Tactics for Hierarchical Proof

David Aspinall, Ewen Denney and Christoph Lüth

Abstract. There is something of a discontinuity at the heart of popular tactical theorem provers. Low-level, fully-checked mechanical proofs are large trees consisting of primitive logical inferences. Meanwhile, high-level human inputs are lexically structured formal texts which include *tactics* describing search procedures. The proof checking process maps from the high-level to low-level, but after that, explicit connections are usually lost. The lack of connection can make it difficult to understand the proof trees produced by successful tactic proofs, and difficult to debug faulty tactic proofs.

We propose the use of hierarchical proofs, also known as *hiproofs*, to help bridge these levels. Hiproofs superimpose a labelled hierarchical nesting on an ordinary proof tree, abstracting from the underlying logic. The labels and nesting are used to describe the organisation of the proof, typically relating to its construction process. In this paper we introduce a foundational tactic language *Hitac* which constructs hiproofs in a generic setting. Hitac programs can be evaluated using a big-step or a small-step operational semantics. The big-step semantics captures the intended meaning, whereas the small-step semantics is closer to possible implementations and provides a unified notion of proof state. We prove that the semantics are equivalent and construct valid proofs. We also explain how to detect terms which are stuck in the small-step semantics, and how these suggest interaction points with debugging tools. Finally we show some typical examples of tactics, constructed using tactical combinators, in our language.

Keywords. hierachical proof, hiproof, tactical theorem proving.

1. Introduction

Interactive theorem proving is a challenging pursuit, made additionally challenging by the present state of the art. Constructing significant sized computer checked proofs requires struggling with unpredictable automation, grappling with low-level system specific details, and using primitive and frustrating user interfaces.

As a way to help improve automation, interactive theorem provers allow users to write embedded proof procedures, known as *tactics*, in various prover-specific languages. Tactics allow the user not only to write shorter proof texts, but can also increase readability, applicability, and maintainability of proofs, because as well as proving one specific goal at hand they may be applicable in more situations.

However, writing tactics is not easy. First, there is only a remote connection between the human inputs and the final low-level proof tree composed of the primitive rules of the logic. The low-level proof tree is usually invisible to the user, and in most systems, not updated.

A second reason that tactics are difficult to write is that they are often expressed in rather general purpose programming languages (such as ML or Lisp), using a programming interface provided by the proof engine. As a result, tactic languages vary widely from one theorem prover to the next, locking users into specific provers they have mastered. Moreover, any attempt to give a semantics to tactics has to tackle the full programming language, and will thus be based on the traditional notions of computation instead of a more central notion of *proof.*

What is needed is a more abstract notion of proof which is independent of a particular prover or logic, but supports the relevant notions needed to interactively explore and construct proofs. To help explore and understand large homogeneous proofs, we seek a way of imposing some modular structure on them. To help construct and debug modularised proofs, we need tactics which add structure and can be referenced from the proofs. And ideally, all of this can be independent of both the underlying logic and the prover being used, just as the basic ideas of proof tree and proof search are system independent.

In this paper, we study a tactic language Hitac designed for *hiproofs*, a notion introduced by Denney et al [7]. Hiproofs allow a *hierarchical* structure on proofs which serves as the primary modularisation mechanism. Although any system in which tactics can be defined from other tactics leads naturally to a notion of hierarchical proof, this is the first work to study the hierarchical nature of tactic languages in detail alongside a natively hierarchical form of proof. Hiproofs and Hitac are very general, building on a simple form of underlying logical framework.

1.1. Introducing Hiproofs

Consider the very high-level view of Gödel's first incompleteness theorem in Figure 1: given a sufficiently powerful logic L, we show that it is either incomplete or inconsistent. This is shown by *reductio ad absurdum*: assuming the logic is both complete and consistent, we derive a contradiction. The proof works using a Gödelisation, i.e., a map $\ulcorner - \urcorner$ from the propositions in L to a subset of the natural numbers such that ϕ is derivable iff $\ulcorner \phi \urcorner$ is true. We then diagonalise, constructing a proposition $\sigma(x)$ in L s.t. $\sigma(\ulcorner \phi \urcorner)$ is true iff ϕ is *not* derivable. Finally, we consider whether $\sigma(\ulcorner \sigma \urcorner)$ is true or false, and in both cases derive a contradiction (if $\sigma(\ulcorner \sigma \urcorner)$ is true, σ must be false, and vice versa).

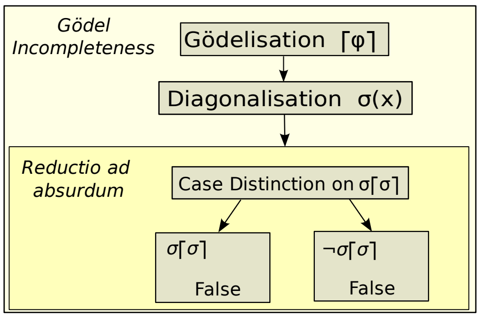

FIGURE 1. The proof of Gödel's first incompleteness theorem.

The picture in Figure 1 is a graphical representation of the process of this proof. It is neither precise nor does it give sufficient detail, but it shows enough to help convey the structure of the *reductio*. For example, one can see the sequence of steps and the internal structure which shows the two subgoals to be solved inside. One can imagine filling out all the boxes with the relevant details to make it a full proof.

On the other hand, given a fully detailed proof tree, we can get a similar abstract representation. Consider a natural deduction proof of the easy theorem that $\forall x.\ x + 0 = x$ in Presburger arithmetic. In its proof tree (Figure 2, left), we can discern the relevant structure: the induction step, usage of elementary axioms P_3, P_4 and the induction hypothesis ax, etc. By eliding unimportant details such as transitivity and congruence rules and then turning the deduction tree on its head, we get a hierarchical proof similar to the outline of Gödel's theorem.

$$\dfrac{\dfrac{}{0+0=0}\,\mathsf{P3} \qquad \dfrac{\dfrac{\dfrac{}{k+0=x}\,\mathsf{ax}}{S(k+0)=S(k)}\,\mathsf{cong}}{S(k)+0=S(k)}\,\mathsf{P4, trans}}{\forall x.\ x+0=x}\,\mathsf{Induction}$$

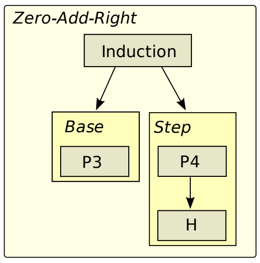

FIGURE 2. A natural deduction proof of the theorem $x + 0 = x$, and the associated hiproof; $S(X)$ is the successor function.

These abstract representations of proofs are called *hiproofs*, and they are the main objects of our study. Hiproofs have nodes, which correspond to basic proof rules, procedures or compositions thereof, and directed edges connecting them in sequence. Nodes may be *labelled* with a name to make a nested box that contains further nodes. Conceptually, or inside an actual implementation, navigation in the hiproof allows "zooming in" by opening boxes to reveal their content. Boxes which are not open may be visualised by just their labels; details inside are suppressed.

Hiproofs are subject to certain well-formedness constraints which facilitate navigation and their connection to proof trees. For example, each nesting level must have a unique root; arrows may only emanate from innermost nodes and only target outer nodes. Two alternative characterisations of hiproofs have been studied in previous work [7]: either as a tree with a level function (representing the nesting depth of nodes), or as a pair of forests on the same set of nodes (one forest captures containment, the other sequencing).

In this paper we begin by introducing a corresponding formal syntax for hiproofs. The syntax in fact ensures well-formed hiproofs by construction; we are more interested in the correctness question: when is a hiproof a correct abstraction of a real underlying proof? We shall call a hiproof *valid* if it can be mapped onto an ordinary proof tree in an underlying derivation system, where nodes are given by derivable judgements in the normal sense.

The central topic and main novelty, is our tactic language *Hitac* for constructing valid hiproofs. The language is general and not tied to a specific logic or implementation. It is deliberately minimal: we seek to understand the connection between hierarchical structure and the essential core constructs for tactic programming, namely, alternation, repetition and assertion.

1.2. Contributions and relationship to existing systems

Our Hitac language is a step towards a rigorous but simple semantic foundation for interactive hierarchical tactical theorem proving. The contribution we make is in defining the language and its semantics, and in proving that the definitions are, in a technical sense, correct.

Our wider goal is to build this foundation in stages, maintaining the ability to prove correctness and other desirable properties. By doing this at the same time as studying hierarchical structuring, we hope to shed light on underlying mechanisms which are not altogether straightforward and at best only partially described in the existing literature. The language Hitac presented here, therefore, is a core calculus and omits some features which would be desirable in an implementation. For example, we do not yet treat meta-variables, and we do not attempt to explain higher-order tactics. See Section 8 for more comparison and Section 9 for more discussion of as yet unmodelled features.

1.3. Outline

The rest of this paper is organised as follows. In Section 2, we introduce a syntax for hiproofs and explain the notion of validity. By extending this syntax we define *tactics* which can be used to construct tactic programs, both described in Section 3.

In Sections 4 and 5 we study the notion of *evaluation* with operational semantics: a tactic can be applied to a goal and, if successful, evaluated to produce a valid hiproof. There are two operational semantics: a big-step relation (Section 4) which lays down the meaning of our constructs, and a finer-grained small-step semantics (Section 5) which provides a notion of *proof state* that evolves while the proof is constructed.

A proof state can be examined to determine the progress of a proof. Section 6 considers this in more detail, to examine the parts of a proof which will not succeed; we call these proof parts *futile*. Futile subterms can be inspected to determine the cause of failure.

In Section 7 we demonstrate how our language can be used to define some familiar tactics, and show how our evaluation works on them. Finally, Section 8 describes some of the related work, and Section 9 concludes.

2. Hiproofs and derivation systems

Hiproofs have a concise syntax, introduced next. They add structure to an underlying *derivation system*, which is a simple form of logical framework introduced in Section 2.2 and furnished with examples in Section 2.4. A hiproof is only useful if it is *valid*, which means that it models a proof in the underlying derivation system. Validity is described in Section 2.3.

2.1. Syntax of hiproofs

The concrete syntax is defined by the following grammar:

$$\begin{array}{rcll}
s & ::= & a & \text{atomic} \\
 & \mid & \mathsf{id} & \text{identity} \\
 & \mid & [l]\, s & \text{labelling} \\
 & \mid & s \,;\, s & \text{sequencing} \\
 & \mid & s \otimes s & \text{tensor} \\
 & \mid & \langle\rangle & \text{empty}
\end{array}$$

Hiproofs are ranged over by s. We assume $a \in \mathcal{A}$ where $\mathcal{A}$ is the set of *atomic tactics* to be given by an underlying derivation system. Labels $l \in \mathcal{L}$ are taken from an unspecified set of identifiers. In the hiproof $[l]\, s$, the label l may be freely reused in s; labelling is not a binder. A term which has no subterm of the form $[l]\, s$ is called *label-free*. By convention, a labelled subterm extends as far right as possible, and tensor binds more tightly than sequencing.

The hiproof terms provide the structure described in the introduction. Labelling introduces named boxes (which can be nested arbitrarily deeply); sequencing composes one hiproof after another, and tensor composes two hiproofs in parallel, operating on two separate groups of goals. The identity hiproof has no effect, but is used for "wiring", to fill in structure. It can be applied to a single (atomic) goal. Finally, the empty hiproof is the vacuous proof for an empty set of goals.

