of which all but 9 do little more than set up tables. However, we agree with Paulson, who concedes that while higher-order functions provide good control and efficiency, they can be hard to understand [52].

Looking at even a simple example of the higher-order programming style bring several questions to mind: is the high-level proof plan visible to a non-expert looking at the implementation? How easy would it be to locate a mistake in the code if it failed to prove a tautology? How would we go about refactoring the code? In the rest of this section, we will illustrate how to encode *simple_taut_tac* in PSGraph using the Tinker system. We will show how to support developing a correct and optimised PSGraph implementation through a set of refactoring and analysis supported by the Tinker framework. For the most readable version of this tactic, we refer to the final version (Fig. 18)[14].

4.1.1 *Version 1: A generic PSGraph of the tautology tactic.* *simple_taut_tac* repeats *taut_strip_tac* until it is no longer applicable; if all subgoals are discharged at this point then the tactic succeeds, and it fails otherwise. Fig. 12 implements this tactic at a very high level of atomicity where *taut_strip_tac* is treated as an atomic tactic.

Graphically, repetition is simply represented as a feedback loop. By making this feedback loop the only output wire we achieve the same termination semantics as *simple_taut_tac*. This can be justified as follows: if *taut_strip_tac* fails on any subgoal then the overall tactic will fail: in *simple_taut_tac* this means that the *REPEAT* combinator will terminate with a subgoals which result in failure. If the tactic produces subgoals then *taut_strip_tac* is re-applied, as is the case for the

[14]It may be easier to understand the example by working backwards from this final version.

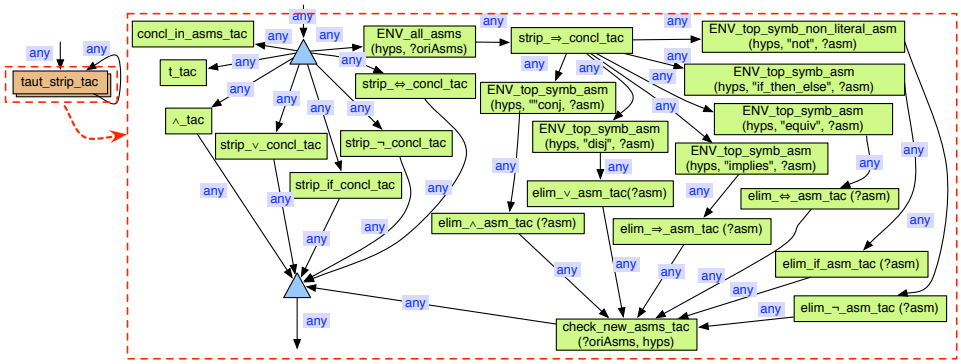


Fig. 13. Version 2: Flat and parallel tactic application.

REPEAT combinator. Finally, if there are no more subgoals, then the PSGraph will successfully terminate; this is also the success case for PSGraph.

4.1.2 *Version 2: From sequential to parallel tactic application.* The example of Fig. 12 does not show sufficient details to understand how *simple_taut_tac* works. This require a further "unfolding" of *taut_strip_tac* into a graph. Fig. 13 shows the same tactic as a graph tactic and its subgraph.

This is achieved by "unpacking" all the tacticals, and represent each of the components as a tactic, with some minor modifications. To illustrate, the left part of the subgraph in Fig. 13 corresponds to the *taut_strip_concl_ts* tactic, with the conversions in the list of *taut_strip_concl_conv* represented by the 4 atomic tactics starting with 'strip'.

The right part of the subgraph corresponds to the work conducted on the hypothesis by *taut_strip_thm_thens* when an implication introduction rule is applied. It first uses the *ENV_all_asms* environment tactic to store all hypothesis in a variable ?*oriAsms*, before applying the introduction rule. For each propositional combinators, there is then a case adapted from *taut_strip_thm_thens* and the conversions in *taut_strip_thm_conv*. In most cases, they follow the pattern illustrated in Fig. 9 (§3.1), where the hypothesis is first bound by one environment tactic and then the elimination rule is applied. At the end of this branch of the graph, *check_new_asms_tac* will get the lists of new hypothesis by comparing the current hypothesis (*hyps*) with the hypothesis on entry (?*oriAsms*) to this part of the graph.

A conceptual difference between Fig. 13 and the *simple_taut_tac* ProofPower tactic is that in PSGraph we no longer need to enforce a sequential order; if two or more tactics are mutually independent, we can put them next to each other using identity tactics as necessary to split inputs and merge outputs.

Note that each wire is labelled by `any`, meaning it will always succeed. This means that for a given subgoal generated by a given tactic, all possible output wires will be attempted in a separate branch of the search space. Thus, this graph can be seen as a generalisation of *taut_strip_tac* in the sense that it will succeed if *taut_strip_tac* succeeds (albeit it is not as efficient).

4.1.3 *Version 3: Modularising the graph through hierarchies.* PSGraph aims to support development of proof strategies that are easy understand, maintain and

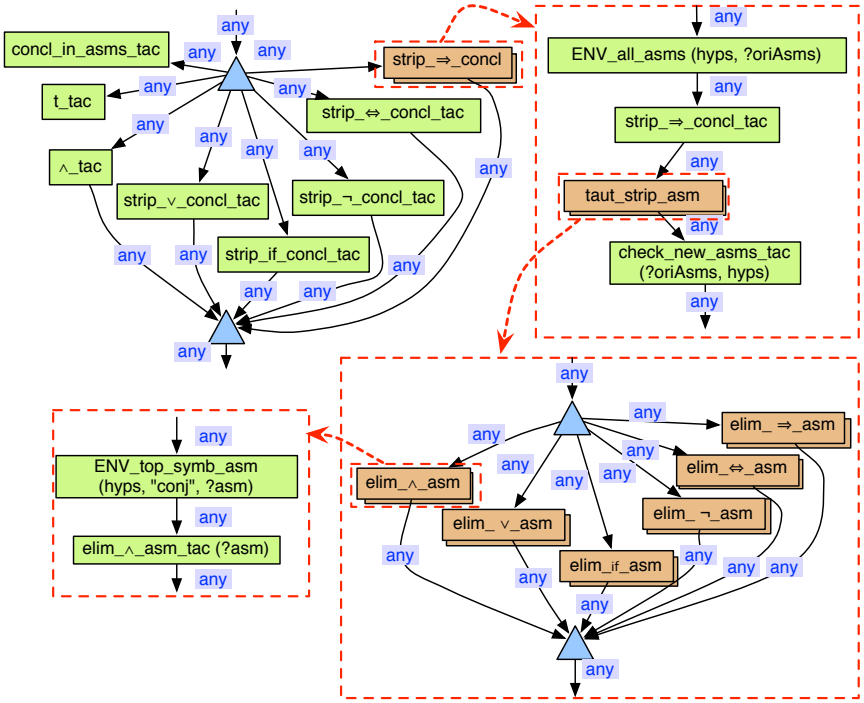


Fig. 14. *simp_taut_tac* version 3: with hierarchies

refactor. To achieve this it should be intuitive to see what the proof strategy is meant to do. Whilst the flat graph of Fig. 13 gives a detailed account of how the goals flow, it mixes high-level descriptive details of the proof strategy with low-level implementation details that are required to run it. It also "merges" different operations which are best to split, e.g. operations on the conclusion and operations on the hypothesis. This should be avoided when a more declarative and readable strategy is sought.

For tactic languages modularity is handled by sub-tactics, as is the case for *taut_strip_tac*. Within PSGraph such *modularity* is achieved through hierarchical graph tactics. Fig. 14 refactors the graph of Fig. 13 into a more modular graph.

The top-level graph (top-left) contains the atomic operations on the goal, but has refactored the case that handles implications (and following operations on the hypothesis) into a graph tactic called strip_⇒_concl. This is shown in the top-right corner of Fig. 14. Within strip_⇒_concl, the actual operations on the hypothesis are refactored into a graph tactic taut_strip_asm, which is comparable to the *taut_strip_thms_thens* tactic, shown on the bottom left corner of Fig. 14. Each case of this level corresponds to a propositional operator and is a nested graph tactic, where each of these follow the structure shown on the bottom right side. Here, an environment tactic first bind the operator and then the tactic is applied. The example illustrates the case for a conjunction, but the other cases are similar.

4.1.4 *Version 4: The use of goal types to explain and optimise the tactic.* The hierarchies help in exposing the high-level proof idea by hiding lower-level details

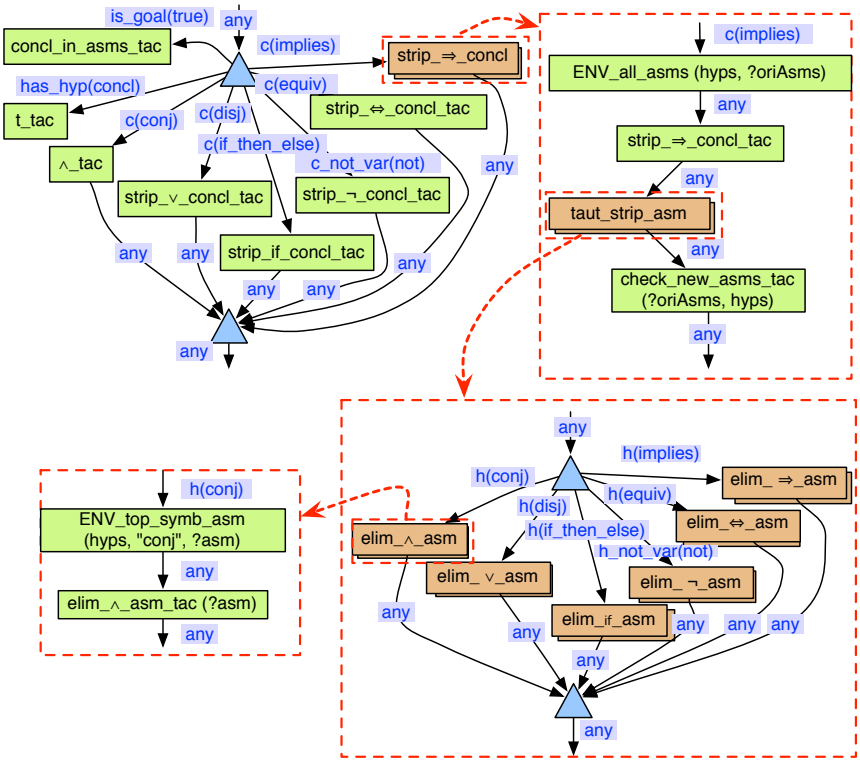


Fig. 15. *simp_taut_tac* version 4: with hierarchies and goal types

and "grouping" together sub-strategies, such as separating operations on hypothesis from operations on the conclusion. However, all wires are labelled by the any goal type, which always succeeds. This use of any has at least threes problems:

—*Explanation:* the proof strategy does not explain *why* a goal should choose a particular path. This is crucial in order to understand the proof strategy.

—*Evaluation:* the use of any means is that all paths are attempted, which is inefficient.

—*Debugging:* a side-effect of the evaluation is that it debugging becomes hard as:
  —there more (failed) branches in the search space to analyse;
  —it is not clear what the intention of a particular path is which makes it hard (and time consuming) to find the "correct" branch;
  —the error may manifest itself at different place further down the "flow" of the strategy.

