A Tactic Language for Hiproofs

David Aspinall[1], Ewen Denney[2], and Christoph Lüth[3]

[1] LFCS, School of Informatics
University of Edinburgh
Edinburgh EH9 3JZ, Scotland
[2] RIACS, NASA Ames Research Center
Moffett Field, CA 94035, USA
[3] Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany

Abstract. We introduce and study a tactic language, *Hitac*, for constructing hierarchical proofs, known as *hiproofs*. The idea of hiproofs is to superimpose a labelled hierarchical nesting on an ordinary proof tree. The labels and nesting are used to describe the organisation of the proof, typically relating to its construction process. This can be useful for understanding and navigating the proof. Tactics in our language construct hiproof structure together with an underlying proof tree. We provide both a big-step and a small-step operational semantics for evaluating tactic expressions. The big-step semantics captures the intended meaning, whereas the small-step semantics hints at possible implementations and provides a unified notion of proof state. We prove that these notions are equivalent and construct valid proofs.

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 incomplete and partial automation, and grappling with many low-level system specific details. Once they have been written, understanding and maintaining such proofs is in some ways even harder: small changes often cause proofs to break completely, and debugging to find the failure point is seldom easy.

Moreover, notions of proof vary from one theorem prover to another, locking users into specific provers they have mastered. 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, thus improving the management of large proofs. In this paper, we study the notion of *hiproof* [1], which takes the *hierarchical* structure of proofs as primary. 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. By developing an idealised tactic language, therefore, we aim to work towards a generic proof representation language.

Figure 1 shows three example hiproofs which illustrate the basic ideas. Diagram (a) shows the structure of an induction procedure: it consists of the application of an induction rule, followed by a procedure for solving the base case and a procedure for solving the step case. The step case uses rewriting and the induction hypothesis to complete the proof. Diagram (b) shows a procedure for solving a positive propositional statement, using implication and then conjunction introduction, solving one subgoal using an axiom and leaving another subgoal unsolved, indicated by an arrow exiting the "Prop" box. The second subgoal is then solved using reflexivity. Diagram (c) shows two labelled hiproofs labelled l and m, respectively. l applies rule a, which produces two subgoals, the first of which is solved inside l with b, and the second of which is solved by the proof labelled m (consisting of a single rule c). We will use this third example as a test case later.

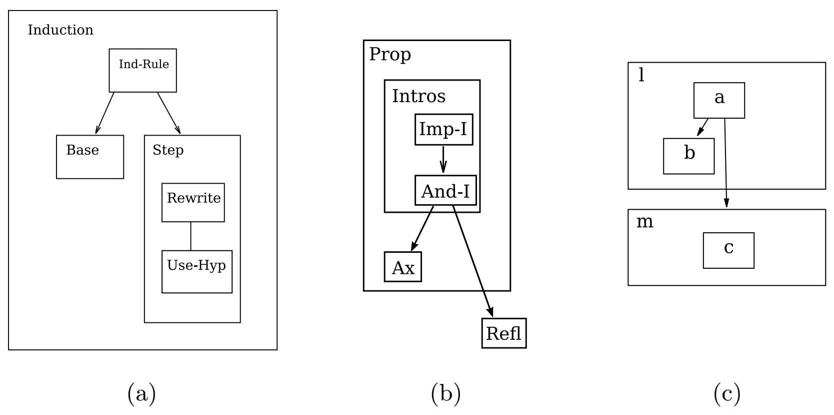

Fig. 1. Example Hiproofs

A hiproof is abstract: nodes are given names corresponding to basic proof rules, procedures or compositions thereof. Each node may be *labelled* with a name: navigation in the proof allows "zooming in" by opening boxes to reveal their content. Boxes which are not open are just visualised with their labels; details inside are suppressed. Hiproofs are an abstraction of a proof in an underlying logic or derivation system; we call a hiproof *valid* if it can be mapped on to an underlying proof tree, where nodes are given by derivable judgements.

The central topic, and novelty, introduced in this paper is a tactic programming language for constructing valid hiproofs. The language is general and not tied to a specific system. It is deliberately restrictive: at the moment we seek to understand the connection between hierarchical structure and some core constructs for tactic programming, namely, alternation, repetition and assertion; features such as meta-variable instantiation and binding are left to future work. Part of the value of our contribution is the semantically precise understanding of this core.