The connection between the concrete syntax for hiproofs and their graphical presentation is intuitively straightforward. Labelling corresponds to labelled boxes we can zoom into and atomic tactics are boxes which cannot be zoomed; sequencing corresponds to the arrows, and tensor puts boxes side-by-side. In principle one might give a formal mapping between the graphical and linear notations, but we resist doing so here. For this paper the reader should take the linear syntax as primary and the graphical representation for illustration only.

Hiproofs have a denotational semantics [7] mentioned above; our syntax above serves as an internal language for models in that semantics. The denotational semantics justifies certain equations between terms; in particular, identity is a unit for sequencing, empty is a unit for the tensor, and tensor and sequencing are associative.

Figure 3 shows an example hiproof, and its graphic representation. Notice the role of id corresponding to the line exiting the box labelled l. We will use this example as a test case later.

$$s = ([l]\,a \;;\, b \otimes \mathsf{id})\;;\,[m]\,c$$

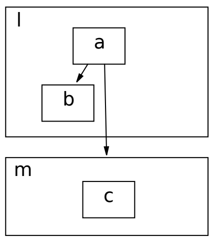

FIGURE 3. A hiproof and its graphical representation.

2.2. Derivation systems

For us, a *derivation system* is a very simple form of logical framework. It defines sets of *atomic goals* $\gamma \in \mathcal{G}$ and *atomic tactics* $a \in \mathcal{A}$. Our nomenclature is influenced by the forthcoming application in the tactic language: typically, what we call an *atomic goal* is a judgement form in the underlying derivation system and an *atomic tactic* is an elementary rule of inference.

Elementary rules of inference in the underlying derivation system can be seen as atomic tactics of the following form

$$\frac{\gamma_1 \cdots \gamma_n}{\gamma}\; a$$

which says that the tactic a, given goal γ returns goals $\gamma_1, \ldots, \gamma_n$; intuitively, given proofs of $\gamma_1, \ldots, \gamma_n$ it produces a proof of γ (as in [10]). Formally, we do not make any assumptions about the notion of proof in the underlying system.

We intend atomic tactics to stand for rules rather than rule instances. Thus for a particular atomic tactic a, there may be a family of goals γ to which it applies. Moreover, there may be a choice of the precise $\gamma_1, \ldots, \gamma_n$ which are introduced as subgoals, particularly in the case of elimination rules whose premises are not determined by their conclusion. We shall not formalise how rules and rule instances are related, but we make the important restriction that every instance of a must have the same *arity*, i.e., the same number of premises n.

By composing atomic tactics, we can produce *proofs* in the underlying derivation system. These are ordinary, non-hierarchical proof trees. We describe how these proofs are related to hiproofs next, then return in Section 2.4 to give some explicit example derivation systems.

2.3. Hiproof validation

For most of the paper, we will suppose a fixed underlying derivation system, occasionally specifying it when we want to be concrete. Given a proof in this underlying derivation system, we may want to decorate it with additional structure and abstract from concrete rule instantiation, describing it with a hiproof s. Conversely, a hiproof s may correspond to a family of underlying proofs which consist of applications of instances of the underlying atomic tactics.

This relationship is captured by *validation.* To formalise this, we work with finite lists $g \in \mathcal{G}^*$ of atomic goals $[\,\gamma_1, \ldots, \gamma_n\,]$ where $\gamma_i \in \mathcal{G}$. The empty list is written $[\,]$, and given two lists g_1 and g_2, their concatenation is written $g_1 \,^\wedge\, g_2$. We will sometimes abuse notation and write γ to stand for the single element list $[\,\gamma\,]$; we will often abuse terminology and call goal lists simply goals. To indicate that the length of g is n, we write $g : n$, and call n the *arity* of g.

We say that s *validates* proofs from g_2 to g_1, written $s \vdash g_1 \longrightarrow g_2$, when the composition of atomic tactics described by s indeed corresponds to a valid underlying derivation. Validation is defined by the rules in Figure 4.

Validation is a well-formedness check: it checks that atomic tactics are applied properly to construct a proof, and that the structural regime of hiproofs is obeyed. Notice that, although g_1 and g_2 are not determined by s, the arity restriction means that every underlying proof that s validates must have the same shape, i.e., the same underlying tree of atomic tactics. This underlying tree is known as the *skeleton* of the hiproof [7].

For tactic programming, the aspect of the skeleton which interests us is the number of goals that are solved, and the number that remain. We define the (input) *arity* of a hiproof s with $s \vdash g_1 \longrightarrow g_2$ to be n where $g_1 : n$. This is also written as $s : n$. Note that by the restriction on atomic tactics, a hiproof has a unique arity, if it has any.

$$\frac{\dfrac{\gamma_1 \cdots \gamma_n}{\gamma}\ a\ \in \mathcal{A}}{a\ \vdash\ \gamma \longrightarrow [\,\gamma_1, \ldots, \gamma_n\,]} \qquad \text{(V-Atomic)}$$

$$\mathsf{id}\ \vdash\ \gamma \longrightarrow \gamma \qquad \text{(V-Id)}$$

$$\frac{s\ \vdash\ \gamma \longrightarrow g}{[l]\, s\ \vdash\ \gamma \longrightarrow g} \qquad \text{(V-Label)}$$

$$\frac{s_1\ \vdash\ g_1 \longrightarrow g \quad s_2\ \vdash\ g \longrightarrow g_2}{s_1\ ;\ s_2\ \vdash\ g_1 \longrightarrow g_2} \qquad \text{(V-Seq)}$$

$$\frac{s_1\ \vdash\ g_1 \longrightarrow g_1' \quad s_2\ \vdash\ g_2 \longrightarrow g_2'}{s_1 \otimes s_2\ \vdash\ g_1 \,{}^\wedge g_2 \longrightarrow g_1' \,{}^\wedge g_2'} \qquad \text{(V-Tensor)}$$

$$\langle\rangle\ \vdash\ [\,] \longrightarrow [\,] \qquad \text{(V-Empty)}$$

FIGURE 4. Validation of Hiproofs.

Example 1. Suppose we have a goal γ_1 which can be proved like this:

$$\frac{\dfrac{}{\gamma_2}\,\mathrm{b} \qquad \dfrac{}{\gamma_3}\,\mathrm{c}}{\gamma_1}\,\mathrm{a}$$

Then $([l]\, a\ ;\ b \otimes \mathsf{id})\ ;\ [m]\, c\ \ \vdash\ \ \gamma_1 \longrightarrow [\,]$, and has arity 1.

2.4. Example derivation systems

To show how the abstract hiproofs may be used with real underlying derivation systems, we give three examples with different sorts of underlying goals.

Example 2. Simple *propositional logic* has formulae given by

$$P ::= \bot \mid P \to P \mid P \wedge P \mid X$$

where X stands for a propositional variable. Goals in propositional logic have the form $\Gamma \vdash P$, where Γ is a set of propositions, the assumptions. The atomic tactics are given by the well-known natural deduction rules

$$\frac{}{\Gamma, P \vdash P}\ ax \qquad \frac{\Gamma \vdash P}{\Gamma \cup \{Q\} \vdash P}\ wk \qquad \frac{\Gamma \cup \{P \to \bot\} \vdash \bot}{P}\ raa \qquad \frac{\Gamma \vdash \bot}{P}\ false$$

$$\frac{\Gamma \cup \{P\} \vdash Q}{\Gamma \vdash P \to Q}\ imp_I \qquad \frac{\Gamma \vdash P \to Q \quad \Gamma \vdash P}{\Gamma \vdash Q}\ mp$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ and_I \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}\ and_{EL} \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}\ and_{ER}$$

Atomic tactics are rule instances (e.g.,, $\mathrm{ax}_{\{X\} \vdash X}$), which are viewed as being applied backwards; they have the obvious arities.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{P \wedge Q\} \vdash P \wedge Q}\,\mathit{ax}}{\{P \wedge Q\} \vdash Q}\,\mathit{and}_{ER} \qquad \dfrac{\dfrac{\dfrac{}{\{P \wedge Q\} \vdash P \wedge Q}\,\mathit{ax}}{\{P \wedge Q\} \vdash P}\,\mathit{and}_{EL} \quad \dfrac{}{\{P \to Q \to R\} \vdash P \to Q \to R}\,\mathit{ax}}{\{P \to Q \to R, P \wedge Q\} \vdash Q \to R}\,\mathit{mp}}{\dfrac{\{P \to Q \to R, P \wedge Q\} \vdash R}{\{P \to Q \to R\} \vdash P \wedge Q \to R}\,\mathit{imp}_I}\,\mathit{mp}}{\vdash (P \to Q \to R) \to (P \wedge Q) \to R}\,\mathit{imp}_I$$

FIGURE 5. Derivation tree for $\vdash (P \to Q \to R) \to (P \wedge Q) \to R$.

Figure 5 shows an example derivation. We can use the hiproof mechanisms to impose some explicit structure on the derivation by marking up the introduction rules as *intros*, and the combination of *modus ponens* and conjunction elimination on the left as *conj-mp*, which eliminates both a conjunction and implication. The hiproof is

$$\begin{aligned} h = \quad & ([\mathit{intros}]\ \mathit{imp}_I\ ;\ \mathit{imp}_I)\ ; \\ & ([\mathit{conj\text{-}mp}]\ \mathit{mp}\ ;\ \mathsf{id} \otimes \mathit{and}_{ER})\ ; \\ & (([\mathit{conj\text{-}mp}]\ \mathit{mp}\ ;\ \mathsf{id} \otimes \mathit{and}_{EL})\ ;\ \mathit{ax} \otimes \mathit{ax}) \otimes \mathit{ax} \end{aligned}$$

and its graphical representation is shown in Figure 6.

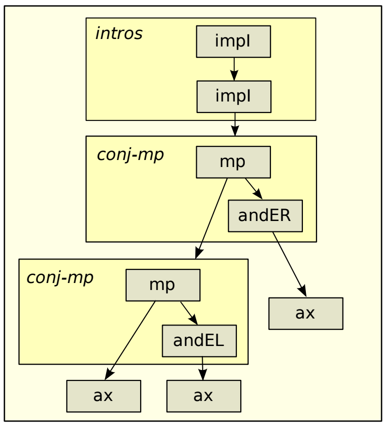

FIGURE 6. Hiproof for derivation in Figure 5.

Example 3. *Equational logic* is specified by a signature Σ, giving a set of terms $\mathcal{T}_\Sigma(X)$ over a countably infinite set of variables X, together with a set of equations E of the form $a = b$ with a, b terms. Goals in this derivation system are equations of the same form. These can be established using the following atomic tactics (where

$a, b, c, d \in \mathcal{T}_\Sigma(X)$):

$$\frac{}{a = a}\,\text{refl} \qquad \frac{a = b \quad b = c}{a = c}\,\text{trans} \qquad \frac{a = b}{b = a}\,\text{sym} \qquad \frac{a = b \quad c = d}{a[c/x] = b[d/x]}\,\text{subst}$$

Here, $a[c/x]$ denotes the term a with the variable x replaced by term c throughout. For example, the more usual substitutivity rule

$$\frac{a = b}{a[c/x] = b[c/x]}\,\text{subst}'$$

can be derived with the hiproof $h_1 = \mathit{subst} \,;\, \mathsf{id} \otimes \mathit{refl}$, whereas the usual congruence rule say for a binary operation f

$$\frac{a_1 = b_1 \quad a_2 = b_2}{f(a_1, a_2) = f(b_1, b_2)}\,\text{ctxt}$$

can be derived with the hiproof $h_2 = \mathit{subst} \,;\, (\mathit{subst} \,;\, \mathit{refl} \otimes \mathsf{id}) \otimes \mathsf{id}$.