Developing goal types is one of the more challenging tasks of developing PSGraphs, and is also where development deviates most from standard tactic developments. In tactics we often end up trying one tactic first (e.g. if it is very quick or normally works) and if it fails we try something else. This is essentially what the tacticals in the list used by *taut_strip_tac* does. Although this is possible in PSGraph, it is better to think about *why* a particular tactic (or sub-strategy) should be applied.

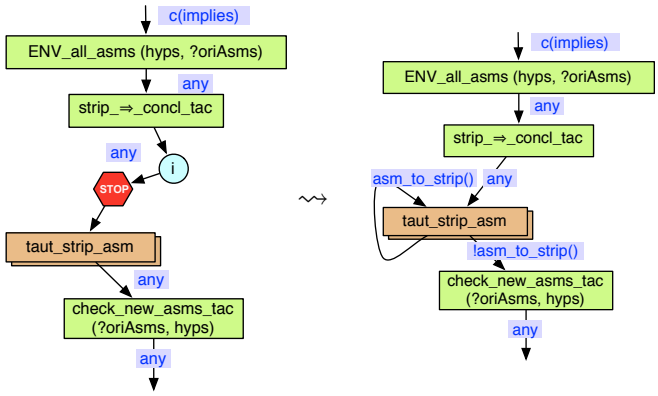


Fig. 16. Left: strip_⇒_concl (version 4) illustrating a bug with a breakpoint. Right: strip_⇒_concl (version 5) with bug fixed by feedback loop.

Moreover, when it fails it is hard to analyse where and why the failure happened. If the tactic/PSGraph also contains the "reason" for why a tactic is applied, in form of a goal type, then any failure is likely to show up at the right place and not several tactic applications later. This will create much more *maintainable* proof strategies; as we will illustrate below, it also becomes easier to analyse and patch a mistake in a proof strategy.

Fig. 15 updates the graph of Fig. 14 with goal types. Some of these goal types were introduced in §3.2, while we will introduce some new ones here. Firstly, recall the atomic goal type

$$dest\_trm(X, Y, Z)$$

from §3.2, which holds if term X is "destructed" into its left Y and right Z subterms. We use $c(X)$ and $h(X)$ as shorthand for the top symbol of the conclusion and a hypothesis, respectively:

$$\begin{aligned} c(X) &\leftarrow top\_symbol(concl, X). \\ h(X) &\leftarrow member(hyps, Z), top\_symbol(Z, X). \end{aligned}$$

The conversions applied to deal with negations by *strip_¬_concl_tac* (conclusion) and *elim_¬_asm* (hypothesis) require that the top level symbol is a negation, and the body is not just a variable (i.e. it is either compound or a constant). These properties are expressed by the goal types:

$$\begin{aligned} c\_not\_var() &\leftarrow c(not), dest\_trm(concl, \_, Z), !trm\_var(Z). \\ h\_not\_var() &\leftarrow member(hyps, Y), top\_symbol(Y, not), dest\_trm(Y, \_, Z), \\ &\quad\ !trm\_var(Z). \end{aligned}$$

With the goal types one can see in which cases a tactic should be applied, and evaluation will only try the branches where a goal satisfies the goal type.

4.1.5 *Version 5: Discovery and patching of a bug.* The proof strategy of Fig. 15 will succeed for a large set of propositional tautologies, such as:

GOAL

$(* \vdash *)\ulcorner A \wedge B \Rightarrow B \wedge A \urcorner$

However, it fails for the following (correct) goal:

GOAL

$(*\ ?\vdash\ *)\ulcorner A \wedge B \wedge C \Rightarrow C \wedge B \wedge A\urcorner$

As we have an idea where the problem is, we insert a breakpoint and automatically evaluate the strategy until the break point is reached[15]. This is shown in Fig. 16 (left). At this point the goal, labelled by i, has been simplified to:

GOAL

$(*\ \ 1\ *)\ulcorner A \wedge B \wedge C\urcorner$
$(*\ ?\vdash\ *)\ulcorner C \wedge B \wedge A\urcorner$

We can now step through the proof from that point in the nested *taut_strip_asm* graph tactic. The goal satisfies $h(conj)$, as the top level symbol of an hypothesis is a conjection, and correctly splits up the conjunction in the hypothesis *(* 1 *)*:

GOAL

$(*\ \ 1\ *)\ulcorner A\urcorner$
$(*\ \ 2\ *)\ulcorner B \wedge C\urcorner$
$(*\ ?\vdash\ *)\ulcorner C \wedge B \wedge A\urcorner$

At this point it will exit the graph tactic, and (via *check_new_asms_tac*) return to the top of the top-level graph. The problem is that one hypothesis *(* 2 *)* contains a conjunction, and the aim of the overall proof plan is that all connectives in the hypothesis should have been eliminated. One could still continue to run the proof, where it will eventually will have the goal:

GOAL

$(*\ \ 1\ *)\ulcorner A\urcorner$
$(*\ \ 2\ *)\ulcorner B \wedge C\urcorner$
$(*\ ?\vdash\ *)\ulcorner C\urcorner$

which will not satisfy the goal type (of the top level graph), and thus fail. The problem is that *taut_strip_asm* has to be repeated until there are no more connectives. This is reflected in the updated proof strategy shown in Fig. 16 (right). Here, we need a goal type to identify when there are more goals to be satisfied and label the loop with this:

$$\begin{aligned} asm\_to\_strip() &\leftarrow h(conj).\\ asm\_to\_strip() &\leftarrow h(disj).\\ asm\_to\_strip() &\leftarrow h(equiv).\\ asm\_to\_strip() &\leftarrow h(implies).\\ asm\_to\_strip() &\leftarrow h(if\_then\_else).\\ asm\_to\_strip() &\leftarrow h\_not\_literal(not). \end{aligned}$$

This is essentially a disjunction of all the possible symbols. The output wire, representing termination of the loop, is labelled by its negation: $!asm\_to\_strip()$. Note that if we had this goal type instead of *any* as output of *taut_strip_asm* in Fig. 15 (left), then the error would manifested itself at the correct place, which illustrates the importance of goal types. The above goal will now succeed.

[15] If we had no idea where the problem may have been we could just have evaluated the strategy from the beginning – the same approach as described in the rest of the section would still be applicable.

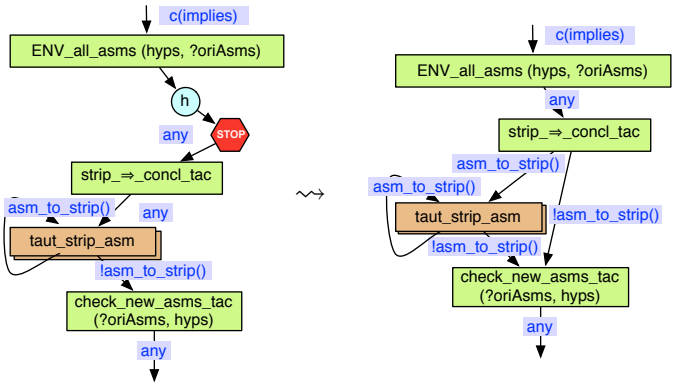


Fig. 17. Left: strip_⇒_concl (version 5) illustrating bug with goal node and breakpoint. Right: strip_⇒_concl (final version 6) with bug fixed by new wire.

4.1.6 *Version 6 & final version: Discovery and patching of another bug.* Next, we try to prove the following goal:

GOAL

$(*\ ?\vdash\ *)\ulcorner A \Rightarrow A \wedge A\urcorner$

Again, the tautology strategy fails. As with the previous case, one would suspect the issues is related to the hypothesis, thus we insert a breakpoint just before this part as shown in Fig. 17 (left). At this point, the goal is the same as the original goal. In the next step, *strip_* ⇒ *_concl_tac* is applied generating:

GOAL

$(*\ \ 1\ *)\ulcorner A\urcorner$
$(*\ ?\vdash\ *)\ulcorner A \wedge A\urcorner$

It will then enter the *taut_strip_asm* graph tactic, but it will then fail when stepping over the identity tactic. At this point, we can use the *logging mechanism* of Tinker, which gives the following message:

```
FAILURE : Fail to match any Loop for the output goal node:
          [Goal i : A |- A & A]
```

In this case, the only assumption is

ASSUMPTIONS

$(*\ \ 1\ *)\ulcorner A\urcorner$

which does not have any logical connectives and should therefore not be further simplified. The problem is that we have forgotten to bypass *taut_strip_asm* when there are no assumptions to simplify. To rectify the strategy, this missing case is added, using the *asm_to_strip*() goal type to separate the two cases. This is shown in Fig. 17 (right)

This completes the development of *simple_taut_tac* as a PSGraph. A complete version of the tautology tactic is shown in Fig. 18. In this final version we have also added a goal type to the input of of the overall strategy to show which type of

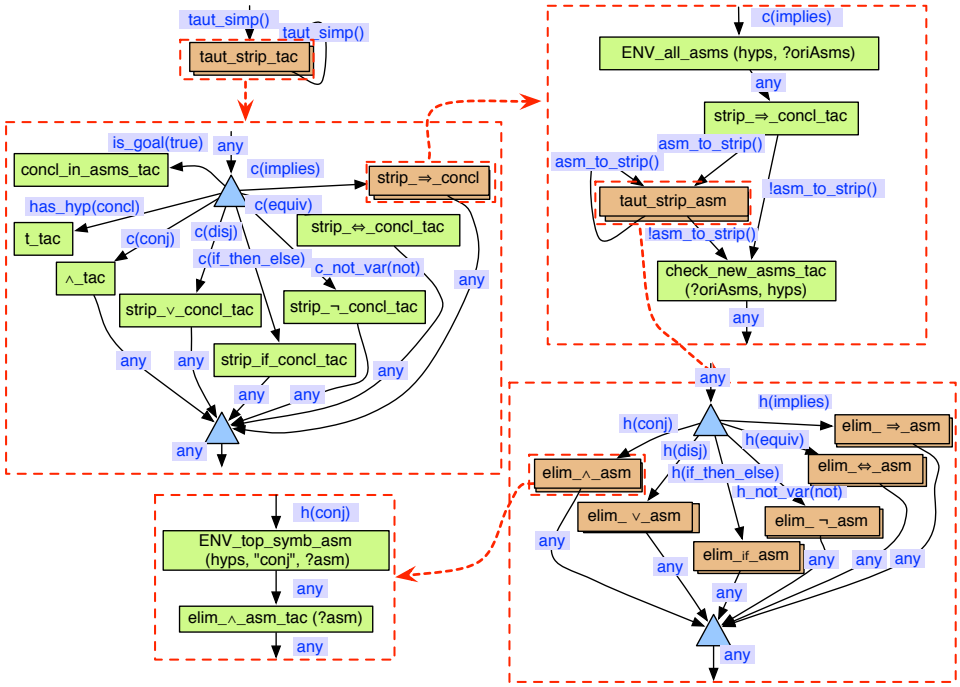


Fig. 18. *simp_taut_tac* completed version

goals it will work for:

$$
\begin{aligned}
taut\_simp() &\leftarrow is\_goal(true).\\
taut\_simp() &\leftarrow has\_hyp(concl).\\
taut\_simp() &\leftarrow has\_no\_hyp(concl), c(conj).\\
taut\_simp() &\leftarrow has\_no\_hyp(concl), c(disj).\\
taut\_simp() &\leftarrow has\_no\_hyp(concl), c(if\_then\_else).\\
taut\_simp() &\leftarrow has\_no\_hyp(concl), c(equiv).\\
taut\_simp() &\leftarrow has\_no\_hyp(concl), c(implies).\\
taut\_simp() &\leftarrow has\_no\_hyp(concl), c\_not\_literal(not).
\end{aligned}
$$

4.1.7 *Discussion.* A simple and instructive example has been used to illustrate many features of PSGraph and Tinker, which can be contrasted to the existing sentential encoding of *simple_taut_tac* in ML. One of the key advantages is that a non-expert can read the overall proof plan directly from the graph. Compare that for example Fig. 18 (starting from top-left) with the use of the *REPEAT_TTCL* tactical used in *taut_strip_tac*. Without a deep understanding of underlying semantics of this (and other) tactical, it is very hard to see what the tactic does. For example, it matters if *REPEAT_TTCL* is applied 1 or more times or 0 or more times. Our original version only applied it a single time; the second version 1 or more times; and the final version got it right and applied it 0 or more times. Through the debugging features of Tinker and PSGraph, this was fairly easy to

find[16]. To find a similar mistake in the original code would not be as easy, and would very likely require us to tear the tactic apart so that we can step through the execution, which in itself is not always an easy task. For a subset of ML, the *Tactician* tool for HOL light [2] can automate such tearing apart of tacticals, but in most cases it has to be done manually. To summarise, while short "one liners", such as *taut_strip_tac*, are elegant, they are not necessarily that easy to understand. Anecdotally, getting the ML code right is hard even for an expert – then what about novel users and non-experts? Would you show the one-liner to your (non-technical) line manager or would you draw up a high-level diagram?