Outline. The next section introduces a syntax for hiproofs and explains the notion of validity. By extending this syntax we introduce *tactics* which can be used to construct programs. In Section 3 we study notions of *evaluation*: a tactic can be applied to a goal and, if successful, evaluated to produce a valid hiproof. We consider two operational semantics: a big-step relation which defines the meaning of our constructs, and a finer-grained small-step semantics with a notion of *proof state* that evolves while the proof is constructed. In Section 4 we demonstrate how our language can be used to define some familiar tactics. Section 5 concludes and relates our contribution to previous work.

2 Syntax for hiproofs and tactics

Hiproofs add structure to an underlying derivation system, introduced shortly. Hiproofs are ranged over by s and are given by terms in the following grammar:

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

We assume $a \in \mathcal{A}$ where $\mathcal{A}$ is the set of *atomic tactics* given by the underlying derivation system. The remaining constructors add the structure introduced above: labelling introduces named boxes (which can be nested arbitrarily deep); sequencing composes one hiproof after another, and tensor puts two hiproofs side-by-side, operating on two 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. The empty hiproof is the vacuous proof for an empty set of goals. Hiproofs have a denotational semantics given in previous work [1]; the syntax above serves as an internal language for models in that semantics. The denotational semantics justifies certain equations between terms, in particular, empty is a unit for the tensor, and tensor and sequencing are associative.

When writing hiproof terms we use the following syntactic conventions: the scope of the label l in $[l]\, s$ extends as far as possible and tensor binds more tightly that sequencing.

Example 1. The hiproof in Fig. 1(c) is written as

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

Notice the role of id corresponding to the line exiting the box labelled l.

The underlying derivation system defines sets of *atomic tactics* $a \in \mathcal{A}$ and *atomic goals* $\gamma \in \mathcal{G}$. Typically, what we call a goal is given by a judgement form

$$\frac{\dfrac{\gamma_1 \cdots \gamma_n}{\gamma} \quad a_\gamma \in \mathcal{A}}{a \vdash \gamma \longrightarrow \gamma_1 \otimes \cdots \otimes \gamma_n} \qquad \text{(V-Atomic)}$$

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

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

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

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

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

Fig. 2. Validation of Hiproofs

in the underlying derivation system. We work with lists of goals g written using the binary tensor:

$$g ::= \gamma \;\big|\; g \otimes g \;\big|\; \langle\rangle$$

The tensor is associative and unitary, with $\langle\rangle$ the unit (empty list). We write $g : n$ to indicate the length of g, i.e., when $g = \gamma_1 \otimes \cdots \otimes \gamma_n$ or $g = \langle\rangle$ for $n = 0$, called the *arity*. Elementary inference rules in the underlying derivation system can be seen as atomic tactics of the following form:

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

which given goals $\gamma_1, \ldots, \gamma_n$ produce a proof for γ. There are, of course, other atomic tactics possible. However, for a particular atomic tactic a, there may be a family of goals γ to which it applies; we write a_γ to make the instance of a precise. A restriction is that every instance of a must have the same arity, i.e., number of premises n. By composing atomic tactics, we can produce *proofs* in the derivation system. Thus, each hiproof s has a family of underlying proofs which consist of applications of the instances of the underlying atomic tactics. We say that s *validates* proofs from g_2 to g_1, written $s \vdash g_1 \longrightarrow g_2$. Validation is defined by the rules in Fig. 2.
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 [1]. The (input) arity $s : n$ of a hiproof s with $s \vdash g_1 \longrightarrow g_2$ is n where $g_1 : n$; note that again by the restriction on atomic tactics, a hiproof has a unique arity, if it has any.

Example 2. Suppose we have a goal γ_1 which can be proved like this:

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

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

To show how the abstract hiproofs may be used with a real underlying derivation system, we give two small examples with different sorts of underlying goals.

Example 3. Minimal propositional logic MPL has the formulae:

$$P ::= \mathsf{TT} \;\big|\; \mathsf{FF} \;\big|\; X \;\big|\; P \Longrightarrow P$$

where X is a propositional variable. Goals in MPL have the form $\Gamma \vdash P$, where Γ is a set of propositions. The atomic tactics correspond to the natural deduction rules:

$$\dfrac{}{\Gamma, P \vdash P}\,\mathrm{ax} \qquad \dfrac{\Gamma \cup \{P\} \vdash Q}{\Gamma \vdash P \Longrightarrow Q}\,\mathrm{impI} \qquad \dfrac{\Gamma \vdash P \Longrightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}\,\mathrm{impE}$$

$$\dfrac{\Gamma \vdash P}{\Gamma \cup \{Q\} \vdash P}\,\mathrm{wk}$$

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

Example 4. Minimal equational logic MEL 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 $a, b, c, d \in \mathcal{T}_\Sigma(X)$):