Example 4. First-order *predicate logic* combines propositional logic and equational logic, adding universal (and existential) quantification:

$$P ::= \ldots \;\big|\; \forall X.P \;\big|\; \exists X.P$$

The rules for the quantifiers are

$$\frac{\Gamma \vdash P \quad X \notin FV(\Gamma)}{\Gamma \vdash \forall X.P}\,\mathit{all}_I \qquad \frac{\Gamma \vdash \forall X.P}{\Gamma \vdash P\,[t/X]}\,\mathit{all}_E$$

$$\frac{\Gamma \vdash P\,[t/X]}{\Gamma \vdash \exists X.P}\,\mathit{ex}_I \qquad \frac{\Gamma \vdash \exists X.P \quad \Gamma \cup \{P\} \vdash Q \quad X \notin FV(Q)}{\Gamma \vdash Q}\,\mathit{ex}_E$$

In predicate logic, we can formulate the five axioms of Presburger arithmetic, including the induction rule used in Section 1.1. With these axioms, we can formalise arithmetic proofs as in the example from Figure 2. We use boxes to label the base and step case of the induction rule; the full hiproof corresponding to the proof on the left is $\mathit{induction} \,;\, ([\mathit{Base}]\, P_3) \otimes ([\mathit{Step}]\, \mathit{trans} \,;\, P_4 \otimes (\mathit{cong} \,;\, \mathit{ax}))$. Note that the graphical representation in Figure 2 on the right is an abstraction of this hiproof, not a faithful representation.

3. Tactics and programs

The hiproofs introduced in Section 2 are static proof structures which we want to use to represent the result of executing a tactic. We now present a language of tactics which can be evaluated to do that.

3.1. Tactics

The *tactics* are the main object of study. They are defined by extending the grammar for hiproofs with four new cases:

$$
\begin{array}{rcll}
t & ::= & a \mid \mathsf{id} \mid [l]\,t \mid t\,;t \mid t \otimes t \mid \langle\rangle & \text{as for hiproofs} \\
 & \mid & t \mid t & \text{alternation} \\
 & \mid & \mathsf{assert}\ \gamma & \text{goal assertion} \\
 & \mid & T(t,\ldots,t) & \text{defined tactic, applied to tactics} \\
 & \mid & X & \text{tactic variables}
\end{array}
$$

The new cases allow proof search: alternation allows alternatives, assertion allows controlling the control flow, and defined tactics and tactic variables allow us to build up a program of possibly mutually recursive definitions.

Syntactic conventions for hiproofs are extended to tactics, with alternation having lowest precedence below sequencing, and then tensor (highest). Alternation is also associative. For a defined tactic with no arguments, we will elide the empty argument list, writing just T instead of $T()$.

For a tactic variable X and tactics t and t', the substitution of X in t by t' is written as $t\,[t'/X]$ and is defined easily in the absence of variable bindings and other technical complications; this will be used when applying a tactic below.

3.2. Tactic programs

A *tactic program* in Hitac is a set *Prog* of parametrised definitions of the form

$$T_i(X_{i,1},\ldots,X_{i,n_i}) \overset{def}{=} t_i,$$

together with a *goal matching* relation on atomic goals $\gamma \lesssim \gamma'$ which is used to define the meaning of the assertion expression. The definition set must not define any T more than once, and no label may appear more than once in the whole program.

The uniqueness requirement on labels is so that we can map a label in a hiproof back to a unique origin within the program. We will see that although labels are unique in programs, because of recursion the same label may appear many times in a hiproof which is produced when a program is evaluated.

We do not make restrictions on the goal matching relation. In some cases it may simply be an equivalence relation on goals. In equational logic, a pre-order is more natural: the matching relation can be given by instantiations of variables, so a goal given by an equation $b_1 = b_2$ matches a goal $a_1 = a_2$ if there is an instantiation $\sigma : X \to \mathcal{T}_\Sigma(X)$ such that $b_i = a_i\sigma$.

Example 5. We can give a tactic program for producing the hiproof shown in Figure 3 by defining:

$$
\begin{array}{rcl}
T_l & \overset{def}{=} & [l]\,a\,;b \otimes \mathsf{id} \\
T_m & \overset{def}{=} & [m]\,c \\
T_u & \overset{def}{=} & \mathsf{assert}\ \gamma_3\,;T_m \mid T_l\,;T_u
\end{array}
$$

If we *evaluate* the tactic T_u applied to the goal γ_1, we expect to get a hiproof similar to the one shown earlier. Intuitively, T_u repeatedly tries to solve a goal that matches γ_3 using tactic c, or uses the tactic a to split the goal, solves the first subgoal with b, and then repeats.

The next sections provide operational semantics to define a suitable notion of evaluation.

4. Big-step operational semantics

To give a meaning to Hitac programs, we will consider an operational semantics as primary. This is in contrast to some other formalised approaches which model tactics as the original LCF-style tactic programming does, i.e., using a fixed-point semantics to explain recursion. An operational semantics is desirable because we want to explain the steps used during tactic evaluation at an intensional level: this gives us a precise understanding of the internal proof state notion, which allows us to explore mechanisms for tactic debugging. At the same time, an operational semantics is closer to a direct implementation by an interpreter.

We begin by defining a big-step semantics that gives meaning to expressions without explicitly specifying the intermediate states during evaluation. Then we prove a correctness theorem which establishes that evaluation produces valid hiproofs. In Section 5 we give a small-step semantics which provides a notion of intermediate proof state.

4.1. Big-step evaluation

The big-step evaluation relation $\langle g, t\rangle \Downarrow \langle s, g'\rangle$ is defined inductively by the rules in Figure 7. The rules explain how applying a tactic t to the list of goals g results in the hiproof s and the remaining (unsolved) goals g'. A tactic t *proves* a goal, g, therefore, if $\langle g, t\rangle \Downarrow \langle s, [\,]\rangle$, for some hiproof s. The relation is defined with respect to a tactic program *Prog* containing a set of definitions.

The rules directly capture the intended meaning of tactics. To explain an example rule, (B-Label) evaluates a labelled tactic $[l]\,t$, by first evaluating the body t using the same goal γ, to get a hiproof s and remaining goals g. The result is then the labelled hiproof $[l]\,s$ and remaining goals g. Like (V-Label), this rule reflects one of the key restrictions in the notion of hiproof, namely that a box has a unique entry point, its root, accepting a single (atomic) goal.

In (B-Assert), assertion terms evaluate to identity if the goal matches, or they do not evaluate at all. Similarly, (B-Atomic) only allows an atomic tactic a to evaluate if it can be used to validate the given goal γ. Hence, failure is modelled implicitly by the lack of a target for the overall evaluation (i.e., evaluation fails iff there is a subterm $\langle g, t\rangle$ for which there is no $\langle s, g'\rangle$ it evaluates to).

Notice that tactic variables X do not reduce at all; they have to be substituted for actual tactics when a parameterised definition is applied in rule (B-Def). The

$$\frac{\dfrac{\gamma_1 \cdots \gamma_n}{\gamma}\ a \in \mathcal{A}}{\langle \gamma, a\rangle \Downarrow \langle a, [\,\gamma_1, \ldots, \gamma_n\,]\rangle} \quad (\textsf{B-Atomic})$$

$$\frac{}{\langle \gamma, \textsf{id}\rangle \Downarrow \langle \textsf{id}, \gamma\rangle} \quad (\textsf{B-Id})$$

$$\frac{\langle \gamma, t\rangle \Downarrow \langle s, g\rangle}{\langle \gamma, [l]\, t\rangle \Downarrow \langle [l]\, s, g\rangle} \quad (\textsf{B-Label})$$

$$\frac{\langle g_1, t_1\rangle \Downarrow \langle s_1, g_2\rangle \quad \langle g_2, t_2\rangle \Downarrow \langle s_2, g_3\rangle}{\langle g_1, t_1 \,;\, t_2\rangle \Downarrow \langle s_1 \,;\, s_2, g_3\rangle} \quad (\textsf{B-Seq})$$

$$\frac{\langle g_1, t_1\rangle \Downarrow \langle s_1, g_1'\rangle \quad \langle g_2, t_2\rangle \Downarrow \langle s_2, g_2'\rangle}{\langle g_1 \mathbin{^\wedge} g_2, t_1 \otimes t_2\rangle \Downarrow \langle s_1 \otimes s_2, g_1' \mathbin{^\wedge} g_2'\rangle} \quad (\textsf{B-Tensor})$$

$$\langle [\,], \langle\rangle\rangle \Downarrow \langle \langle\rangle, [\,]\rangle \quad (\textsf{B-Empty})$$

$$\frac{\langle g, t_1\rangle \Downarrow \langle s, g'\rangle}{\langle g, t_1 \mid t_2\rangle \Downarrow \langle s, g'\rangle} \quad (\textsf{B-Alt-L})$$

$$\frac{\langle g, t_2\rangle \Downarrow \langle s, g'\rangle}{\langle g, t_1 \mid t_2\rangle \Downarrow \langle s, g'\rangle} \quad (\textsf{B-Alt-R})$$

$$\frac{\gamma \lesssim \gamma'}{\langle \gamma', \textsf{assert}\ \gamma\rangle \Downarrow \langle \textsf{id}, \gamma'\rangle} \quad (\textsf{B-Assert})$$

$$\frac{T(X_1, \ldots, X_n) \overset{def}{=} t \in \mathit{Prog} \quad \langle g, t\,[t_i/X_1] \cdots [t_n/X_n]\rangle \Downarrow \langle s, g'\rangle}{\langle g, T(t_1, \ldots, t_n)\rangle \Downarrow \langle s, g'\rangle} \quad (\textsf{B-Def})$$

FIGURE 7. Big-step semantics for Hitac.

parameter list must have the right length: if a definition $T(X_1, \ldots, X_n) = t$ is not applied to n arguments it simply fails to reduce. Similarly, if the definition contains a variable X on the right-hand side t which does not occur in $\{X_1, \ldots, X_n\}$, it may (in a subterm) reduce to the uninstantiated variable X which fails to reduce.

The rules for alternation allow an angelic choice, as they allow us to pick the one of the two tactics which evaluate to a hiproof (if either of them does). If both alternatives evaluate, the alternation is non-deterministic. While this is the obvious source of non-determinism, the tensor rule also allows the (perhaps angelic) splitting of an input goal list into two halves $g_1 \mathbin{^\wedge} g_2$, including the possibility that g_1 or g_2 is the empty goal list $[\,]$.

Example 6. Returning to Examples 1 and 5, we can indeed derive $\langle \gamma_1, T_u\rangle \Downarrow \langle s', [\,]\rangle$, where $s' = ([l]\, a \,;\, b \otimes \textsf{id}) \,;\, \textsf{id} \,;\, [m]\, c$. This hiproof is equal to the one shown earlier.