We have also found other advantages of the graphical representation. One example is that PSGraph allows us to delay decision on control flow, and make it efficient later (by adding goal types). This is not possible in ML, where we must give the order straightaway. Secondly, hierarchies enable hiding of low-level (implementation) details: we can use the top-level graph(s) to show the high-level (often declarative) proof ides, while the graph tactic contains implementation details. Consider for example how *elim_∧_asm* hides how we use an environment tactic to apply conjunction elimination in Fig. 18. Thirdly, meta-level properties of the proof strategies can be read directly from the graph – which is not as obvious in the ML code. One example of this is the symmetry between the proof steps in the conclusion (top-left) and the hypothesis (bottom-right) of Fig. 18.

As a general *guidance*, picking sensible names for goal types and tactics is crucial if the PSGraph should act as an explanation of a proof strategy. It is also useful to try to combine parts that "belong" together in graph tactics. To work with PSGraph, one need to change to a more *declarative* way of thinking compared with the more *procedural* way of developing LCF tactics. It is important to think about *why* certain tactics should be applied, and encode this knowledge into the goal types. It is also important to be aware that there is still work required at the ML level: one has to develop atomic tactics and atomic goal types, and also think about the arguments (if any) of them. Through goal types, PSGraph ensures *locality* when changing part of the proof strategy. You can safely assume that changes you make will only effect goals that satisfy the goal types leading to the sub-graph that is change: it will not have any impact on parallel sub-graphs.

This case study has highlighted a current *limitation* with respect to *parametrised graph tactics*. All the tactics within *taut_strip_asm* of Fig. 18 have the structure illustrated by *elim_∧_asm*. The only difference is that logical connective and tactic used. It would have been desirable to be able to make this an argument for a generic graph tactic, to avoid having to re-implement each version. The next example will illustrate how we can parametrise over tactic and goal type arguments, but not the actual tactics.

4.2 Domain specific tactics from the *Front End Filter* (FEF) project

The FEF Project [55] was an early application of ProofPower to verify the security properties of a multi-level secure database system called SWORD. In this section we will look at some very domain-specific tactics taken from the FEF proof scripts

[16]Note that these bugs where not artificial by any means: they were genuine mistakes we did during development and the graphical representation help to locate.

as an experiment in porting an existing application proof to PSGraph with a view to making future maintenance easier by presenting the proof at a higher level.

It would be inappropriate here to give a very detailed description of the four tactics we investigate. Such a description would be long and not very instructive and would not reflect the process by which the tactics came into existence, which was by interactive trial and error. The best way to get a feeling for this kind of process is by replaying the proofs interactively. See [55] for instructions for downloading the proof scripts. The ProofPower documents that are most relevant to the present paper are `fef032.doc` for specifications and `fef033.doc` for the proofs. In the present paper, we will set the relevant proof in context, which should give enough background to understand our transcription of the tactics into PSGraph and should help anyone who is interested in more detail to locate and work with the FEF documents.

The query language for the SWORD database was Secure SQL (SSQL) an extension of standard SQL whose semantics support a notion of security classification. Data in the database would be a assigned a classification drawn from some lattice. e.g., SECRET > COMPANY-RESTRICTED > COMPANY-IN-CONFIDENCE > UNCLASSIFIED. If c_1 and c_2 are elements of the lattice we say c_1 *dominates* c_2 if $c_1 \geq c_2$. Database users are assigned a security clearance which is also drawn from the lattice of classifications: a user cleared at classification c is only allowed to see data whose classification is dominated by c. SSQL was implemented via a preprocessor (referred to as the Front End Filter or FEF) that translates SSQL queries to queries in ordinary SQL on a database whose schema augmented the SSQL schema with security classification for each item of data. The translated query uses these classifications to prohibit information flows that would violate the security policy, e.g., by revealing SECRET data to a user who is only cleared to see information at COMPANY-IN-CONFIDENCE or UNCLASSIFIED.

The high-level security property for FEF requires that for every query q and every user u, if two states s_1 and s_2 differ only in respect of data that u is not cleared to see, then, when executed by u, q will deliver the same result in state s_1 as it doesn in state s_2. To achieve this, the semantics of SSQL label the result of any calculation with a security classification. If the label on the end result of a query is c, then the implementation will erase the information content of the result if the user is not cleared to see data of classification c. The formal specification of the SSQL semantics includes for each language construct both the derivation of the result and the derivation of the classification label from the values and classifications of the operands of the construct and of the database items it accesses. There is then a proof obligation to show that the semantics satisfies the high-level security property. Like conventional SQL, the syntax of SSQL queries involves a mutual recursion between table-expressions and value-expressions. These are given formal semantics as what we refer to as table computations and value computations. To prove that the semantics satisfies the high-level security property involves an induction over the syntax to prove a property OK_TC_d on table computations that can be used fairly directly to prove the high-level security property. OK_TC_d is parametrised by a security classification c and is defined as follows:

$$\begin{array}{lll}\forall c\ tc\bullet & tc \in OK\_TC_d\ c \Leftrightarrow & \\ & \forall tl_0\ tl_1\bullet & \\ & & Map\ (HideDerTable\ c)\ tl_0 = Map\ (HideDerTable\ c)\ tl_1 \\ & \wedge & \neg HideDerTable\ c\ (Snd(tc\ tl_0)) = HideDerTable\ c\ (Snd(tc\ tl_1)) \\ & \Rightarrow & \neg c\ dominates\ Fst(tc\ tl_0)\end{array}$$

Here we see that the table computation tc is a function with one argument: a list of tables. The result of the table computation is a pair whose first component is the classification label and whose second is the computed result. The function *HideDerTable* is parametrised by the classification c of a user and replaces all items in its operand that the user is not cleared to see by dummy values. The antecedent of the implication in the theorem therefore asserts that a value calculated by tc has revealed information about the operands that a user at classification c is not cleared to see. A table computation tc belongs to the set OK_TC_d $c$iff whenever the antecedent holds tc labels the return value with a classification that will prevent a user with classification c from seeing it.

To get the induction to go through, the property OK_TC_d needs to be strengthened by adding an additional property OK_TC_c(which ensures that the classification labels do not provide a covert channel) and we have to define analogous properties OK_VC_d and OK_TC_con the value computations (the definition of OK_VC_d is given below; see [55, `fef032.doc`] for the other definitions).

As we shall see in the example, these properties are actually repesented as sets and are parametrised by a security classification (and the induction proves that the SSQL table and value computations belong to the appropriate sets at every classification). In this paper, we are going to look at some tactics defined to complete the proof of a lemma about CASE-expressions that is needed in the induction. The goal is as follows:

$$\begin{array}{ll}?\vdash \forall c\ te\ cel\ ee\bullet & \\ & te \in OK\_VC_d\ c\ \wedge \\ & Elems\ (Map\ Fst\ cel) \subseteq OK\_VC_d\ c\ \wedge \\ & Elems\ (Map\ Snd\ cel) \subseteq OK\_VC_d\ c\ \wedge \\ & ee \in OK\_VC_d\ c \Rightarrow \\ & CaseVal\ c\ te\ cel\ ee \in OK\_VC_d\ c\end{array}$$

Here *CaseVal* is the constant that captures the semantics of the SSQL CASE-expression. As in SQL, this has the syntax:

```
CASE <te> WHEN <c1> THEN <e1> ... WHEN <cN> THEN <eN> ELSE <ee>
```

In the goal, c is the security classification of a user executing the query and *te*, *cel* and *ee* give the semantic values of the operands of the CASE-expression, with the WHEN/THEN pairs combined into a list of pairs *cel*. *Elems* is the function that maps a list to its set of elements and *Fst* and *Snd* are the projections. Hence the four conjuncts in the antecedent in the goal assert in turn that the semantics of the test expression `<te>`, the condition expressions `<c1>`...`<cN>`, the result expressions `<e1>`...`<eN>` and the else expression `<ee>` satisfy the OK_VC_d part of the inductive hypothesis. So our lemma asserts that the CASE-expression preserves this part of the inductive hypothesis.

The property OK_VC_dhas the following defining theorem:

$$
\begin{array}{lll}
\forall c\ vc\bullet & vc \in OK\_VC_d\ c \Leftrightarrow & \\
& \forall tl_0\ tl_1\ rl_0\ rl_1\ r_0\ r_1\bullet & \\
& \qquad Map\ (HideDerTable\ c)\ tl_0 = Map\ (HideDerTable\ c)\ tl_1 & \\
& \wedge \qquad Map\ (HideDerTableRow\ c)\ rl_0 = Map\ (HideDerTableRow\ c)\ rl_1 & \\
& \wedge \qquad HideDerTableRow\ c\ r_0 = HideDerTableRow\ c\ r_1 & \\
& \wedge \qquad \neg Snd(vc\ tl_0\ rl_0\ r_0) = Snd(vc\ tl_1\ rl_1\ r_1) & \\
& \Rightarrow \qquad \neg c\ dominates\ Fst(vc\ tl_0\ rl_0\ r_0) &
\end{array}
$$

Here we see that the value computation vc is a function with three arguments: a list of tables (needed for nested SELECTs), a list of table rows (needed for GROUPBY) and a table row (the row from which individual values are extracted by name in simple expressions). The result of the value computation is a pair whose first component is the classification label and whose second is the value of the expression. The hide functions set items in their operand that the user is not cleared to see to dummy values. So very like OK_TC_d discussed above, membership of $OK\_VC_d\ c$ asserts that if a value calculated by vc has revealed information about the operands that a user at classification c is not cleared to see then vc must label the value with a classification that will prevent that user seeing it.