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

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

$$\dfrac{a = b}{a[c/x] = b[c/x]}\,\mathrm{subst}'$$

can be derived with the hiproof $h_1 = \mathit{subst} \,;\, \mathsf{id} \otimes \mathit{refl}$ of arity 1, whereas the usual congruence rule say for a binary operation f

$$\dfrac{a_1 = b_1 \quad a_2 = b_2}{f(a_1, a_2) = f(b_1, b_2)}\,\mathrm{ctxt}$$

can be derived with the hiproof $h_2 = \mathit{subst} \,;\, (\mathit{subst} \,;\, \mathit{refl} \otimes \mathsf{id}) \otimes \mathsf{id}$ of arity 1.

2.1 Tactics and programs

The hiproofs introduced above are static proof structures which represent the result of executing a tactic. We now present a language of tactics which can be evaluated to construct such hiproofs. These *tactic expressions* will be the main object of study. They are defined by extending the grammar for hiproofs with three new cases:

$$\begin{array}{lll} t ::= a \mid \mathsf{id} \mid [l]\,t \mid t\,;t \mid t \otimes t \mid \langle\rangle & & \text{as for hiproofs} \\ \quad\;\mid t \mid t & & \text{alternation} \\ \quad\;\mid \mathsf{assert}\ \gamma & & \text{goal assertion} \\ \quad\;\mid T & & \text{defined tactic} \end{array}$$

Together the three new cases allow proof search: alternation allows alternatives, assertion allows controlling the control flow, and defined tactics allow us to build up a program of possibly mutually recursive definitions. Syntactic conventions for hiproofs are extended to tactic expressions, with alternation having precedence between sequencing (lowest) and tensor (highest). Alternation is also associative. Further, the arity $t : n$ of a tactic is defined as follows: for a hiproof s, it has been defined above; $[l]\,t : 1$; if $t_1 : n$, then $t_1 \,;\, t_2 : n$; if $t_1 : n$ and $t_2 : m$, then $t_1 \otimes t_2 : n+m$; if $t_1 : n$ and $t_2 : n$, then $t_1 \mid t_2 : n$; $\mathsf{assert}\ \gamma : 1$; and if $T \overset{def}{=} t \in \mathit{Prog}$ and $t : n$, then $T : n$. This definition is partial, not all tactics can be given an arity.

Programs. A *tactic program Prog* is a set of definitions of the form $T_i \overset{def}{=} t_i$, 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 all of the definition bodies t_i.

The uniqueness requirement on labels is so that we can map a label in a hiproof back to a unique origin within the program – although notice that, because of recursion, the same label may appear many times on the derivation.

We do not make restrictions on the goal matching relation. In some cases it may simply be an equivalence relation on goals. For MEL, 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 Fig. 1(c) by defining:

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

If we *evaluate* the tactic T_u applied to the goal γ_1, we get the hiproof shown earlier, $([l]\,a\,;\,b \otimes \mathsf{id})\,;\,[m]\,c$, with no left over goals.

The next section provides an operational semantics to define a notion of evaluation that has this effect.

3 Operational semantics

To give a semantics to tactic expressions, we will consider an operational semantics as primary. This is in contrast to other approaches which model things as the original LCF-style tactic programming does, i.e., using a fixed-point semantics to explain recursion (e.g., ArcAngel [2]). We believe that an operational semantics is more desirable at this level, 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, and hope for providing better debugging tools (a similar motivation was given for Tinycals [3]).

There is a crucial difference between hiproofs and tactic expressions. Because of alternation and repetition, tactic evaluation is non-deterministic: a tactic expression 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 – which is why our notion of arity is partial. This is not surprising because part of the point of tactic programming is to write tactics that can apply to varying numbers of subgoals using arbitrary recursive functions. It is one of the things that makes tactic programming difficult. In future work we plan to investigate richer static type systems for tactic programming, in the belief that there is a useful intermediate ground between ordinary unchecked tactics and what could be called "certified tactic programming" [4, 5], where tactics are shown to construct correct proofs using dependent typing.

Here, we use untyped tactic terms, as is more usual. We begin by defining a big-step semantics that gives meaning to expressions without explicitly specifying the intermediate states. In Sect. 3.2 we give a small-step semantics which provides a notion of intermediate proof state.