4.2. Correctness property

The crucial property is *correctness* of the semantics: if a hiproof is produced, it is a valid hiproof for the claimed input and output goals.

Theorem 1 (Correctness of big-step semantics).
If $\langle g, t\rangle \Downarrow \langle s, g'\rangle$ *then* $s \vdash g \longrightarrow g'$.

Proof. By induction on the derivation of $\langle g, t\rangle \Downarrow \langle s, g'\rangle$. The reduction rules (B-Atomic) to (B-Empty) directly match the validation rules (V-Atomic) to (V-Empty). In the rules (B-Alt-L) and (B-Alt-R), the expression in the conclusion which we evaluate appears in the precondition, and hence can be validated. For (B-Assert), we can use rule (V-Id), and for (B-Def), we directly use the induction assumption. □

A further useful property is the label origin provision.

Proposition 2 (Label origin). *If t is label-free, $\langle g, t\rangle \Downarrow \langle s, g'\rangle$ and the label l appears in s, then l has a unique origin within some tactic definition $T(X_1, \ldots, X_n)$ from Prog.*

Proof. Follows immediately by the definition of program and the observation that evaluation can only introduce labels taken from the program. □

This property means that we can use labels as indexes into the program to find where a subproof was produced, which is a key motivation for labelling, and allows a source level debugging of tactical proofs. This could not be done if tactics were viewed merely as denotational functions from goals to goals.

5. Small-step semantics

Besides the big-step semantics given in the previous section, it is desirable to explain tactic evaluation using a small-step semantics. In programming languages, the usual reason for providing small-step semantics is to give meaning to non-terminating expressions. In principle we don't need to do this here, since non-terminating tactics do not produce proofs. But in practice we are interested in debugging tactics during their evaluation, including ones which may fail or loop. A big-step semantics provides no direct help for that, since the inductive relation is undefined in both cases.

Our small step-semantics provides a notion of intermediate state which has the potential to help in tactic debugging, identifying proof failure points as places where evaluation gets stuck.

5.1. Proof states

The small step semantics needs to keep track of the stages of evaluation consisting of goals partially applied to tactics, which do not correspond to a well-formed hiproof. Describing this turns out to be rather tricky: we must record which tactics have been evaluated and which not, and rearrange terms to move left over goals in subtrees out of their hierarchical boxes. The graphical intuition for the language helps here: we can imagine goals moving along the edges of the graph, in and out of the boxes, and interacting with the atomic tactics.

Using this idea, we define an evaluation relation which evolves a *proof state* configuration in each step, eventually producing a hiproof. The reduction is non-deterministic; some paths may get stuck or not terminate. Compared with the big-step semantics, the non-determinism in alternation does not need to be predicted wholly in advance, but the rules allow exploring both alternation branches of a tactic tree in parallel to find one which results in a proof.

This suggests a unified notion of proof state, where goals appear directly in the syntax with tactics. To this end, we define a compound term syntax for proof states which has hiproofs, tactics and goal lists as sublanguages. The syntax for this intermediate syntax is:

$$\begin{array}{lcl} p & ::= & g \triangleright t \;\big|\; p \triangleright t \;\big|\; s \triangleleft p \;\big|\; [l]\, p \;\big|\; p \otimes p \;\big|\; p \mid p \;\big|\; v \\ v & ::= & s \triangleleft g \end{array}$$

A proof state, p, consists of a mixture of open goals, g, active tactics, t, and successfully applied tactics, i.e., hiproofs, s. The new operators $\triangleright$ and $\triangleleft$ have higher precedence than alternation, but lower than sequencing; intuitively these represent forms of sequencing involving partially evaluated proofs. Composing proof states can be understood as connecting, or applying, the tactics of one state to the open goals of another. The notion of value, i.e., a normal form for the small-step reduction, is a fully reduced proof state with the form $s \triangleleft g'$, which symbolises the resulting left over goals g' after the hiproof s has been produced.

The general judgement form is $p \;\Rightarrow\; p'$, defined by the rules shown in Figures 8 and 9, where the second set of rules simply close reduction under congruence. A *successful* reduction starts with the application of tactic t to goal g, and ends with a value:

$$g \triangleright t \;\Rightarrow^*\; s \triangleleft g'.$$

What happens is that the goals g move through the tactic t, being transformed by atomic tactics, until (if successful) the result is a simple hiproof s and remaining goals g'. The intermediate states consist of terms $p \triangleright t$ and $s \triangleleft p$, where p is not a value but contains an incomplete proof (goal entering a tactic), or a result state where goals have yet to exit nested constructs.

To see the semantics at work, consider the tactic program from Example 5. The resulting reduction of T_u applied to the goal γ_1 is shown in Figure 10.

Note that the rules are careful to distinguish between syntactic categories for proof states, p, and the sublanguages of tactics t, hiproofs s and goals g, which can be embedded into the language of proof states. For example, in the rule (S-In-Alt), g has to be a goal and t_1, t_2 must be tactics.

The appearance of constrained subterms, and in particular, value forms $s \triangleleft g$, restricts the reduction relation and hints at evaluation order. Intuitively, joining tensors in (S-Out-Tens) only takes place after a sub-proof state has been fully evaluated. Similarly, in (S-Out-Lab), when evaluation is complete inside a box, the remaining goals are passed out on to subsequent tactics. Alternatives are only discarded in (S-Out-Alt-L) or (S-Out-Alt-R) after a successful proof has been found.

5.2. Equivalence with big-step semantics

Our main result is that the two semantics we have given coincide. This shows that the small-step semantics is indeed an accurate way to step through the evaluation

$$\frac{\dfrac{\gamma_1 \cdots \gamma_n}{\gamma} \quad a \in \mathcal{A}}{\gamma \rhd a \;\Rightarrow\; a \lhd [\,\gamma_1, \ldots, \gamma_n\,]} \qquad \textsf{(S-Atomic)}$$

$$\gamma \rhd \mathsf{id} \;\Rightarrow\; \mathsf{id} \lhd \gamma \qquad \textsf{(S-Id)}$$

$$\gamma \rhd [l]\, t \;\Rightarrow\; [l]\, \gamma \rhd t \qquad \textsf{(S-In-Lab)}$$

$$[l]\, s \lhd g \;\Rightarrow\; ([l]\, s) \lhd g \qquad \textsf{(S-Out-Lab)}$$

$$g \rhd t_1 \,;\, t_2 \;\Rightarrow\; (g \rhd t_1) \rhd t_2 \qquad \textsf{(S-In-Seq)}$$

$$(s \lhd g) \rhd t_2 \;\Rightarrow\; s \lhd (g \rhd t_2) \qquad \textsf{(S-Move-Seq)}$$

$$s_1 \lhd (s_2 \lhd g) \;\Rightarrow\; s_1 \,;\, s_2 \lhd g \qquad \textsf{(S-Out-Seq)}$$

$$g_1 \mathbin{^\wedge} g_2 \rhd t_1 \otimes t_2 \;\Rightarrow\; (g_1 \rhd t_1) \otimes (g_2 \rhd t_2) \qquad \textsf{(S-In-Tens)}$$

$$(s_1 \lhd g_1) \otimes (s_2 \lhd g_2) \;\Rightarrow\; s_1 \otimes s_2 \lhd g_1 \mathbin{^\wedge} g_2 \qquad \textsf{(S-Out-Tens)}$$

$$[\,] \rhd \langle\rangle \;\Rightarrow\; \langle\rangle \lhd [\,] \qquad \textsf{(S-Empty)}$$

$$g \rhd (t_1 \mid t_2) \;\Rightarrow\; g \rhd t_1 \mid g \rhd t_2 \qquad \textsf{(S-In-Alt)}$$

$$s_1 \lhd g \mid p_2 \;\Rightarrow\; s_1 \lhd g \qquad \textsf{(S-Out-Alt-L)}$$

$$p_1 \mid s_2 \lhd g \;\Rightarrow\; s_2 \lhd g \qquad \textsf{(S-Out-Alt-R)}$$

$$\frac{\gamma \lesssim \gamma'}{\gamma \rhd \mathsf{assert}\ \gamma' \;\Rightarrow\; \mathsf{id} \lhd \gamma} \qquad \textsf{(S-Assert)}$$

$$\frac{T(X_1, \ldots, X_n) \overset{\mathit{def}}{=} t \in \mathit{Prog}}{g \rhd T(t_1, \ldots, {}_n) \;\Rightarrow\; g \rhd t\, [t_1/X_1] \cdots [t_n/X_n]} \qquad \textsf{(S-Def)}$$

Figure 8. Small-step semantics for Hitac.

$$\frac{p \;\Rightarrow\; p'}{[l]\, p \;\Rightarrow\; [l]\, p'} \qquad \textsf{(S-Lab)}$$

$$\frac{p \;\Rightarrow\; p'}{p \rhd t \;\Rightarrow\; p' \rhd t} \quad \textsf{(S-Seq-L)} \qquad \frac{p \;\Rightarrow\; p'}{s \lhd p \;\Rightarrow\; s \lhd p'} \quad \textsf{(S-Seq-R)}$$

$$\frac{p_1 \;\Rightarrow\; p_1'}{p_1 \otimes p_2 \;\Rightarrow\; p_1' \otimes p_2} \quad \textsf{(S-Tens-L)} \qquad \frac{p_2 \;\Rightarrow\; p_2'}{p_1 \otimes p_2 \;\Rightarrow\; p_1 \otimes p_2'} \quad \textsf{(S-Tens-R)}$$

$$\frac{p_1 \;\Rightarrow\; p_1'}{p_1 \mid p_2 \;\Rightarrow\; p_1' \mid p_2} \quad \textsf{(S-Alt-L)} \qquad \frac{p_2 \;\Rightarrow\; p_2'}{p_1 \mid p_2 \;\Rightarrow\; p_1 \mid p_2'} \quad \textsf{(S-Alt-R)}$$

Figure 9. Small-step semantics for Hitac (congruence rules).