A design goal of SSQL was to classify data as liberally as the security requirements and the desire for an efficient and maintainable implementation permitted. For the CASE-expression, the classification is that of the test-expression if the user is not cleared to see the test expression; if the user is not allowed to read one of the condition expressions that comes before the selected condition, then the classification is the classification of that condition expression (since the fact that that condition was not selected reveals information about the expression); otherwise the classification is the classification is that of the selected result expression. The main proof plan for the lemma is an induction on the list of WHEN/THEN pairs. The inductive step is the following goal:

$$
\begin{array}{l}
(*\ \ 5\ *)\ulcorner CaseVal\ c\ te\ cel\ ee \in OK\_VC_d\ c\urcorner \\
(*\ \ 4\ *)\ulcorner te \in OK\_VC_d\ c\urcorner \\
(*\ \ 3\ *)\ulcorner Elems\ (Map\ Fst\ (Cons\ x\ cel)) \subseteq OK\_VC_d\ c\urcorner \\
(*\ \ 2\ *)\ulcorner Elems\ (Map\ Snd\ (Cons\ x\ cel)) \subseteq OK\_VC_d\ c\urcorner \\
(*\ \ 1\ *)\ulcorner ee \in OK\_VC_d\ c\urcorner
\end{array}
$$

$$
(*\ ?\vdash\ *)\ulcorner CaseVal\ c\ te\ (Cons\ x\ cel)\ ee \in OK\_VC_d\ c\urcorner
$$

To make the presentation more readable, the calculation of the classification of a CASE-expression is defined separately from the calculation of the value of the expression. To make the inductive hypothesis (assumption 5) usable, these definitions have to be combined (in the lemma *CaseVal_lemma*) into a single primitive recursion over the WHEN/THEN list. Unfortunately, this combined definition involves 5 conditionals and when we expand the definition of OK_VC_d and do some normalisation, the goal ends up containing 15 conditionals: 5 in each expression corresponding to the 3 applications of vc in the definition OK_VC_d. The last few lines of the goal are as follows:

```
...
⇒ ¬ c dominates (
        if Snd (te tl_0 rl_0 r_0) = Snd (Fst x tl_0 rl_0 r_0)
        then
                if c dominates Fst (te tl_0 rl_0 r_0)
                        ∧ c dominates Fst (Fst x tl_0 rl_0 r_0)
                then Fst (Snd x tl_0 rl_0 r_0)
                else if ¬ c dominates Fst (te tl_0 rl_0 r_0)
                then Fst (te tl_0 rl_0 r_0)
                else Fst (Fst x tl_0 rl_0 r_0)
        else if c dominates Fst (te tl_0 rl_0 r_0)
                ∧ c dominates Fst (Fst x tl_0 rl_0 r_0)
        then Fst (CaseVal c te cel ee tl_0 rl_0 r_0)
        else if ¬ c dominates Fst (te tl_0 rl_0 r_0)
        then Fst (te tl_0 rl_0 r_0)
        else Fst (Fst x tl_0 rl_0 r_0))
```

A case split on the first 3 tests and some rewriting eliminates 6 of the 8 cases leaving 2 outstanding subgoals '4.1' and '4.2'. In each of these subgoals, a case split on what are now the first 3 tests and some rewriting simplifies away all the conditionals leaving 16 outstanding subgoals '4.1.1' ... '4.1.8' and '4.2.1' ... '4.2.8'. An interactive attack on '4.1.1' finds a combination of tactics that proves both it and '4.1.2', so we package this up into an *ad hoc* tactic *tac1* and try it on all of '4.1.1' ... '4.1.8' and find that it proves the first 4 of them leaving '4.1.5' to prove. Continuing in this way we end up with 4 very application-specific tactics each of which proves 4 of the 16 subgoals and with these we can complete the proof. The ML code of the 4 tactics is shown in Figure 19.

Representing these application-specific tactics in PSGraph prevents novel challenges. In the tautology example we were faced with mature production level code, implementing a clear underlying proof plan. In this example, we are faced with a set of *ad hoc* tactics that "did the job". An interesting problem is to understand the underlying proof idea, with the ultimate aim of providing a tactic that is robust to changes and can be re-used for similar proofs. In the remainder of this section we will make steps by showing a systematic and modular way of transferring tactics to PSGraph to improve understandability and maintainability.

4.2.1 *Encoding common proof patterns.* The first observation we make about the four *FEF* tactics is that of code repetition, which we can extract into separate proof patterns. Our first step is to develop these patterns in PSGraph; this will enable us to develop a library and re-use the patterns using the *library* functionality of Tinker (see §2.2.2) We have found three patters, which we call *LEMMA_THEN1*, *CASE_THENLIST* and *DROP_ASM*. These are highlighted in Fig. 19, which shows the ProofPower code of the tactics. Note that code is only included to show the repetition: we will not go into details of the actual tactics except for the encoding of the patterns.

4.2.1.1 *The* LEMMA_THEN1 *pattern.* The *LEMMA_THEN1* can be implemented as follows:

SML

```
val tac1 = REPEAT strip_tac THEN
  LEMMA_T ⌜Snd x ∈ OK_VC_d c⌝                                   LEMMA_THEN1
  (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])
  THEN1 PC_T1 "sets_ext" asm_prove_tac(map get_spec[⌜Elems⌝, ⌜Map⌝]);

val tac2 = REPEAT strip_tac THEN
  cases_tac⌜Snd (te tl_0 rl_0 r_0) = Snd (te tl_1 rl_1 r_1)⌝     CASE_THENLIST
  THEN_LIST [
  asm_ante_tac ⌜¬ Snd (te tl_1 rl_1 r_1) = Snd (Fst x tl_1 rl_1 r_1)⌝
  THEN POP_ASM_T (asm_rewrite_thm_tac o eq_sym_rule)
  THEN REPEAT strip_tac THEN
  LEMMA_T ⌜Fst x ∈ OK_VC_d c⌝                                   LEMMA_THEN1
    (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])
  THEN1 PC_T1 "sets_ext" asm_prove_tac(map get_spec[⌜Elems⌝, ⌜Map⌝]),
  DROP_ASM_T ⌜te ∈ OK_VC_d c⌝                                   DROP_ASM
  (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])];

val tac3 = REPEAT strip_tac THEN
  cases_tac⌜Snd (te tl_0 rl_0 r_0) = Snd (te tl_1 rl_1 r_1)⌝     CASE_THENLIST
  THEN_LIST [
  asm_ante_tac ⌜¬ Snd (te tl_0 rl_0 r_0) = Snd (Fst x tl_0 rl_0 r_0)⌝
  THEN asm_rewrite_tac[]
  THEN STRIP_T (asm_tac o conv_rule(RAND_C eq_sym_conv)) THEN
  LEMMA_T ⌜Fst x ∈ OK_VC_d c⌝                                   LEMMA_THEN1
    (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])
  THEN1 PC_T1 "sets_ext" asm_prove_tac(map get_spec[⌜Elems⌝, ⌜Map⌝]),
  DROP_ASM_T ⌜te ∈ OK_VC_d c⌝                                   DROP_ASM
  (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])];

val tac4 = REPEAT strip_tac THEN
  DROP_ASM_T ⌜CaseVal c te cel ee ∈ OK_VC_d c⌝                  DROP_ASM
  (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th]);
```

Fig. 19. FEF tactics with patterns highlighted

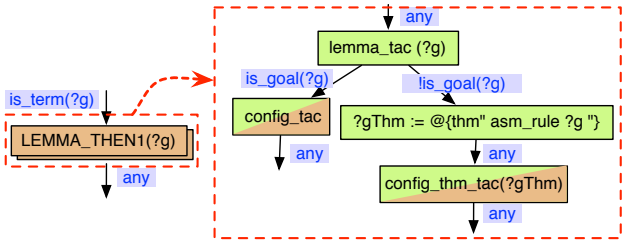


Fig. 20. PSGraph encoding of *LEMMA_THEN1*

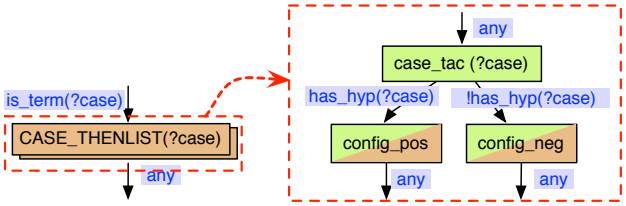


Fig. 21. PSGraph encoding of *CASE_THENLIST*

fun LEMMA_THEN1 g thm_tac tac = (*LEMMA_T g thm_tac*) *THEN1 tac*;

It will apply the cut rule by adding a new subgoal g. This is followed by an application of tactic *thm_tac* to the original goal (with g added to the list of hypothesis), and tactic *tac* to the new subgoal. This pattern can for example be seen in *tac1* of Fig. 19, where we can instantiate the pattern as follows:

LEMMA_THEN1
⌜*Snd x* ∈ *OK_VC*$_d$ *c*⌝
PC_T1 "*sets_ext*" *asm_prove_tac*(*map get_spec*[⌜*Elems*⌝, ⌜*Map*⌝])
(*fn th* => *all_fc_tac*[*rewrite_rule*(*map get_spec*[⌜*OK_VC*$_d$⌝])*th*])

Fig. 20 shows the PSGraph encoding of the *LEMMA_THEN1* pattern. The tactic is encapsulated in a graph tactic *LEMMA_THEN1(?g)* (left), where ?g is the subgoal in which the pattern is parametrised over. The ?g argument of the *LEMMA_THEN1* graph tactic is used to make ?g available for the nested graph, while the incoming goal type *is_term*(?g) ensures that ?g is bound to a term in the environment before the pattern is applied.

The right hand side of the figure shows the body of the tactic, which "unfolds" the *LEMMA_T* and *THEN1* tacticals. First, *lemma_tac(?g)* is applied, which applies the cut rule. The graph then depicts the parallel nature of how the new subgoal ?g and the existing subgoal are handled separately. The goal type is used to guide the goal to the correct tactic. However, the example highlights the limitation of parametrised graph tactics we have already discussed (and return to in §4.2.3). We need to introduce two dummy tactics, *config_tac* and *config_thm_tac*, which has to be manually replaced (renamed) when this pattern is used (as shown below).

4.2.1.2 *The* CASE_THENLIST *pattern.* The second pattern applies a case-split on a given variable *case*, followed by tactic *tac*1 when *case* holds, and tactic *tac*2 for its negation. This is called *CASE_THENLIST*:

fun CASE_THENLIST case tac1 tac2 = *case_tac case THEN_LIST*[*tac1*, *tac2*];

The PSGraph encoding of this pattern can be seen in Fig. 21. It is similar in structure of *LEMMA_THEN1*, and also illustrates how naturally PSGraph highlights that the two cases should be handled separately. It should be noted that, albeit sufficient in this case, *case_tac* may simplify the hypothesis thus !*has_hyp*(?*case*) will not work as expected in all cases.