3.1 Big-step semantics

The big-step semantics is shown in Fig. 3. It defines a relation $\langle g, t\rangle \Downarrow \langle s, g'\rangle$ inductively, which explains 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 $\langle g, t\rangle \Downarrow \langle s, \langle\rangle\rangle$, for some hiproof s. The relation is defined with respect to a program *Prog* containing a set of definitions.

The rules directly capture the intended meaning of tactic expressions. For example, (B-Label) evaluates a labelled tactic expression $[l]\,t$, 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 key restrictions in the notion of hiproof (motivated in [1]), namely that a box has a unique entry point, its root, accepting a single (atomic) goal.

Notice that 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

$$\frac{\dfrac{\gamma_1 \cdots \gamma_n}{\gamma} \quad a_\gamma \in \mathcal{A}}{\langle \gamma, a\rangle \Downarrow \langle a, \gamma_1 \otimes \cdots \otimes \gamma_n\rangle} \ \text{(B-Atomic)} \qquad \langle\langle\rangle, \langle\rangle\rangle \Downarrow \langle\langle\rangle, \langle\rangle\rangle \ \text{(B-Empty)}$$

$$\frac{}{\langle \gamma, \mathsf{id}\rangle \Downarrow \langle \mathsf{id}, \gamma\rangle} \ \text{(B-Id)} \qquad \frac{\langle g, t_1\rangle \Downarrow \langle s, g'\rangle}{\langle g, t_1 \mid t_2\rangle \Downarrow \langle s, g'\rangle} \ \text{(B-Alt-L)}$$

$$\frac{\langle \gamma, t\rangle \Downarrow \langle s, g\rangle}{\langle \gamma, [l]\, t\rangle \Downarrow \langle [l]\, s, g\rangle} \ \text{(B-Label)} \qquad \frac{\langle g, t_2\rangle \Downarrow \langle s, g'\rangle}{\langle g, t_1 \mid t_2\rangle \Downarrow \langle s, g'\rangle} \ \text{(B-Alt-R)}$$

$$\frac{\langle g_1, t_1\rangle \Downarrow \langle s_1, g_2\rangle \quad \langle g_2, t_2\rangle \Downarrow \langle s_2, g_3\rangle}{\langle g_1, t_1 \ ; t_2\rangle \Downarrow \langle s_1 \ ; s_2, g_3\rangle} \ \text{(B-Seq)} \qquad \frac{\gamma \lesssim \gamma'}{\langle \gamma', \mathsf{assert}\ \gamma\rangle \Downarrow \langle \mathsf{id}, \gamma'\rangle} \ \text{(B-Assert)}$$

$$\frac{\langle g_1, t_1\rangle \Downarrow \langle s_1, g_1'\rangle \quad \langle g_2, t_2\rangle \Downarrow \langle s_2, g_2'\rangle}{\langle g_1 \otimes g_2, t_1 \otimes t_2\rangle \Downarrow \langle s_1 \otimes s_2, g_1' \otimes g_2'\rangle} \ \text{(B-Tensor)} \qquad \frac{T \overset{\mathit{def}}{=} t \in \mathit{Prog} \quad \langle g, t\rangle \Downarrow \langle s, g'\rangle}{\langle g, T\rangle \Downarrow \langle s, g'\rangle} \ \text{(B-Def)}$$

Fig. 3. Big-step semantics for *Hitac*

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., there must be some subterm $\langle g, t\rangle$ for which there is no $\langle s, g'\rangle$ it evaluates to). 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 the obvious source of non-determinism is alternation, the tensor rule also allows the (possibly angelic) splitting of an input goal list into two halves $g_1 \otimes g_2$, including the possibility that g_1 or g_2 is the empty tensor $\langle\rangle$.

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 $\langle g, t\rangle \Downarrow \langle s, g'\rangle$.

Fact 1 (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 x from Prog.*

The label origin property is immediate by the definition of program and the observation that evaluation only introduces labels from the program. It means that we can use labels as indexes into the program to find where a subproof was produced, which is the core motivation for labelling, and allows a source level debugging of tactical proofs.

3.2 Small-step semantics

Besides the big-step semantics given above, it is desirable to explain tactic evaluation using a small-step semantics. The typical reason for providing small-step semantics is to give meaning to non-terminating expressions. In principle we don't need to do this here (non-terminating tactics do not produce proofs), but in practice we are interested in debugging tactic expressions during their evaluation, including ones which may fail. A small step-semantics provides a notion of intermediate state which helps do this.