$$
\begin{array}{lll}
\multicolumn{2}{l}{\gamma_1 \rhd T_u} & \\
\Rightarrow & \gamma_1 \rhd (\mathsf{assert}\ \gamma_3 \,;\, T_m \mid T_l \,;\, T_u) & \textsf{(S-Def)} \\
\Rightarrow & \gamma_1 \rhd \mathsf{assert}\ \gamma_3 \,;\, T_m \mid \gamma_1 \rhd T_l \,;\, T_u & \textsf{(S-In-Alt)} \\
\Rightarrow & \ldots \mid (\gamma_1 \rhd T_l) \rhd T_u & \text{reduce on right, } \textsf{(S-In-Seq)} \\
\Rightarrow & \ldots \mid (\gamma_1 \rhd ([l]\, b \,;\, c \otimes \mathsf{id})) \rhd T_u & \textsf{(S-Def)} \\
\Rightarrow & \ldots \mid ([l]\, \gamma_1 \rhd b \,;\, c \otimes \mathsf{id}) \rhd T_u & \textsf{(S-In-Lab)} \\
\Rightarrow & \ldots \mid ([l]\, (\gamma_1 \rhd b) \rhd c \otimes \mathsf{id}) \rhd T_u & \textsf{(S-In-Seq)} \\
\Rightarrow & \ldots \mid ([l]\, (b \lhd [\, \gamma_2, \gamma_3 \,]) \rhd c \otimes \mathsf{id}) \rhd T_u & \textsf{(S-Atomic)} \\
\Rightarrow & \ldots \mid ([l]\, b \lhd ([\, \gamma_2, \gamma_3 \,] \rhd c \otimes \mathsf{id})) \rhd T_u & \textsf{(S-Move-Seq)} \\
\Rightarrow & \ldots \mid ([l]\, b \lhd (\gamma_2 \rhd c) \otimes (\gamma_3 \rhd \mathsf{id})) \rhd T_u & \textsf{(S-In-Tens)} \\
\Rightarrow & \ldots \mid ([l]\, b \lhd (c \lhd [\,]) \otimes (\gamma_3 \rhd \mathsf{id})) \rhd T_u & \textsf{(S-Atomic)} \\
\Rightarrow & \ldots \mid ([l]\, b \lhd (c \lhd [\,]) \otimes (\mathsf{id} \lhd \gamma_3)) \rhd T_u & \textsf{(S-Id)} \\
\Rightarrow & \ldots \mid ([l]\, b \lhd (c \otimes \mathsf{id} \lhd \gamma_3)) \rhd T_u & \textsf{(S-Out-Tens)} \\
\Rightarrow & \ldots \mid ([l]\, (b \,;\, c \otimes \mathsf{id}) \lhd \gamma_3) \rhd T_u & \textsf{(S-Out-Seq)} \\
\Rightarrow & \ldots \mid (([l]\, b \,;\, c \otimes \mathsf{id}) \lhd \gamma_3) \rhd T_u & \textsf{(S-Lab-Out)} \\
\Rightarrow & \ldots \mid ([l]\, b \,;\, c \otimes \mathsf{id}) \lhd (\gamma_3 \rhd T_u) & \textsf{(S-Move-Seq)} \\
\Rightarrow & \ldots \mid \ldots \lhd (\gamma_3 \rhd (\mathsf{assert}\ \gamma_3 \,;\, T_m \mid T_l \,;\, T_u)) & \textsf{(S-Def)} \\
\Rightarrow & \ldots \mid \ldots \lhd (\gamma_3 \rhd \mathsf{assert}\ \gamma_3 \,;\, T_m \mid [\, \gamma_3 \,] \rhd T_l \,;\, T_u) & \textsf{(S-Alt)} \\
\Rightarrow & \ldots \mid \ldots \lhd ((\gamma_3 \rhd \mathsf{assert}\ \gamma_3) \rhd T_m \mid \ldots) & \textsf{(S-In-Seq)} \\
\Rightarrow & \ldots \mid \ldots \lhd ((\mathsf{id} \lhd [\, \gamma_3 \,]) \rhd T_m \mid \ldots) & \textsf{(S-Assert)} \\
\Rightarrow & \ldots \mid \ldots \lhd (\mathsf{id} \lhd (\gamma_3 \rhd [m]\, c) \mid \ldots) & \textsf{(S-Move-Seq), (S-Def)} \\
\Rightarrow & \ldots \mid \ldots \lhd (\mathsf{id} \lhd ([m]\, \gamma_3 \rhd c) \mid \ldots) & \textsf{(S-In-Lab)} \\
\Rightarrow & \ldots \mid \ldots \lhd (\mathsf{id} \lhd ([m]\, c \lhd [\,]) \mid \ldots) & \textsf{(S-Atomic)} \\
\Rightarrow & \ldots \mid \ldots \lhd (\mathsf{id} \lhd (([m]\, c) \lhd [\,]) \mid \ldots) & \textsf{(S-Out-Lab)} \\
\Rightarrow & \ldots \mid \ldots \lhd ((\mathsf{id} \,;\, [m]\, c) \lhd [\,] \mid \ldots) & \textsf{(S-Out-Seq)} \\
\Rightarrow & \ldots \mid ([l]\, b \,;\, c \otimes \mathsf{id}) \lhd ((\mathsf{id} \,;\, [m]\, c) \lhd [\,]) & \textsf{(S-Out-Alt-L)} \\
\Rightarrow & \ldots \mid (([l]\, b \,;\, c \otimes \mathsf{id}) \,;\, \mathsf{id} \,;\, [m]\, c) \lhd [\,] & \textsf{(S-Out-Seq)} \\
\Rightarrow & (([l]\, b \,;\, c \otimes \mathsf{id}) \,;\, \mathsf{id} \,;\, [m]\, c) \lhd [\,] & \textsf{(S-Out-Alt-R)}
\end{array}
$$

FIGURE 10. Reduction of tactic from Example 5. The steps name the major rule applied at each point.

of tactics; moreover, reduction using the small-step relation does not limit the language.

Theorem 3 (Completeness of small-step semantics).
If $\langle g, t \rangle \Downarrow \langle s, g' \rangle$, *then* $g \rhd t \ \Rightarrow^* \ s \lhd g'$

Proof. Straightforward induction on big-step derivation. Each big step reduction rule can be given by a sequence of small-step reductions. For example, for (B-Seq):

$$
\begin{array}{llllll}
g_1 \rhd t_1 \,;\, t_2 & \Rightarrow & (g_1 \rhd t_1) \rhd t_2 & \Rightarrow & (s_1 \lhd g_2) \rhd t_2 & \\
 & \Rightarrow & s_1 \lhd (g_2 \rhd t_2) & \Rightarrow & s_1 \lhd (s_2 \lhd g_3) & \Rightarrow \quad s_1 \,;\, s_2 \lhd g_3 .
\end{array}
$$

Other rules are similar. □

Soundness is the harder direction of the equivalence proof. We make use of an auxiliary lemma given later below, which needs to be proved using a stronger induction than merely induction over the inductive relation.

Theorem 4 (Soundness of small-step semantics).
If $g \triangleright t \;\Rightarrow^*\; s \triangleleft g'$ then $\langle g, t\rangle \Downarrow \langle s, g'\rangle$.

Proof. By induction on the length of the derivation. Assume the derivation has length n, and the proposition holds for all derivations of length $m < n$. Now do a case distinction on t, considering all possible rules starting a reduction, using Lemma 5 for the remainder of the reduction if necessary.

As an example, consider $t = t_1 \;;\; t_2$. All possible reductions from $g \triangleright t_1 \;;\; t_2$ have to start with (S-In-Seq) (other rules are not possible), and are hence of the form $g \triangleright t_1 \;;\, t_2 \;\Rightarrow\; (g \triangleright t_1) \triangleright t_2 \;\Rightarrow^*\; s \triangleleft g'$. By Lemma 5 (case 3), we have s_1, s_2, g_1 s.t. $s = s_1 \;;\; s_2$, and $g \triangleright t_1 \;\Rightarrow^*\; s_1 \triangleleft g_1$ and $g_1 \triangleright t_1 \;\Rightarrow^*\; s_2 \triangleleft g'$ in m steps with $m < n - 1$. By the induction assumption (since $m < n$), we have $\langle g, t_1\rangle \Downarrow \langle s_1, g_1\rangle$, and $\langle g_1, t_1\rangle \Downarrow \langle s_2, g'\rangle$. By rule (B-Seq), we get the reduction $\langle g, t_1 \;;\; t_2\rangle \Downarrow \langle s_1 \;;\; s_2, g'\rangle$ as required. Other cases are similar using other parts of Lemma 5. □

Lemma 5 (Structure preservation).

1. *If $[l]\,p \;\Rightarrow^*\; s \triangleleft g$ then for some s', $s = [l]\,s'$ and there exists a reduction $p \;\Rightarrow^*\; s' \triangleleft g$ with strictly shorter length.*
2. *If $p_1 \otimes p_2 \;\Rightarrow^*\; s \triangleleft g$ and $p_1, p_2 \neq \langle\rangle$, then for some s_1, s_2, g_1 and g_2, we have $s = s_1 \otimes s_2$ and $g = g_1 \,^\wedge\, g_2$ and there exist reductions $p_i \;\Rightarrow^*\; s_i \triangleleft g_i$ with strictly shorter lengths.*
3. *If $p \triangleright t \;\Rightarrow^*\; s \triangleleft g$ where p is not a goal, then for some s_1, s_2, g_1, we have $s = s_1 \;;\; s_2$ and there exist reductions $p \;\Rightarrow^*\; s_1 \triangleleft g_1$ and $g_1 \triangleright t \;\Rightarrow^*\; s_2 \triangleleft g$ each with strictly shorter length.*
4. *If $s \triangleleft p \;\Rightarrow^*\; s' \triangleleft g$ where p is not a goal, then for some s_2, we have $s' = s \;;\; s_2$, and there exists a reduction $p \;\Rightarrow^*\; s_2 \triangleleft g$ of strictly shorter length.*
5. *If $p_1 \mid p_2 \;\Rightarrow^*\; s \triangleleft g$ then there exists a strictly shorter reduction of $p_1 \;\Rightarrow^*\; s \triangleleft g$ or of $p_2 \;\Rightarrow^*\; s \triangleleft g$.*

Proof. The lemma is proven by simultaneous induction on the lengths of sequences involved, and a case distinction on the constructors of the term starting the reduction.

For example, in case (3), we have to consider all possible reductions from $p \triangleright t$. The rules (S-In-Seq), (S-Id), (S-Assert) and others are not applicable, as p cannot be a goal; the only two rules which can start a reduction are (S-Move-Seq) and (S-Seq-L). For (S-Move-Seq), we have $p = s' \triangleleft g'$, and $(s' \triangleleft g') \triangleright t \;\Rightarrow\; s' \triangleleft (g' \triangleleft t) \;\Rightarrow^*\; s \triangleleft g$. We can apply the induction assumption, case (4), to $s' \triangleleft (g' \triangleleft t) \;\Rightarrow^*\; s \triangleleft g$ (in $n-1$ steps), and obtain s_2 such that $s = s' \;;\; s_2$ and $g' \triangleright t \;\Rightarrow\; s_2 \triangleleft g$ in $m < n-1$ steps. Then we have s_1 as s', s_2 as was, g_1 as g', with $p \;\Rightarrow^*\; s_1 \triangleleft g_1$ in 0 steps, and $g_1 \triangleright t \;\Rightarrow^*\; s_2 \triangleleft g$ in $m < n - 1$ steps as required.

For (S-Seq-L), we have $p \rhd t \ \Rightarrow\ p' \rhd t \ \Rightarrow^*\ s \lhd g$, with $p \ \Rightarrow\ p'$. By induction assumption on $p' \rhd t \ \Rightarrow^*\ s \lhd g$, we obtain s_1, s_2, g_1 such that $s = s_1 \ ; \ s_2$ and $p' \ \Rightarrow^*\ s_1 \lhd g_1$, $g_1 \rhd t \ \Rightarrow^*\ s_2 \lhd g$ in length m with $m < n - 1$. Then s_1, s_2, g are as required, with $p \rhd t \ \Rightarrow\ p' \rhd t \ \Rightarrow^*\ s_1 \lhd g_1$ in length at most $m + 1 < n - 1 + 1 = n$ as required. □

Theorems 1 and 4 together show that correctness also holds for the small step semantics:

Corollary 6 (Correctness of small-step semantics).
If $g \rhd t \ \Rightarrow^*\ s \lhd g'$ *then* $s \ \vdash \ g \longrightarrow g'$.