4.2.1.3 *The* DROP_ASM *pattern.* In Fig. 19, the final pattern *DROP_ASM*, are instances of the *DROP_ASM_T* tactical. For purposes of the underlying pattern

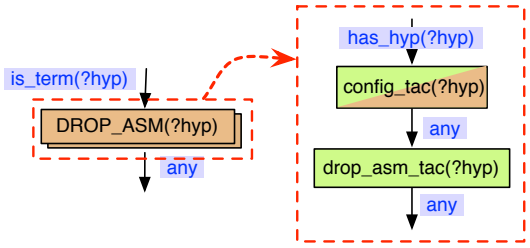


Fig. 22. PSGraph encoding of *DROP_ASM*

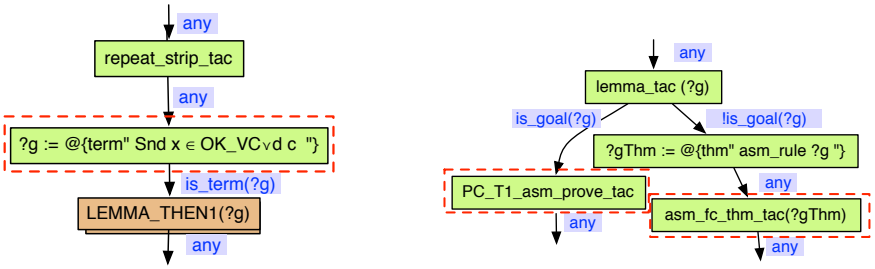


Fig. 23. PSGraph *tac1*: top level (left) and the sub-graphs of node *LEMMA_THEN1(?g)*(right)

representation in PSGraph, we unfold the meaning of *DROP_ASM_T*, creating the ML function:

fun DROP_ASM hyp thm_tac = thm_tac hyp THEN drop_asm_tac hyp;

It will first apply a tactic *thm_tac* using the given hypothesis *hyp*, and then remove the hypothesis by the *drop_asm_tac* tactic. Fig. 22 shows the PSGraph version.

4.2.2 *Encoding the FEF tactics in PSGraph.* The patterns can easily be used with Tinker's library functionality in a drag-and-drop manner[17].

We can then develop the tactics $tac1, \cdots, tac4$ in a modular way. The two smallest tactics are *tac*1 and *tac*4. They only use one single pattern. To illustrate, Fig. 23 shows *tac*1 encoded in PSGraph. In the figure, the stippled boxes illustrates the instantiation of the parametrised components of the pattern. ProofPower's *strip_tac* is first applied, followed by an environment tactics that binds variable $?g$ to $\ulcorner Snd\ x \in OK\_VC_d c \urcorner$[18]. This is followed by the *LEMMA_THEN1* pattern, where *config_tac* is replaced by *PC_T1_asm_prove_tac*, which is defined as

SML

PC_T1 "*sets_ext*" *asm_prove_tac*(*map get_spec*[$\ulcorner Elems \urcorner$, $\ulcorner Map \urcorner$]);

and *dummy_thm_tac* is replaced by *asm_fc_thm_tac(?gThm)*, which is defined as

SML

all_fc_tac[*rewrite_rule*(*map get_spec*[$\ulcorner OK\_VC_d \urcorner$]) *?gThm*]

[17] See the dedicate web page [42] for a screencast of the library functionality.
[18] Note that within the Tinker tool, we have followed the syntax used by Isabelle's anti-quotation mechanism where a term t is written @{*term* "t"}.

```
val tac2 = REPEAT strip_tac THEN
    cases_tac⌜Snd (te tl0 rl0 r0) = Snd (te tl1 rl1 r1)⌝      X: CASE_THENLIST
    THEN_LIST [
    asm_ante_tac ⌜¬ Snd (te tl1 rl1 r1) = Snd (Fst x tl1 rl1 r1)⌝      Y_pos
    THEN POP_ASM_T (asm_rewrite_thm_tac o eq_sym_rule)
    THEN REPEAT strip_tac THEN
    LEMMA_T ⌜Fst x ∈ OK_VC_d c⌝                      Z: LEMMA_THEN1
            (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])
    THEN1 PC_T1 "sets_ext" asm_prove_tac(map get_spec[⌜Elems⌝, ⌜Map⌝]),
    DROP_ASM_T ⌜te ∈ OK_VC_d c⌝                      Y_neg: DROP_ASM
    (fn th => all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝])th])];
```

Fig. 24. ProofPower code of FEF *tac2* tactic

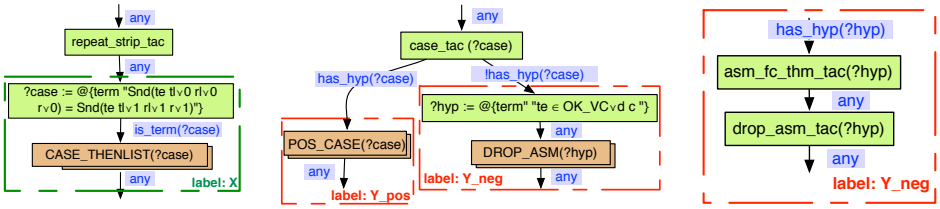


Fig. 25. The top-level, second-level and subgraphs of *DROP_ASM* of *tac2* in PSGraph

For more complicated tactics, such as *tac2* and *tac3*, we have broken the tactic up into components representing the identified proof patterns (where appropriate). We will demonstrate this with *tac2*. Fig. 24 gives the ProofPower code for this tactic, where the patterns have been highlighted as follows: the green box shows the *CASE_THENLIST* pattern; the red boxes show the positive and negative cases for *CASE_THENLIST*. Within the positive case, the blue box shows the *LEMMA_THEN1* pattern, while the negative case uses the *DROP_ASM* pattern.

Fig. 25 (left) shows the top-level view of the graph, which follows the same structure as *tac1*, albeit with the *CASE_THENLIST* pattern applied. The body of this graph tactic is shown in the middle of the figure. Here, *?case* is first instantiated to $\ulcorner Snd(tetl_0rl_0r_0) = Snd(tetl_1rl_1r_1) \urcorner$.Note the use of stippled labelled boxes to show which part of the ProofPower code each sub-graph corresponds to.

In the instantiation of *CASE_THENLIST* (middle), the tactics for the positive and negative cases, i.e. $config\_pos$ and $config\_neg$, are replaced by two sub-components. The positive case is an instance of the *DROP_ASM* tactic, shown at the right hand side of Fig. 25. Here, *?hyp* is initialised to $\ulcorner te \in OK\_VC_dc \urcorner$ and *config_tac(?hyp)* is replaced by *asm_fc_thm_tac(?hyp)*, which is defined to be

SML

```
all_fc_tac[rewrite_rule(map get_spec[⌜OK_VC_d⌝]) ?hyp])
```

The positive case is wrapped in a graph tactic *POS_CASE* shown in Fig. 26. The left hand side shows a flat version. Here, $?z$ is first bound to the term

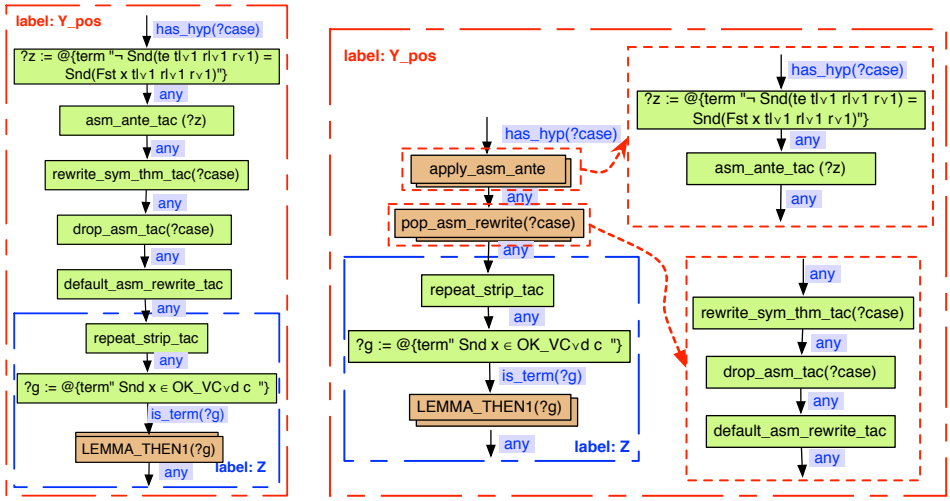


Fig. 26. *POS_CASE* of *tac2*: flat version (left) and a hierarchical version (right)

$\ulcorner \neg Snd(tetl_1rl_1r_1) = Snd(Fstxtl_1rl_1r_1) \urcorner$. The $asm\_ante\_tac(?z)$ tactic is then applied. This is followed by a sequential application of $rewrite\_sym\_thm\_tac(?case)$, $rewrite\_sym\_thm\_tac(?case)$ and $default\_asm\_rewrite\_tac$. The remaining part are the same as *tac1*, i.e. it can be developed by using the *LEMMA_THEN1* pattern. The right hand of Fig. 26 is the same as the left hand side, with hierarchies introduced for increased readability.

4.2.3 *Discussion.* The FEF case study is both *ad hoc* and domain specific, which is very different from the tautology example in the previous section. It has provided useful principles of how to apply PSGraph with a new perspective, where the goal is to extract/understand rather than reflect the proof structure. As a result, the proof structure follows more from instantiation of discovered proof patterns, rather than an overall proof plan.

The case study has provided us with a workbench to test robustness, and our next step will be to change the FEF specification and see how to fix the proofs in both ML and PSGraph and then compare the relative advantages. We could also re-create the real world issue that the proof is part of a big induction and the induction hypothesis has to be strengthened. This is a very common problem for inductive proofs.

The example has shown use of Tinker's library function to reuse patterns and how graph tactics can be parametrised by its input. However, it has also further indicated the need to parametrise graph tactic by actual tactics rather than their arguments, as we saw in the tautology example. Here, we provided dummy graph nodes that had to be manually replaced. One example to overcome this, is to allow tactics to be bound in the goal node environment; another example is to configure/replace tactics in the graph tactics. For example,

$$LEMMA\_THEN(config\_tac := another\_tac, ?g)$$

would replace tactic $config\_tac$ with a tactic $another\_tac$ for this particular use of the $LEMMA\_THEN$ graph tactic.

As the Tinker tool is foremost a research vehicle we have by design made it generic to work with multiple provers. Whilst we still think this is the right choice, a more specialised version for a particular prover could offer a closer level of integration, e.g. within the tool we have used @{*term* " − "}, more familiar for Isabelle users, compared with ProofPower's ⌜ − ⌝ representation.

4.3 Developing conversions

In this section we consider part of the implementation of a proof procedure described in [3] that automates proofs that a function formed from a given set of atomic morphisms by composition and pairing is a morphism in a concrete category. The proof procedure is itself parametrized by theorems that characterize the concrete category of interest and identify the atomic morphisms. So, for example, when instantiated for the category of topological spaces and continuous functions, with the arithmetic operators and the basic transcendental functions as the atomic morphisms, the proof procedure will automatically prove that the function $\lambda(x,y)\bullet\,(\mathsf{sin}(x)+\mathsf{cos}(y)+1)^2$ is continuous.

The first step in the decision procedure is to rewrite the λ-abstraction into a combinator form. In our example, the first step will prove the following theorem (which we state using the notation of [3] rather than ProofPower concrete syntax):

$$\begin{aligned}&\vdash(\lambda(x,y)\bullet\,(\mathsf{sin}(x)+\mathsf{cos}(y)+1)^2) = \qquad (*)\\ &\quad(\lambda x\bullet x^2)\circ\mathsf{Uncurry}(+)\circ\langle\mathsf{sin}\circ\pi_1,\mathsf{Uncurry}(+)\circ\langle\mathsf{cos}\circ\pi_2,\mathsf{K}\,1\rangle\rangle\end{aligned}$$

The proof procedure then proves that the right-hand side of (*) is a morphism in our category (i.e., a continuous function) by a process of backchaining with theorems stating that the combinators preserve morhphismhood and that the atomic functions are morphisms. See [3] for details. In this paper, we are only concerned with the implementation of the first step.

The implementation proves (*) by first applying a conversion λ_unpair_conv to eliminate the paired abstraction[19], yielding

$$\vdash(\lambda(x,y)\bullet\,(\mathsf{sin}(x)+\mathsf{cos}(y)+1)^2) = (\lambda p\bullet\,(\mathsf{sin}(\pi_1(p))+\mathsf{cos}(\pi_2(p))+1)^2)$$

(*) is then obtained by repeated application of the following system of rewrite rules to the right-hand side of the above equation.

$$\begin{array}{rcll}(\lambda x\bullet x) & \rightsquigarrow & \mathsf{I} & \\ (\lambda x\bullet t) & \rightsquigarrow & \mathsf{K}\,t & x\notin\mathsf{frees}(t)\\ (\lambda x\bullet(t_1,t_2)) & \rightsquigarrow & \langle(\lambda x\bullet t_1),(\lambda x\bullet t_2)\rangle & \\ (\lambda x\bullet f\,t) & \rightsquigarrow & f\circ(\lambda x\bullet t) & f\in\mathsf{unary}\\ (\lambda x\bullet g\,t_1\,t_2) & \rightsquigarrow & \mathsf{Uncurry}\,g\circ\langle(\lambda x\bullet t_1),(\lambda x\bullet t_2)\rangle & g\in\mathsf{binary}\\ (\lambda x\bullet h\,t\,j) & \rightsquigarrow & (\lambda x\bullet h\,x\,j)\circ(\lambda x\bullet t) & h\in\mathsf{parametrized}\end{array}$$

Here I and K are the identity and constant combinators, $\circ$ is functional composition, $\langle f,g\rangle$ is $\lambda(x,y)\bullet\,(f\,x,g\,y)$ and $\mathsf{Uncurry}\,g$ is $\lambda(x,y)\bullet\,g\,x\,y$. unary, binary and parametrized denote sets of atomic morphisms that are parameters to the proof procedure: unary and binary contain functions of one and two arguments respectively

[19] The implementation described here is slightly different from the algorithm described in [3] in that it handles paired abstractions by pre-processing rather than inside the rewrite system.

and parametrized contains functions of two arguments where the second argument is expected to be a constant. (The set parametrized is used to deal with families of functions like exponentiation with natural number coefficients.)

The ProofPower code that implements this rewrite system is the following.

ProofPower Code

```
val rec rec_conv : CONV = (fn t => (FIRST_C [
        i_conv,
        k_conv,
        pair_conv THEN_C RAND_C(RANDS_C(TRY_C rec_conv)),
        unary_conv THEN_C RIGHT_C (TRY_C rec_conv),
        binary_conv THEN_C
                RIGHT_C (RAND_C(RANDS_C (TRY_C rec_conv))),
        parametrized_conv THEN_C RIGHT_C (TRY_C rec_conv)]
                AND_OR_C simp_conv) t);
```

Here we have a conversion for each rule in the rewrite system, these are combined using the function *FIRST_C* which applies its argument conversions to a term in turn until it finds one that does not fail. Note the above definition local to a function *morphism_conv* which takes as parameters the sets of unary, binary and parametrized operators. The conversions *i_conv*, *k_conv* etc. each perform one rewriting step. *k_conv*, for example, attempts to rewrite a term at the outermost level using the theorem

$$\vdash \forall c\bullet\ (\lambda x\bullet\ c) = \mathsf{K}\, c$$

while *binary_conv* attempts to rewrite at the outermost level using theorems obtained from the following theorem:

$$\vdash \forall s\, t\bullet\ (\lambda x\bullet\ f(sx)(tx) = \mathsf{Uncurry}\, f \circ \langle s, t\rangle$$

by instantiating f to each of the binary atomic morphisms. If either of the first two conversions succeeds there is nothing more to do. In the other four cases, we must recursively apply the rewrite system to the λ-abstractions introduced by the conversion. This is done using functions *RAND_C* and *RANDS_C* of type $CONV -> CONV$, which apply their argument to the operand, respectively operands, of a function application. I.e., if *conv* proves $\vdash t = s, \vdash t_1 = s_1, \ldots, \vdash t_m = s_m$, then *RAND_C conv* proves all theorems of the form $\vdash f\, t = f\, s$ and *RANDS_C conv* proves all theorems of the form $\vdash g\, t_1 \ldots t_m = g\, s_1 \ldots s_m$, where g is not itself an application. The reader will observe that *rec_conv* also does something that is not specified in the rewrite system: *simp_conv* is a minor optimisation: it was found in practice that the rewrite system is prone to produce subterms of the form $f \circ \mathsf{I}$. While such subterms cause no problems with later processing, it is tidy to rewrite away the unnecessary composition and this is done by *simp_conv*, which is combined with the implementation of the rewrite system using the conversion combinator *AND_OR_C*, which does what its name suggests.

4.3.1 *The problem of transforming conversions into tactics.* As with the other case studies, we would like to transfer *rec_conv* into PSGraph in order to achieve a more intuitive representation to support future maintenance. However, PSGraph works with *tactics* while *rec_conv* is a *conversion*, and in order to use a conversion

in PSGraph a conversion must first be transformed into a tactic. Functionality to achieve this is provided by ProofPower via

SML
```
val conv_tac : CONV -> TACTIC
```

Using this function we can for example turn our *rec_conv* conversion into a tactic:

SML
```
val conv_rec_tac : TACTIC = conv_tac rec_conv
```

As we have seen, conversions are combined via a set of *conversion combinators*, comparable to tacticals for tactics. To follow the same approach as in the previous examples, we need to turn these into tacticals instead. For some examples we can define algebraic laws to support this. For example, sequential composition:

$$conv\_tac(conv1\ \ THEN\_C\ conv2) = (conv\_tac\ conv1)\ \ THEN\ (conv\_tac\ conv2)$$

We can then use the same approach as before and turn *conv_tacconv*1 and *conv_tac conv*2 into atomic tactics with a wire between them. The problem with this approach is in the presence of combinators such as *RAND_C*, which doesn't apply a conversion to a term, but to the operands of a function of the term. E.g. given a term $\ulcorner fx \urcorner$, *RAND_Cconv* will apply conversion *conv* to $\ulcorner x \urcorner$. We will call the family of conversion combinators that changes the "focus" of a term to a sub-term, such as *RAND_C* and *RANDS_C*, for *term focus* combinators.

Now, consider a conversion

SML
```
RAND_C (conv1  THEN_C conv2)
```

This will first change the focus of the term to the operand, and then apply *conv*1 followed by *conv*2 to the operand. We cannot naively break up the *THEN_C* combinator into the *THEN* tactical, as *conv*2 should work on the sub-term as a result of the use of *RAND_C*. For this case, we could again develop suitable algebraic laws and "push" *RAND_C* inwards, i.e.

SML
```
(RAND_C conv1)  THEN_C (RAND_C conv2) )
```

We can then use the same approach as above to turn this into a tactic.

A deeper problem for *rec_conv* is the recursion: in several cases a conversion is applied, followed by a recursive call wrapped in a term focus combiner, before *simp_conv* is applied to the pre-recursive focus. To achieve this in PSGraph, where recursion becomes a feedback loop, each call needs to keep track of the "current" term focus, together with the focus of the pre-call in order to apply *simp_conv* correctly.

4.3.2 *Encoding recursion with "term focusing" in PSGraph.* Our solution when implementing the *rec_conv* conversion, is to augment the PSGraph with a variable ?*rec*. This will act as a stack (frame) that keeps track of each change of focus to a term when entering a recursive step. This variable has type[20]

[20]We have updated the allowed types in an environment for simplicity – we could just as well used a list of names, and developed a parser to turn this into a conversion.

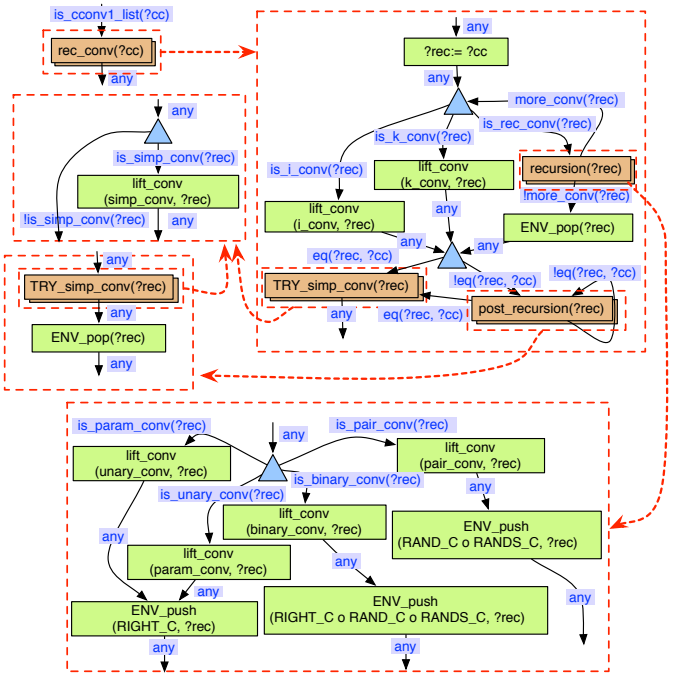


Fig. 27. Encoding recursion with "term focusing" in PSGraph

$?rec : (CONV \texttt{ -> } CONV)\ list$

The idea is as follows. For each application of a conversion, we first sequentially compose all elements of $?rec$ and apply this to focus the correct place in a term followed by the application of a conversion. We specialise *conv_tac* with a tactic that takes these two arguments:

ML

$$lift\_conv\text{: } (CONV \texttt{ -> } CONV)\ list \texttt{ -> } CONV \texttt{ -> } tactic$$

Here, the first argument is a list of term focusing combinators and the second argument is desired conversion. *lift_conv* has the following semantics:

$$lift\_conv\ [f_1; \ldots; f_n] conv = conv\_tac\ \big((f_n \circ \ldots \circ f_1)\ conv\big)$$

If this conversion is followed by a recursive application, then the "term focus combinators" are pushed to the $?rec$ stack. This is achieved by an environment tactic *ENV_push(conv,var)*, which takes one of these conversion combinators and a variable as argument.

Once the recursion has completed *simp_conv* should be applied to each operand, in the inverse order of the recursion (i.e. the last sub-term that a conversion was applied to should be simplified first). This is achieved by using $?rec$, and apply *simp_conv*, followed by popping an element of it until $?rec$ is empty. The environment tactic *ENV_pop(var)* applies such a single 'pop' operation.

The encoding of this strategy is shown in Fig. 27. It takes an input goal with an environment containing a variable $?cc$, which has the "term focus" on entry

(and exit). It initialises ?*rec* to this value. The *i_conv* and *k_conv* conversions are applied without any recursion, as described above. For the other cases, the *recursion* graph tactic is added. For each case conversion *c*, i.e. each element of the list of *rec_conv* we develop an atomic goal type, called *is_c_conv*. For the graph tactic, *is_rec_conv* is just the disjunction of the four recursive cases:

$$\begin{aligned} is\_rec\_conv(X) &\leftarrow is\_unary\_conv(X).\\ is\_rec\_conv(X) &\leftarrow is\_param\_conv(X).\\ is\_rec\_conv(X) &\leftarrow is\_binary\_conv(X).\\ is\_rec\_conv(X) &\leftarrow is\_pair\_conv(X). \end{aligned}$$

For each of the cases, the conversion is applied and then the focus of the term is changed. For example *pair_conv* is followed by two applications of *RAND_C* and then a recursive call again to *rec_conv*. Thus, for this case, *RAND_C* ∘ *RAND_C* is added to ?*rec*.