6. Proofs that fail

Both operational semantics explain the way that tactics successfully evaluate to proofs, but what happens when things go wrong? We have suggested that our tactic language should allow us to examine buggy proofs in detail to understand points of failure. Informally, a proof fails if it cannot reduce successfully. This may happen because the reduction is stuck on a non-value, because it only reduces to failing proofs, or because it only has non-terminating reduction sequences. In this section we study failing proofs by characterising a set of reduction sequences that cannot succeed. We consider the small step semantics and classify failure points during reduction, as they become known. We call such terms *futile*. Futile terms may be pruned from the search space during evaluation, and they can be investigated to explain failure. This also shows the necessity of a small-step semantics; the big-step semantics does not allow this kind of analysis.

Figure 11 gives an inductive definition of futile terms, by induction over the structure of proofstates p in the small-step semantics from Section 5. We write fut p to indicate that p is futile.

The main base cases for futility are the atomic tactic case, where there is no matching atomic instance of the given tactic a to the input goal γ', in rule (F-Atomic), and the assertion case (F-Assert), when the assertion fails. The other base cases are the five rules for for mis-matched numbers of goals; for example, (F-Empty-Tac) says that the empty tactic cannot solve a list of goals with length greater than 0. Unbound variables or improper applications of definitions are also futile. The remaining structural rules lift futility to larger contexts; notice in particular that a tensor proofstate is futile if either branch is futile, while an alternation is only futile if both branches are, reflecting the fact that to reduce a tensor, both arguments need to reduce, whereas to reduce an alternation, we can reduce either argument.

Example 7. Suppose we give a tactic program which aims at the hiproof shown in Figure 3, by defining:

$$T_l \ \overset{def}{=}\ [l]\, a \ ; b$$

$$\frac{\text{for all } \gamma,\ \dfrac{\gamma_1 \cdots \gamma_n}{\gamma}\ a \in \mathcal{A} \quad \text{implies} \quad \gamma \neq \gamma'}{\mathsf{fut}\ \gamma' \rhd a}\ (\textsf{F-Atomic})$$

$$\frac{\gamma \not\lesssim \gamma'}{\mathsf{fut}\ \gamma \rhd \mathsf{assert}\ \gamma'}\ (\textsf{F-Assert})$$

$$\frac{g : n \quad n \neq 1}{\mathsf{fut}\ g \rhd a}\ (\textsf{F-Atomic-N}) \qquad \frac{g : n \quad n \neq 1}{\mathsf{fut}\ g \rhd \mathsf{id}}\ (\textsf{F-Id-N})$$

$$\frac{g : n \quad n \neq 1}{\mathsf{fut}\ g \rhd [l]\, p}\ (\textsf{F-Lab-N}) \qquad \frac{\mathsf{fut}\ g \rhd t_1}{\mathsf{fut}\ g \rhd t_1\ ;\ t_2}\ (\textsf{F-Seq})$$

$$\frac{g : n \quad n > 0}{\mathsf{fut}\ g \rhd \langle\rangle}\ (\textsf{F-Empty-Tac}) \qquad \frac{}{\mathsf{fut}\ g \rhd X}\ (\textsf{F-Var})$$

$$\frac{\mathsf{fut}\ p}{\mathsf{fut}\ [l]\, p}\ (\textsf{F-Lab}) \qquad \frac{\mathsf{fut}\ p_1 \quad \mathsf{fut}\ p_2}{\mathsf{fut}\ p_1 \mid p_2}\ (\textsf{F-Alt})$$

$$\frac{\mathsf{fut}\ p_1}{\mathsf{fut}\ p_1 \otimes p_2}\ (\textsf{F-Tens-L}) \qquad \frac{\mathsf{fut}\ p_2}{\mathsf{fut}\ p_1 \otimes p_2}\ (\textsf{F-Tens-R})$$

$$\frac{\mathsf{fut}\ p}{\mathsf{fut}\ p \rhd t}\ (\textsf{F-Seq-L}) \qquad \frac{\mathsf{fut}\ p}{\mathsf{fut}\ s \lhd p}\ (\textsf{F-Seq-R})$$

$$\frac{T \notin \mathit{Prog} \text{ or } T(X_1, \ldots, X_n) \overset{def}{=} t \in \mathit{Prog} \text{ for } n \neq m}{\mathsf{fut}\ g \rhd T(t_1, \ldots, t_m)}\ (\textsf{F-Def})$$

FIGURE 11. Characterisation of failing proofs.

and T_m and T_u as they were in Example 5. Notice that the program is now buggy because T_l feeds two subgoals into the 1-ary atomic tactic b.

Because of this, we have fut $[\gamma_2, \gamma_3] \rhd b$, for example. The attempted reduction of $\gamma_1 \rhd T_1$ is not immediately futile, but can reduce to $[l]\, \gamma_1 \rhd a \rhd b$ and then to $[l]\, (a \lhd [\gamma_2, \gamma_3]) \rhd b$. However, the next step by (S-Move-Seq) is to $[l]\, a \lhd ([\gamma_2, \gamma_3] \rhd b)$ which is futile by (F-Seq-R) and (F-Lab).

The following results establish that this notion of futile behaves as desired.

Lemma 7 (Futile proofs will fail). *If* fut p, *then there is no value* v *such that* $p \Rightarrow^* v$.

Proof. By induction on the structure of p. For base cases, note that fut p implies that p is stuck. For the inductive step, we apply Lemma 5 to show the congruence of futility. □

Observe that, as one would hope, Lemma 7 implies that values are not futile, so we never have $\mathsf{fut}\ v$ for any value v. However, unlike values, futile terms may still contain reductions. The next lemma shows that once the reduction has reached a futile term, then further reductions cannot get us out of this state.

Lemma 8 (Preservation of futility). *If* $\mathsf{fut}\ p$ *and* $p \Rightarrow p'$*, then* $\mathsf{fut}\ p'$.

Proof. By case analysis on the small-step semantics. □

Lemmas 7 and 8 tell us that futility behaves as expected, but do not tell us that we have an optimal characterisation of futility. One might hope for a converse to Lemma 7, so we could see that a term is futile as soon as possible, but in the presence of recursive tactics this would be hard to achieve.

Finally, the main theorem is a progress property for our small-step semantics. Every proofstate must either have successfully terminated, must be capable of further reductions, or must be futile. The last two cases may overlap, as we have pointed out, but we find the contextual closure of futility more convenient to use than the more restrictive notion of stuckness.

Theorem 9 (Progress). *For all* p*, either*

- *for some value* v*,* $p \equiv v$*, or*
- p *is a non-value and for some* p'*,* $p \Rightarrow p'$*, or*
- $\mathsf{fut}\ p$.

Proof. By induction on the structure of p. □

By the definition of futility, we can isolate the futile subterms in a futile term and explain their stuck points by looking at the base cases for futility. Futility also allows early pruning of the search space when more than one reduction is possible. However, stuck subterms are not the only way tactic programs can misbehave: they might also loop forever, for example repeatedly applying atomic rules (consider repeated application of rule sym from Example 3). In general there can be no way to detect looping, although approximate tests would be useful in practice.

7. Tactic programming

Tactics as above are procedures which produce hiproofs. To help with writing tactics, most theorem provers provide *tacticals*, which are tactic functionals or higher-order tactics. These combine existing tactics into new ones. Our language is more restrictive, but does admit first-order tacticals.

$$
\begin{array}{ll}
[\,\gamma_2, \gamma_2\,] \triangleright \mathtt{ALL}(b) & \\
\;\Rightarrow [\,\gamma_2, \gamma_2\,] \triangleright (b \otimes \mathtt{ALL}(b) \mid \langle\rangle) & \textsf{(S-Def)} \\
\;\Rightarrow [\,\gamma_2, \gamma_2\,] \triangleright b \otimes \mathtt{ALL}(b) \mid [\,\gamma_2, \gamma_2\,] \triangleright \langle\rangle & \textsf{(S-In-Alt)} \\
\;\Rightarrow (\gamma_2 \triangleright b) \otimes (\gamma_2 \triangleright \mathtt{ALL}(b)) \mid \ldots & \textsf{(S-In-Tens)} \\
\;\Rightarrow (b \triangleleft [\,]) \otimes (\gamma_2 \triangleright (b \otimes \mathtt{ALL}(b) \mid \langle\rangle)) \mid \ldots & \textsf{(S-Atomic), (S-Def)} \\
\;\Rightarrow \ldots \otimes (\gamma_2 \triangleright b \otimes \mathtt{ALL}(b) \mid \gamma_2 \triangleright \langle\rangle) \mid \ldots & \textsf{(S-In-Alt)} \\
\;\Rightarrow \ldots \otimes ((\gamma_2 \triangleright b) \otimes ([\,] \triangleright \mathtt{ALL}(b)) \mid \ldots) \mid \ldots & \textsf{(S-In-Tens)} \\
\;\Rightarrow \ldots \otimes ((b \triangleleft [\,]) \otimes ([\,] \triangleright (b \otimes \mathtt{ALL}(b) \mid \langle\rangle)) \mid \ldots) \mid \ldots & \textsf{(S-Atomic),(S-Def)} \\
\;\Rightarrow \ldots \otimes (\ldots \otimes ([\,] \triangleright b \otimes \mathtt{ALL}(b) \mid [\,] \triangleright \langle\rangle) \mid \ldots) \mid \ldots & \textsf{(S-In-Alt)} \\
\;\Rightarrow \ldots \otimes (\ldots \otimes (\ldots \mid \langle\rangle \triangleleft [\,]) \mid \ldots) \mid \ldots & \textsf{(S-Empty)} \\
\;\Rightarrow \ldots \otimes ((b \triangleleft [\,]) \otimes (\langle\rangle \triangleleft [\,]) \mid \ldots) \mid \ldots & \textsf{(S-Out-Alt-R)} \\
\;\Rightarrow \ldots \otimes (b \triangleleft [\,]) \mid \ldots \mid \ldots & \textsf{(S-Out-Tens)},\ b \otimes \langle\rangle = b \\
\;\Rightarrow (b \triangleleft [\,]) \otimes (b \triangleleft [\,]) \mid \ldots & \textsf{(S-Out-Alt-L)} \\
\;\Rightarrow b \otimes b \triangleleft [\,] \mid \ldots & \textsf{(S-Out-Tens)} \\
\;\Rightarrow b \otimes b \triangleleft [\,] & \textsf{(S-Out-Alt-L)}
\end{array}
$$

FIGURE 12. Example reduction using the ALL(b) tactic.

7.1. Tacticals

The simplest examples of tacticals are the alternation and sequencing operations for tactics. Theorem provers like Edinburgh LCF, Isabelle, HOL or Coq provide more advanced patterns of applications; we concentrate on a few characteristic examples.

We write tacticals as parameterised tactic definitions in our tactic program *Prog*. A simple example is $\mathtt{TRY}(t)$, which applies t and returns the result if it succeeds, or it does nothing:

$$\mathtt{TRY}(X) \;\overset{def}{=}\; X \mid \mathsf{id}$$

On the other hand, we may want to apply t to as many goals as possible (possibly zero), using $\mathtt{ALL}(t)$:

$$\mathtt{ALL}(X) \;\overset{def}{=}\; X \otimes \mathtt{ALL}(X) \mid \langle\rangle$$

Figure 12 shows how $\mathtt{ALL}(b)$ reduces the goal $[\,\gamma_2, \gamma_2\,]$. A more involved example applies the tactic t as often as possible. It uses ID, the 'polymorphic identity':

$$
\begin{array}{rcl}
\mathtt{ID} & \overset{def}{=} & \mathtt{ALL}(\mathsf{id}) \\
\mathtt{REPEAT}(X) & \overset{def}{=} & X \,;\, \mathtt{REPEAT}(X) \mid \mathtt{ID}
\end{array}
$$

ID applied to any goal $g : n$ reduces to id^n, the n-fold tensor of id:

$$[\,\gamma_1, \ldots \gamma_n\,] \triangleright \mathtt{ID} \;\Rightarrow^*\; \mathsf{id}^n \triangleleft [\,\gamma_1, \ldots \gamma_n\,] \qquad (1)$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\vdash A \Longrightarrow (B \Longrightarrow A) \ ; \texttt{IMP-INTRO}} & \\
\Rightarrow & \vdash A \Longrightarrow (B \Longrightarrow A) \rhd (\mathit{imp}_I \ ; \texttt{REPEAT}(\mathit{imp}_I) \mid \texttt{ID}) & \textsf{(S-Def)} \\
\Rightarrow & \vdash A \Longrightarrow (B \Longrightarrow A) \rhd \mathit{imp}_I \ ; \texttt{REPEAT}(\mathit{imp}_I) \mid \vdash A \Longrightarrow (B \Longrightarrow A) \rhd \texttt{ID} & \textsf{(S-In-Alt)} \\
\Rightarrow & (\vdash A \Longrightarrow (B \Longrightarrow A) \rhd \mathit{imp}_I) \rhd \texttt{REPEAT}(\mathit{imp}_I) \mid \ldots & \textsf{(S-In-Seq)} \\
\Rightarrow & (\mathit{imp}_I \lhd \{A\} \vdash B \Longrightarrow A) \rhd \texttt{REPEAT}(\mathit{imp}_I) \mid \ldots & \textsf{(S-Atomic)} \\
\Rightarrow & \mathit{imp}_I \lhd (\{A\} \vdash B \Longrightarrow A \rhd \texttt{REPEAT}(\mathit{imp}_I)) \mid \ldots & \textsf{(S-Move-Seq)} \\
\Rightarrow & \mathit{imp}_I \lhd (\{A\} \vdash B \Longrightarrow A \rhd (\mathit{imp}_I \ ; \texttt{REPEAT}(\mathit{imp}_I) \mid \texttt{ID})) \mid \ldots & \textsf{(S-Def)} \\
\Rightarrow & \mathit{imp}_I \lhd (\{A\} \vdash B \Longrightarrow A \rhd \mathit{imp}_I \ ; \texttt{REPEAT}(\mathit{imp}_I) \mid \{A\} \vdash B \Longrightarrow A \rhd \texttt{ID}) \mid \ldots & \\
 & & \textsf{(S-In-Alt)} \\
\Rightarrow & \mathit{imp}_I \lhd ((\{A\} \vdash B \Longrightarrow A \rhd \mathit{imp}_I) \rhd \texttt{REPEAT}(\mathit{imp}_I) \mid \ldots) \mid \ldots & \textsf{(S-In-Seq)} \\
\Rightarrow & \mathit{imp}_I \lhd ((\mathit{imp}_I \lhd \{A, B\} \vdash A) \rhd \texttt{REPEAT}(\mathit{imp}_I) \mid \ldots) \mid \ldots & \textsf{(S-Atomic)} \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \lhd (\{A, B\} \vdash A \rhd \texttt{REPEAT}(\mathit{imp}_I)) \mid \ldots) \mid \ldots & \textsf{(S-Move-Seq)} \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \lhd (\{A, B\} \vdash A \rhd (\mathit{imp}_I \ ; \texttt{REPEAT}(\mathit{imp}_I) \mid \texttt{ID})) \mid \ldots) \mid \ldots & \textsf{(S-Def)} \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \lhd (\{A, B\} \vdash A \rhd \mathit{imp}_I \ ; \texttt{REPEAT}(\mathit{imp}_I) \mid \{A, B\} \vdash A \rhd \texttt{ID}) \mid \ldots) \mid \ldots & \\
 & & \textsf{(S-In-Alt)} \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \lhd (\ldots \mid \mathsf{id} \lhd \{A, B\} \vdash A) \mid \ldots) \mid \ldots & \text{reduction of } \texttt{ID} \ (1) \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \lhd (\mathsf{id} \lhd \{A, B\} \vdash A) \mid \ldots) \mid \ldots & \textsf{(S-Out-Alt-R)} \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \ ; \mathsf{id} \lhd \{A, B\} \vdash A \mid \ldots) \mid \ldots & \textsf{(S-Out-Seq)} \\
\Rightarrow & \mathit{imp}_I \lhd (\mathit{imp}_I \ ; \mathsf{id} \lhd \{A, B\} \vdash A) \mid \ldots & \textsf{(S-Out-Alt-L)} \\
\Rightarrow & \mathit{imp}_I \ ; \mathit{imp}_I \ ; \mathsf{id} \lhd \{A, B\} \vdash A \mid \ldots & \textsf{(S-Out-Seq)} \\
\Rightarrow & \mathit{imp}_I \ ; \mathit{imp}_I \ ; \mathsf{id} \lhd \{A, B\} \vdash A & \textsf{(S-Out-Alt-L)}
\end{array}$$

FIGURE 13. Example reduction using the IMP-INTRO tactic.

An application of REPEAT is a tactic to strip away all implications in the logic PL, shown in Example 2. This is defined as:

$$\texttt{IMP-INTRO} \quad \stackrel{\mathit{def}}{=} \quad \texttt{REPEAT}(\mathit{imp}_I)$$

An example reduction is shown in Figure 13.

A similar, but more powerful tactic for predicate logic (see Example 4) is

$$\texttt{INTROS} \quad \stackrel{\mathit{def}}{=} \quad \texttt{REPEAT}(\mathit{imp}_I \mid \mathit{and}_I \mid \mathit{all}_I)$$

which reduces all conjunctions, implications and universal quantifiers.

7.2. Implementation aspects: non-determinism

Both semantics we have given are deliberately non-deterministic: a tactic t applied to a goal g may evaluate to different pairs of hiproof s and remaining goals g'. There can be many unwanted reductions (those producing no proof, or an incomplete proof) along with successful ones (which produce the desired proof). For example, $\gamma \rhd \texttt{REPEAT}(t)$ can always reduce to $\mathsf{id} \lhd \gamma$, making no progress on solving γ.

Non-determinism has advantages: the tensor splitting allows a tactic such as $\texttt{ALL}(b) \otimes \texttt{ALL}(c)$ to solve the goal $\gamma_2 \otimes \gamma_2 \otimes \gamma_3$ by splitting the tensor judiciously:

$$\begin{array}{rcl} [\,\gamma_2, \gamma_2, \gamma_3\,] \triangleright \texttt{ALL}(b) \otimes \texttt{ALL}(c) & \Rightarrow & ([\,\gamma_2, \gamma_2\,] \triangleright \texttt{ALL}(b)) \otimes (\gamma_3 \triangleright \texttt{ALL}(c)) \\ & \Rightarrow^* & (b \otimes b \triangleleft [\,]) \otimes (c \triangleleft [\,]) \\ & \Rightarrow & b \otimes b \otimes c \triangleleft [\,] \end{array}$$

Thus, a non-deterministic semantics gives us a precise understanding of the full power of the language. We can subsequently develop deterministic variations of the semantics, and investigate whether they retain the same expressiveness.

Clearly, however, the non-deterministic semantics is not pleasant to implement directly: it requires keeping track of all possible reductions, and selecting the right ones after the fact. There is a rapid blow up in the number of possible reductions. Although it is possible in principle to implement this, it may not be desirable: current tactical languages avoid this by severely reducing nondeterminism or providing mechanisms for controlling search explicitly. For example, the traditional alternation tactical (`ORELSE` in the LCF family) selects the first alternative if it is successful, and the second otherwise. Alternatively, one may search in a particular order (perhaps lazily, i.e., only evaluating alternatives when needed) and allow backtracking. This is how the system Isabelle, for example, handles the non-determinism arising from the ambiguities of higher-order unification: when a rule matched on the proof state produces more than one possible unifier, the first match is chosen, but users can step through the matches manually [22].

In our language we could achieve a simple control mechanism by building a futility test into the tactic engine, and rewriting eagerly:

$$\begin{array}{rcll} p_1 \mid p_2 & \longrightarrow & p_1 & \text{when } \mathsf{fut}\, p_2 \\ p_1 \mid p_2 & \longrightarrow & p_2 & \text{when } \mathsf{fut}\, p_1 \end{array}$$

If we additionally specify left-right evaluation order, we would only use the second rule. However, these rewrites lose record of the failure points which might be useful for debugging. A more complete treatment would require exception-like mechanisms to propagate failure.

Tensor non-determinism might be tamed by introducing some form of a “goal stack” containing remaining goals to be solved, and evaluating tensor tactic applications sequentially. In our setting this requires some “rewiring” to ensure that goals are treated at the right nesting level.

The heart of the problem here (from our point of view) is the crucial difference between hiproofs and tactics. Because of alternation and repetition, a tactic can evaluate to many different hiproofs, each of which can validate different proofs, so we cannot extend the validity notion directly, even for (statically) checking arities. Tactics with predictable arities would give enough information for tensor splitting, i.e., predicting the distribution of subgoals in the proof task.

This is one of the things that makes tactic programming difficult. It is interesting to investigate whether richer static type systems could help; perhaps there is a useful intermediate ground between ordinary untyped tactics and what could be called "certified tactic programming" [3, 23], where tactics are shown to construct correct proofs using dependent typing. Unfortunately we must defer further study of this, and of deterministic evaluation schemes for our language, to future work.

8. Related work

Our work with hierarchical structure in the novel form of hiproofs is unique, although there are many related developments on tactic language semantics and structured proofs elsewhere. We highlight some recent and closely connected developments. For more references to related notions of structured proof, see [7]. One basic point to bear in mind when comparing Hitac to existing systems is that Hitac has been devised as a formalism in which to study the mechanics of interactive tactical theorem proving on a generic level, independent of both an underlying prover and logic; thus, the relevant question is whether our formalism is powerful enough to capture the necessary essentials of existing systems.

Since Edinburgh LCF [10], tactic programming has used a full-blown programming language for defining new tactics, as is also done in the modern HOL systems, such as Isabelle [20, 22], HOL [28] and HOL Light [11]. In these systems, tactics are written as functions in variants of the functional programming language ML, constructing proof procedures by composing basic inferences. The direct way of understanding such tactics is as the functions they define over proof states, suggesting a denotational fixed point semantics. This semantics was investigated for the foundational tactic language Angel [19], which, as its name suggests, is centred on the idea of angelic nondeterminism. Angel was studied further in the extended language ArcAngel [21] which added alternation and recursion to Angel to provide a calculus for program refinement. An advantage of the denotational semantics is that a number of equational laws about tactics to be derived.