The recursion stops when there are no further applicable conversions, i.e. when none of the goal types succeed. This is the negation of the case where any of the conversion goal types are satisfied, i.e.:

$$\begin{aligned} more\_conv(X) &\leftarrow is\_i\_conv(X).\\ more\_conv(X) &\leftarrow is\_k\_conv(X).\\ more\_conv(X) &\leftarrow is\_rec\_conv(X). \end{aligned}$$

On exit, the last change of focus is removed (as it failed for this application), and we enter the *post_recursion* graph tactic, unless the case where ?*rec* is still the same as ?*cc*. This indicates that none of the recursive calls succeeded. In this graph tactic, the *simp_conv* will be applied for the same sub-terms as in the recursive case (due to the feedback loop around the tactic). When ?*rec* is the same as ?*cc*, the loop is terminated, which ends the proof.

4.3.3 *Discussion.* When contrasting the ML code of *rec_conv* in the start of the section with the corresponding PSGraph in Fig. 27, one can see that choices made are much more declarative; the parallel view of the choice of conversion, which one had in mind in the first place, is clear to see compared with the sequential ordering of the ML code. One of the most common bugs in such development is to get the scheduling wrong, and, as we saw with the tautology example, PSGraph is good at supporting this type of debugging.

The use of conversions is very common: they are used heavily in our work with D-RisQ's powerful tactic [43], where we are developing conversions to port the tactic from verification of Ada programs to C. The work presented here is a starting point which has exposed a lot interesting questions when dealing with low-levels issue involving terms and rewriting. In particular, how to evaluate sub-parts of terms within a given context across multiple boxes in an efficient way. In the presented solution we had to re-build data structures from outside and inwards for each box before actually applying the rewriting, which is very expensive.

One way to overcome this in the future is to augment the goal node with the *current focus* of a goal (conclusion and/or hypothesis). Huet's Zippers [33], which combines a tree with a sub-tree of focus, including operations to this focus move up/down/sideways could for example be applied. This has been implemented for a term representation similar to ProofPower's in [16]. Another option is Grundy's

window inference[28], which will also build up contextual information when focus is shifted to a sub-term.

Another question is whether tactics and conversions should be treated uniformly or separately. To treat conversions separately would be a more radical extension to PSGraph, perhaps involving a new *conversion graph notation* devoted to equational reasoning, in which the boxes would denote conversions rather than tactics.

5. RELATED WORK

Motivated mainly by large scale proof developments, such as the Coq proof of the four colour theorems [23], the HOL light proof of the Kepler conjecture [29] and the Isabelle verification of the seL4 micro-kernel [38], ideas from software engineering have started to make their way into proof development. The term *proof engineering* has been used for this new discipline [37]. This includes topics such as *productivity* of proofs [62] as well as proof *metrics* [7].

Whilst we put the work presented here in this context of proof engineering, our motivation and focus is different from the above work. Instead of being motivated by large proof developments and their proofs, our motivation has been development and maintenance of a large proof strategy (i.e. *tactic*) in an industrial setting, using a graphical representation [43]. There has been a considerable amount of work on visualising *proof trees*, including: LΩUI [58] for Ωmega; XIsabelle [49] for Isabelle; ProveEasy [12] and Jape [9] for teaching; and some more recent work for Mizar [41, 50]. However, none of these visualise the high-level strategy. This raises the question of what the difference between a proof and proof strategy is (in a mechanical theorem proving setting)? If one think of tactics as proof strategies, then a proof strategy is really just a procedural description of how to conduct the search for a proof. Bundy [11] on the other hand, has argued for the additional role of *explanation*. We hope that we have shown how PSGraph can help explaining the strategy of a proof in addition to be used to guide the search. Such explanation is important for maintenance when team changes, and our work with D-RisQ [43] has had very promising initial results when porting their proof tactics from Ada to C verification.

Through case studies, we have conducted a comprehensive comparison of PSGraph with ProofPower's tactic language. This language is very similar to the tactic languages found in LCF-based provers such as HOL4, HOL light, and Isabelle. LTac for Coq [15] and Eisbach for Isabelle [46] provide support for tactic development within the proof scripts. They support matching features similar to the environment tactic and goal types of PSGraph. However, the composition of tactics is closer to the more traditional tactic languages and they do not contain debugging features as presented for PSGraph here. In *proof planning* [10] tactics are given pre-conditions and post-conditions, which can be achieved by the goal types of PSGraph. IsaPlanner [16] is a proof planner for Isabelle, and the PSGraph project started out as a new composition (tactic) language for IsaPlanner (which was similar to ProofPower's tactic language). There are also several *typed tactic* languages, e.g. VeriML [61], Beluga [53], Delphin [54] and Mtac [67]. Here the types contain information about the relationship between tactics and the proofs produced. For VeriML, Beluga and Delphin there is a clear separation between the tactic and the base logic. It is not clear how they can be incorporated in an

established theorem prover, as we have detailed for PSGraph and ProofPower here. Mtac is an extension to Coq and, as with LTac, does not address the issues of maintenance and usability we are focusing on, albeit with improved static (composition) properties through types. Autexier and Dietrich [8] have developed *a declarative tactic language on top of a declarative proof language* where a strategy is represented as a *schema* which needs to be instantiated. Their work is more declarative than PSGraph, whilst PSGraph handles tactic compositions in a more declarative way. However, it is not clear how issues such as debugging is helped by such schemas. There have also been several attempts to create *declarative tactic languages on top of procedural tactic languages* [31, 22]. Asperti et al [5] argues that these approaches suffer from two drawbacks: goal selection for multiple subgoals, and information flow between tactics – both of these are addressed by goal types in PSGraph. Finally, HiTac is a tactic language with additional support for hiding complexities using hierarchies [6], which our notion of hierarchies is based upon. Hierarchies in HiTacs are restricted to single inputs, which probably makes the semantics simpler and more elegant, whereas our approach, which utilises goal types, has followed from a more practical and empirical approach. HiTac has predominantly been used for more theoretical work, and as with VeriML, Beluga and Delphin, it has not been integrated into an established theorem prover.

As the evaluation was partly exploratory in nature, a case study approach was chosen as case studies are considered suitable for qualitative analysis [57]. The fact that the third author (Arthan) was new to PSGraph also provide new and more objective reflections in this work. For quantitative results, an alternative approach would have been to set up a controlled experiments, e.g. where participants are given suitable training with both ProofPower's tactic languages and PSGraph, and are then asked to explain and identify/repair faulty proof strategies. We can then measure the time and compute and compare their *mean time to repair* (MTTR) metric [20, p.462]. There are also models for maintainability that are based upon *activities*, which may be more suitable as they have been applied to both text-based code and (graphical) Simulink models (in industrial settings) [14]. This can form the basis for future experiments. We could also try to measure and compare structural factors of the two representations, however metrics for both usability [20, p.460] and maintainability (e.g. the *maintainability index* (MI) [48]) are tailored for linear languages and would not be applicable to our graphical language. To illustrate, MI combines measures such as lines of code, cyclomatic complexity and Halstead effort[21] [30].

We have studied and contrasted *programs* written in different languages. However, what we are really interested is the usabilty and maintainability provided by the *languages* themselves. In §1 we gave one example from cognitive science that shows that humans find it more natural to comprehend flow networks diagrammatically compared with linear (term-based) representations [39]. Block-based languages, such as Logo [21], Scratch [56], Alice [13] and AppInventor [66], are similar to PSGraph in that programs are inter-connecting blocks. Block-based languages are popular within CS education and there have been some studies comparing them with traditional languages. E.g. [63] found that the drag-and-drop mechanism

[21] This is computed using the number of operators, operands and program length.

and the ease of browsing block-based libraries were advantageous; whilst perceived drawbacks relate to expressibility and scalability. Besides this, we are not familiar with any work comparing actual languages[22]. Simulink's popularity for embedded systems shows industrial use of graphical languages for certain types of systems.

We are not familiar with any work on case studies in refactoring proof tactics as comprehensive as the work presented in the present paper. Whiteside (with others) has developed a refactoring framework for hierarchical proofs (HiProofs) [65, 64], however this work focused mainly on proofs rather than proof strategies, albeit including some work on folding and unfolding tactics. The most relevant tool to ours, which we are familiar with, is the aforementioned *Tactician* tool to refactor proofs in HOL light [2]. In HOL light (as is also the case for ProofPower) a proof can be a sequence of (interactive) 'apply' step, or they can be combined into a single step (by means of tactic combinators) which is then applied. Tactician is a tool to fold sequences into a single tactic and unfold a tactic into a sequence of steps. This can then be used for debugging by enable users to step through a large tactic, similar to how this can be achieved with Tinker. However, it only work for a small subset of ML and it is not clear how this approach can be generalised to arbitrary tactics. Moreover, it unfolds only one particular branch of the proof which does not necessarily reflect the underlying proof strategy.

Another tool recently developed to support debugging is the new tracing mechanism for the *simp* tactic in Isabelle [34]. This is implemented as plug-in for the Isabelle/jEdit Prover IDE. It supports hierarchical viewing of simplification traces, and, as with Tinker, it enables breakpoints to be inserted where the user can step through and interact with the tactic. The breakpoints can either be an application of certain theorems or if the subgoal matches certain patterns. Note that it is not used to debug the (sub-)tactics used to implement the simplifier: it will only show how the simplifier applies rewrite steps. Our logging mechanism is considerably simpler, and closer to the more rudimentary ones supported in other ITP systems (including the previous tracing mechanism for the *simp* tactic in Isabelle). However, in practice we have found that our logging mechanisms is sufficient as it only relates to a step at a time, while the *simp* tactic could involve hundreds of steps.

6. CONCLUSION & FUTURE WORK

We have extended our graphical approach for tactic development with features to support development and debugging of proof strategies. Through examples, we have shown relative advantages when it comes to *understanding* and *debugging* proof strategies compared with ML code, which is part of the hypothesis addressed here. We would argue that *maintenance* is a consequence of this, and aim to further study this this by completing the tasks set out in the FEF case study.

The experiments have provided some valuable lessons. The inclusion of PSGraph into a theorem prover should be seen to be *evolutionary* rather than revolutionary; it should be seen as a new tool in the ecosystem for LCF-style theorem proving systems. An important feature is that you can still develop standard tactics using tacticals and treat them as atomic in PSGraph. This is a positive, and, with some

[22]There has been some more interest recently, as illustrated by a SIG meeting on *usability of programming languages* at CHI 2016.

better integration with particular theorem proving and SML system, you will still have the old version with an additional tool. It should also be noted that Tinker has been connected to a non-SML based prover: Rodin is developed using Java [40].