In the last decade, Isabelle has moved from ML-based theory programming to programming in the custom proof language *Isar*, which allows users to structure proofs in a declarative style which follows the logical argument. Declarative proofs are checked using an operational mechanism based on an evaluation by an abstract machine [30], although Isar itself currently has only basic facilities for writing tactics and still relies on underlying ML code to do the heavy lifting.

One system that bears a closer structural resemblance to the notion of a hiproof is Nuprl [2, 18]. In Nuprl, a proof tree is built with inference rules, and tactics build inference rules from primitive rules; in contrast to LCF-style tactics, tactics appear as high-level inference rules in the proof tree. The Nuprl proof editor allows navigating the tree, expanding previously applied tactics, or refining a proof by applying a tactic on a node; thus, in Nuprl the proof tree is manipulated more directly as opposed to LCF-style systems, where the series of inferences leading to the current proof state is mostly handled behind the scenes. Nuprl can be captured quite directly in our framework: a proof tree can modelled by a hiproof, and tactics (written in Hitac) would always have a label with their name, such that expanding them produces a label designating the hiproof constructed by applying this tactic. Of course, writing tactics in a Turing equivalent programming language such as Nurpl's ML is both more powerful and hence more difficult to reason about than using Hitac.

Coq offers the power of OCaml for tactic programming, but also provides a dedicated functional language for writing tactics, $\mathcal{L}_{tac}$, designed by Delahaye [6]. This has the advantage of embedding directly in the Coq proof language, and offers powerful matching on the proof context. However, Delahaye did not formalise an evaluation semantics or describe a tactic tracing mechanism.

There are several attempts to encode tactical languages in logical frameworks more powerful than programming languages: Felty using λProlog [8], Appel and Felty using Twelf [3], or Aboul-Housn [1] embedding tactics into theorems and proofs without fixing on a particular meta-logic. The latter has the advantage that users do not need a different language to write tactics, but comes with a loss of expressiveness. Jojgov and Geuvers [13] define a calculus of tactics based on higher-order abstract syntax, which has an operational semantics and can handle phenomena such as meta-variables, but does not account for hierarchical structure, and, because it uses a particular notion of terms and types, cannot express tactics in systems which are not compatible.

Kirchner [14] appears to have been the first to consider formally describing a small-step semantics for tactic evaluation, impressively attempting to capture the behaviour of both Coq and PVS within similar semantic frameworks. He defines a judgement $e/\tau \to e'/\tau'$, which operates on a tactic expression e and proof context τ, to produce a simpler expression and updated context. So, unlike our simpler validation-based scheme, state based side-effecting of a whole proof is possible. However, the reduction notion is very general and the definitions for Coq and PVS are completely system-specific using semantically defined operations on proof contexts; there is a big gulf between providing these definitions and proving them correct. In subsequent work, Kirchner focused on PVS [15].

Tinycals [24] is another small-step tactic language, implemented in Matita [4]. The main motivation is to allow stepping inside tactics to extend step-by-step checking of the proof level. Interactive step-by-step checking of proofs was popularised by Proof General [5] and followed by several other systems including Matita. Aside from Matita, other common systems do not allow single-stepping of defined tactics using their source (i.e., source-level debugging); instead, forms of *tracing* are possible by interrupting the tactic engine after a step and displaying the current state. Tinycals allows tracing linked back to the tactic expression, also showing the user information about remaining goals and backtracking points. The Tinycals language allows nested proof structure to be expressed in tactics, like hiproofs (and the language is also somewhat reminiscent of Isar's style), but there is no naming for the nested structure in either case.

The ΩMEGA system [26] is in the same spirit as Nuprl, Coq, or Isabelle, but also integrates proof planning. It represents proofs and plans by the proof plan data structure PDS, which supports operations to abstract over details, and expand, akin to the zooming facilities of hiproof. An old interface for ΩMEGA called *LΩUI* [27] allowed the user to manipulate the proof tree directly, but the more recent PLATΩ interface [29] instead requires users to write formalised proofs in textual form using the TeXmacs editor (in this respect it is similar to the Isar approach, where Isabelle can produce LaTeX source code from formally checked proofs). The hierarchical structure of PDS is nicely captured by hiproofs, but another similarity is that ΩMEGA uses OMDOC internally as document and proof exchange format. OMDOC [16] is "a content markup scheme for collections of mathematical documents", which also offers facilities to express proofs; it is similar to hiproofs in that it is parametric over an underlying logic, but it aims at interchange between systems, is much broader in scope, and does not include a tactical language.

9. Conclusions and future work

This paper introduced a tactic language, Hitac, for constructing hierarchical proofs. We believe that hierarchical proofs offer the chance for better management of formal proofs, in particular, by making a connection between proofs and procedural methods of constructing them.

In this paper, we have given operational semantics for Hitac. The big-step semantics gives a meaning to our language, while the small-step semantics considers the intermediate proof states. We have shown that both semantics coincide, which shows that our technically intricate definition of the small-step reduction still produces exactly all the reductions that are needed.

One important result for the small-step semantics is to characterise the normal forms. This required a careful analysis of the "stuck" states (such as when an atomic tactic does not match a goal) that can be reached. Isolating failure points in stuck states will be important to help debugging. However, work still remains to describe further formal properties of our calculus and its extensions.

The calculus we have presented here represents an idealised tactic language. We believe that this is a natural starting point for the formal study of tactic languages. We have kept examples concise on purpose to allow the reader to check them. Larger examples have been explored using a prototype implementation, but the calculus needs additional features before it can be exploited for real. Real tactic languages are considerably richer, for example, including first class and higher-order tacticals, and binding of goal or logical term expressions; they also include fixed orders of evaluation and a diversity of methods for controlling search. At least some of these could be usefully studied in detail in our setting.

Another existing tactic language feature not directly modelled here is the provision of meta-variables in logical formulae. Meta-variables allow goals to depend on one another. Instantiation of a meta-variable has an "action-at-a-distance" effect, potentially altering subgoals in other branches of the proof. This breaks the pure independence property of the tensor product used here and would need further machinery to model properly.

Although we have not discussed it in this paper, a topic that we think our approach may be able to handle well is the reuse of tactics, explicitly addressed in [1] (and also [9, 17, 25]). Indeed, since the underlying derivation system is rather generic, we may hope to be able to write tactics which can be reused between different systems, not just different proofs — providing suitable basic atomic tactics could be represented. However, this suggestion certainly warrants further investigation and experimentation before we can really claim it is valid.

On the practical side, the use of a generic tactic language offers hope that we may one day be able to write tactics that can be ported between different systems, to lift the current state of the art in porting proofs a step higher. We plan to investigate this to see if it is feasible within real interface layers on top of existing provers; the Proof General system is an appealing vehicle for this. Heneveld [12] also conducted experiments into building languages at the interface layer rather than by extending the prover language.

Finally, we would like to exploit the new hierarchical structure we have introduced for real proofs. In associated work by colleagues at Edinburgh, a graphical tool is being developed for displaying and navigating in hiproofs. Explicitly structured low-level proofs are likely to be more informative than unstructured ones; this may even be useful for automatically generated proofs, for example, formal proofs generated as evidence in the certification of software systems.

Acknowledgements. The authors would like to thank Gordon Plotkin, John Power, and Alan Bundy for useful discussions. The first author received support from EPSRC grant number EP/E005713/1.

References

[1] K. Aboul-Hosn. A proof-theoretic approach to tactics. In J. M. Borwein and W. M. Farmer, editors, *Mathematical Knowledge Management, 5th International Conference, MKM 2006*, volume 4108 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2006.

[2] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[3] A. W. Appel and A. P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, 14:3–19, 2004.

[4] A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reasoning*, 39(2):109–139, 2007.

[5] D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1785, pages 38–42. Springer, 2000.

[6] D. Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000, Reunion Island, France, November 6-10, 2000*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95, 2000.

[7] E. Denney, J. Power, and K. Tourlas. Hiproofs: A hierarchical notion of proof tree. In *Proceedings of Mathematical Foundations of Programing Semantics (MFPS)*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2005.

[8] A. P. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *J. Autom. Reasoning*, 11(1):41–81, 1993.

[9] A. P. Felty and D. Howe. Generalization and reuse of tactic proofs. In *Fifth International Conference on Logic Programming and Automated Reasoning LPAR*, volume 822 of *Lecture Notes in Computer Science*. Springer, 1994.

[10] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[11] J. Harrison. Towards self-verification of HOL Light. In *Proc. IJCAR 2006, the third International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[12] A. Heneveld. *Using Features in Interactive Theorem Proving*. PhD thesis, School of Informatics, University of Edinburgh, 2006.

[13] G. I. Jojgov and H. Geuvers. A calculus of tactics and its operational semantics. *Electr. Notes Theor. Comput. Sci.*, 93:118–137, 2004.

[14] F. Kirchner. Coq tacticals and PVS strategies: A small-step semantics. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, pages 69–83. NASA/CP-2003-212448, September 2003.

[15] F. Kirchner and C. Muñoz. PVS#: Streamlined tacticals for PVS. In *Proceedings of STRATEGIES'06*, August 2006.

[16] M. Kohlhase. *OMDoc - An Open Markup Format for Mathematical Documents*, volume 4180 of *Lecture Notes in Computer Science*. Springer, 2006.

[17] T. Kolbe and C. Walther. Reusing proofs. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 80–84, Chichester, Aug. 1994. John Wiley and Sons.

[18] C. Kreitz. *The Nuprl Proof Development System, Version 5 — Reference Manual and User's Guide*. Department of Computer Science, Cornell-University, Ithaca, NY 14853-7501, 2002.

[19] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus — full version. *Formal Aspects of Computing*, 8(E):224–285, 1996.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[21] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a tactic language for refinement. *Formal Aspects of Computing*, 15(1):28–47, 2003.

[22] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[23] R. Pollack. On extensibility of proof checkers. In P. Dybjer, B. Nordström, and J. M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 140–161. Springer, 1994.

[24] C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Tinycals: Step by step tacticals. *Electr. Notes Theor. Comput. Sci.*, 174(2):125–142, 2007.

[25] A. Schairer, S. Autexier, and D. Hutter. A pragmatic approach to reuse in tactical theorem proving. In *Proc. 4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)*, volume 58 of *Electronic Notes in Computer Science*, pages 203–216. Elsevier, 2001.

[26] J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with OMEGA. *Journal of Applied Logic*, 4(4):533–559, Dec. 2006.

[27] J. Siekmann, S. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, and V. Sorge. LOUI: Lovely Omega user interface. *Formal Aspects of Computing*, 11:326–342, 1999.

[28] K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher-Order Logics TPHOLs 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[29] M. Wagner, S. Autexier, and C. Benzmüller. PLATO: A mediator between text-editors and proof assistance systems. In S. Autexier and C. Benzmüller, editors, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, volume 174(2) of *Electronic Notes on Theoretical Computer Science*, pages 87–107. Elsevier, August 2006.

[30] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

David Aspinall
LFCS, School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland
e-mail: `David.Aspinall@ed.ac.uk`

Ewen Denney
RIACS, NASA Ames Research Center
Moffett Field, CA 94035, USA
e-mail: `Ewen.W.Denney@nasa.gov`

Christoph Lüth
Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany
e-mail: `Christoph.Lueth@dfki.de`