To use PSGraph effectively, one needs to change from a purely procedural view of tactics to more declarative thinking. There are trade-offs to be made between a graphical, declarative and parallel approach on the one hand and the (hopefully) concise higher-order functional programming approach on the other. Whilst losing some of the power of functional programming, the graphical approach is intrinsically more accessible. In a world where teams change, a solution perceived as elegant by the original developer may be perceived as incomprehensible and unmaintainable by his or her successors, particularly as the raw code of a tactic does not document *why* design choices were made. If this is true for experts on tactics – then what would 'Joe Engineer' prefer if this technology is to become mainstream? One compromise between procedural and declarative styles may be to think about how to represent tacticals graphically. PSGraph is essentially a way of composing tactics. A way forward may be to first just replace tacticals with similar graphical tactics: this was illustrated this in the FEF case study, while [26] shows some common tacticals graphically.

Technically, in order to use PSGraph with a new theorem prover all that has to be done is to implement a ML signature which tells PSGraph how to work with the proof state of that prover. Detailed information of how this is done in ProofPower can be found in [42].

The case studies have shown some limitations with PSGraph that need to be addressed. This includes, more *parametric graph tactics* to enable better re-use and modularity; closer integration with the theorem proving system to make it more natural for users; and low level term manipulations and rewriting, as illustrated by our encoding of conversions. Building on the FEF case study we would also like to test the robustness and maintainability, and contrast this with the linear tactic language such as ML. Another experiment we would like to conduct is to start from scratch and implement known (or even unknown) decision procedures in PSGraph (e.g. quantifier elimination), and start building up a library that can help in general. It will also be interesting to see how PSGraph can be used as explanation of existing large libraries (e.g. the AFP for Isabelle or the Mizar library) – or even as a good representation when writing papers on new decision procedures.

As PSGraph is implemented as a generic tool, and not specialised to a particular theorem prover, it may provide a way of translating proof strategies between theorem proving systems. While OpenTheory [35] relies on the low-level similarities of the LCF kernels to translate proofs, PSGraph could potentially help exporting high-level proof strategies. This would require some minimal set of atomic tactics and goal types and just use them (with a defined semantics). Short term we could at least carry across structures of proof strategies, where the user needs to fill in the details in terms of atomic tactics and goal types. PSGraph could also be an experimental way of developing proof procedures, acting as the "source code" with tactic languages as the "target code" as they are likely to be more efficient than our graph interpreter. It is however not clear how to do such compilation; possibly bottom-up in an interactive manner.

We have shown how our graphical representation can help to explain and support maintenance of selected examples from a real-world system. The next challenge is to continue our work with D-RisQ [43] on their 10K LoC tactic. Here, we believe that PSGraph can provide real economic advantages in terms of reduced development costs, reduced maintenance costs and improved communication with the user community.

References

[1] IEEE Standard 610.12-1990. Standard Glossary of Software Engineering, Terminology. IEEE Computer Society Press, 1993. 70

[2] Mark Adams. Refactoring proofs with tactician. In Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe, editors, *Software Engineering and Formal Methods: SEFM 2015. Revised Selected Papers*, pages 53–67, Berlin, Heidelberg, 2015. Springer. 108, 125

[3] Rob Arthan. Now f is continuous (exercise!). *Journal of Formalized Reasoning*, 9(1):33–52, 2016. 118

[4] Rob Arthan and Roger Bishop Jones. Z in HOL in ProofPower. *BCS FACS FACTS*, 2005-1. http://www.bcs.org/upload/pdf/facts200503.pdf. 70, 71

[5] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A new type for tactics. In *PLMMS'09*, pages 229–232, 2009. 124

[6] David Aspinall, Ewen Denney, and Christoph Lüth. A Tactic Language for Hiproofs. In *MKM'08*, pages 339–354, Berlin, Heidelberg, 2008. Springer-Verlag. 124

[7] David Aspinall and Cezary Kaliszyk. Towards formal proof metrics. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering: 19th International Conference, FASE 2016*, pages 325–341, Berlin, Heidelberg, 2016. Springer. 123

[8] Serge Autexier and Dominik Dietrich. A Tactic Language for Declarative Proofs. In *ITP'10*, volume 6172 of *LNCS*, pages 99–114. Springer, 7 2010. 124

[9] Richard Bornat and Bernard Sufrin. Jape: A calculator for animating proof-on-paper. In *CADE-14*, pages 412–415. Springer, 1997. 123

[10] Alan Bundy. A science of reasoning. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 178–198, 1991. 123

[11] Alan Bundy. A science of reasoning. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 10–17. Springer, 1998. 123

[12] Rod Burstall. Proveeasy: Helping people learn to do proofs. *ENTCS*, 31(0):16 – 32, 2000. 123

[13] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000. 124

[14] Florian Deissenboeck, Stefan Wagner, Markus Pizka, Stefan Teuchert, and J-F Girard. An Activity-Based Quality Model for Maintainability. In *2007 IEEE International Conference on Software Maintenance*, pages 184–193. IEEE, 2007. 124

[15] David Delahaye. A Proof Dedicated Meta-Language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002. 123

[16] Lucas Dixon and Jacques D. Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *CADE-19*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003. 122, 123

[17] Lucas Dixon and Aleks Kissinger. Open Graphs and Monoidal Theories. *CoRR*, abs/1011.4114, 2010. 80

[18] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006. 80

[19] Colin Farquhar, Gudmund Grov, Andrew Cropper, Stephen Muggleton, and Alan Bundy. Typed meta-interpretive learning for proof strategies. In *Late Breaking Papers of ILP2015*, volume 1636, pages 17–32. CEUR Workshop Proceedings, 2016. 89, 93

[20] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 2014. 124

[21] Wallace Feurzeig et al. Programming-languages as a conceptual framework for teaching mathematics. final report on the first fifteen months of the logo project. 1969. 124

[22] Mariusz Giero and Freek Wiedijk. MMode, a Mizar Mode for the proof assistant Coq. Technical report, January 07 2004. 124

[23] Georges Gonthier. A computer-checked proof of the four colour theorem, 2005. 123

[24] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. 71

[25] Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000. 71, 75

[26] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. A Graphical Language for Proof Strategies. In *LPAR*, pages 324–339. Springer, 2013. 70, 88, 126

[27] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. Tinker, Tailor, Solver, Proof. In *UITP 2014*, volume 167 of *ENTCS*, pages 23–34. Open Publishing Association, 2014. 70, 83

[28] Jim Grundy. Window inference in the HOL system. In *International Workshop on HOL Theorem Proving System and Its Applications*, pages 177–189. IEEE, 1991. 123

[29] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, et al. A formal proof of the kepler conjecture. *arXiv preprint arXiv:1501.02155*, 2015. 123

[30] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier, 1977. 124

[31] John Harrison. A Mizar Mode for HOL. In *TPHOLs*, volume 1125 of *LNCS*, pages 203–220. Springer, 1996. 124

[32] John Harrison. HOL light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 60–66. Springer, 2009. 70, 71

[33] Gérard Huet. The zipper. *Journal of functional programming*, 7(05):549–554, 1997. 122

[34] Lars Hupel. Interactive simplifier tracing and debugging in isabelle. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *CICM*, pages 328–343, Cham, 2014. Springer. 125

[35] Joe Hurd. OpenTheory: Package management for higher order logic theories. In Gabriel Dos Reis and Laurent Théry, editors, *PLMMS '09: Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 31–37, August 2009. 126

[36] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A Proof Assistant for Diagrammatic Reasoning. In *CADE-25*, volume 9195 of *LNCS*, pages 326–336. Springer, 2015. 84

[37] Gerwin Klein. Proof engineering considered essential. In *FM 2014: Formal Methods*, pages 16–21. Springer, 2014. 123

[38] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):2, 2014. 123

[39] Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987. 70, 124

[40] Yibo Liang, Yuhui Lin, and Gudmund Grov. 'the tinker' for rodin. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, pages 262–268. Springer, 2016. 70, 126

[41] Tomer Libal, Martin Riener, and Mikheil Rukhaia. Advanced Proof Viewing in ProofTool. In *UITP 2014*, volume 167 of *EPTCS*, pages 35–47. Open Publishing Association, 2014. 123

[42] Yuhui Lin, Gudmund Grov, and Rob Arthan. Debugging and refactoring tactics graphically – paper resources. `http://ggrov.github.io/tinker/jfr16/`. Accessed: July 2016. 71, 84, 115, 126

[43] Yuhui Lin, Gudmund Grov, Colin O'Halloran, and G Priiya. A Super Industrial Application of PSGraph. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 319–325. Springer, 2016. 70, 94, 122, 123, 127

[44] Yuhui Lin, Pierre Le Bras, and Gudmund Grov. Developing and debugging proof strategies by tinkering. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 573–579, Berlin, Heidelberg, 2016. Springer. 70, 83, 84

[45] Makarius Wenzel and many others. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/isar-ref.pdf. 70, 71

[46] Daniel Matichuk, Makarius Wenzel, and Toby Murray. An isabelle proof method language. In Gerwin Klein and Ruben Gamboa, editors, *5th International Conference on Interactive Theorem Proving*, pages 390–405, Cham, 2014. Springer. 123

[47] Colin O'Halloran. Automated Verification of Code Automatically Generated from Simulink. *ASE*, 20(2):237–264, 2013. 70

[48] P Oman, J Hagemeister, and D Ash. A definition and taxonomy for software maintainability, report setl report 91-08-tr. *University of Idaho*, 1991. 124

[49] Maris A Ozols, Anthony Cant, and Katherine A Eastaughffe. Xisabelle: A system description. In *CADE-14*, pages 400–403. Springer, 1997. 123

[50] Karol Pak. The algorithms for improving and reorganizing natural deduction proofs. *Studies in Logic, Grammar and Rhetoric*, 22(35):95–112, 2010. 123

[51] Lawrence C. Paulson. A higher-order implementation of rewriting. *Sci. Comput. Program.*, 3(2):119–149, 1983. 73

[52] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1987. 98, 99, 100

[53] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM SIGPLAN Notices*, volume 43, pages 371–382. ACM, 2008. 123

[54] Adam Poswolsky and Carsten Schürmann. System description: Delphin–a functional programming language for deductive systems. *Electronic Notes in Theoretical Computer Science*, 228:113–120, 2009. 123

[55] Gill Prout, Rob Arthan, Roger Jones, and Roger Stokes. The Front End Filter Project: specification and verification of a secure database system. http://www.lemma-one.com/ProofPower/fef/fef.html. 108, 109, 110

[56] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009. 124

[57] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009. 71, 96, 124

[58] Jörg H. Siekmann, Stephan M. Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. LOUI: Lovely OMEGA User Interface. *Formal Asp. Comput*, 11(3):326–342, 1999. 123

[59] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*. Springer, 2008. 70, 71

[60] J Michael Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992. 72

[61] A. Stampoulis and Z. Shao. VeriML: Typed Computation of Logical Terms inside a Language with Effects. In *ICFP*, pages 333–344. ACM, 2010. 123

[62] Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. Productivity for proof engineering. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 15. ACM, 2014. 123

[63] David Weintrop and Uri Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*, pages 199–208. ACM, 2015. 124

[64] Iain Whiteside. *Refactoring Proofs*. PhD thesis, University of Edinburgh, 2013. 125

[65] Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov. Towards Formal Proof Script Refactoring. In *CICM'11*, volume 6824 of *LNCS*, pages 260–275. Springer, 2011. 125

[66] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor*. " O'Reilly Media, Inc.", 2011. 124

[67] Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: a monad for typed tactic programming in coq. In *ACM SIGPLAN Notices*, volume 48, pages 87–100. ACM, 2013. 123