We now define an evaluation machine which evolves a *proof state* configuration in each step, eventually producing a hiproof. The reduction is again 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.

Formulation of a small-step semantics is not as straightforward as the big-step semantics, because it needs to keep track of the intermediate stages of evaluation, which do not correspond to a well-formed hiproof. The difficulty is in recording which tactics have been evaluated and which not, and moving the goals which remain in subtrees out of their hierarchical boxes. It was surprisingly hard to find an elegant solution. The mechanism we eventually found is pleasantly simple; it was devised by visualising goals moving in the geometric representation of hiproofs; they move along lines, in and out of boxes, and are split when entering a tensor and rejoined afterwards.

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, tactic expressions and goal lists as sublanguages:

$$p ::= a \;\big|\; \mathsf{id} \;\big|\; [l]\, p \;\big|\; p \,;\, p \;\big|\; p \otimes p \;\big|\; p \mid p \;\big|\; \mathsf{assert}\,\gamma \;\big|\; T \;\big|\; g$$

The general judgement form is $p \;\Rightarrow\; p'$, defined by the rules shown in Fig. 4. A proof state, p, consists of a mixture of open goals, g, active tactics, t, and successfully applied tactics, i.e., hiproofs, s. Composing proof states can be understood as connecting, or applying, the tactics of one state to the open goals of another. In particular, the application of tactic t to goal g has the form $g \,;\, t$, and we treat the sequencing operator on proof states as associative. The notion of value is a fully reduced proof state with the form $s \,;\, g'$. A complete reduction, therefore, has the form:

$$g \,;\, t \;\Rightarrow^*\; s \,;\, g'.$$

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'. Note that not all terms in this grammar are meaningful. In the rules, therefore, we will restrict attention to reductions of a meaningful form, and 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-Alt), g has to be a goal and

$$\frac{\dfrac{\gamma_1 \cdots \gamma_n}{\gamma} \quad a_\gamma \in \mathcal{A}}{\gamma \; ; \; a \;\Rightarrow\; a \; ; \; \gamma_1 \cdots \otimes \cdots \gamma_n} \quad \textsf{(S-Atomic)}$$

$$\gamma \; ; \; \textsf{id} \;\Rightarrow\; \textsf{id} \; ; \; \gamma \quad \textsf{(S-Id)}$$

$$\gamma \; ; \; [l]\, t \;\Rightarrow\; [l]\, \gamma \; ; \; t \quad \textsf{(S-Enter)}$$

$$[l]\, s \; ; \; g \;\Rightarrow\; ([l]\, s) \; ; \; g \quad \textsf{(S-Exit)}$$

$$g_1 \otimes g_2 \; ; \; p_1 \otimes p_2 \;\Rightarrow\; (g_1 \; ; \; p_1) \otimes (g_2 \; ; \; p_2) \quad \textsf{(S-Split)}$$

$$(s_1 \; ; \; g_1) \otimes (s_2 \; ; \; g_2) \;\Rightarrow\; s_1 \otimes s_2 \; ; \; g_1 \otimes g_2 \quad \textsf{(S-Join)}$$

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

$$(s_1 \; ; \; g) \mid p_2 \;\Rightarrow\; s_1 \; ; \; g \quad \textsf{(S-Sel-L)}$$

$$p_1 \mid (s_2 \; ; \; g) \;\Rightarrow\; s_2 \; ; \; g \quad \textsf{(S-Sel-R)}$$

$$\frac{\gamma \lesssim \gamma'}{\gamma \; ; \; \textsf{assert}\ \gamma' \;\Rightarrow\; \textsf{id} \; ; \; \gamma} \quad \textsf{(S-Assert)}$$

$$\frac{T \overset{def}{=} t \in \mathit{Prog}}{g \; ; \; T \;\Rightarrow\; g \; ; \; t} \quad \textsf{(S-Def)}$$

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

$$\frac{p_1 \;\Rightarrow\; p_1'}{p_1 \; ; \; p_2 \;\Rightarrow\; p_1' \; ; \; p_2} \quad \textsf{(S-Seq-L)}$$

$$\frac{p_2 \;\Rightarrow\; p_2'}{p_1 \; ; \; p_2 \;\Rightarrow\; p_1 \; ; \; p_2'} \quad \textsf{(S-Seq-R)}$$

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

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

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

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

Fig. 4. Small-step semantics for *Hitac*

t_1, t_2 must be tactics. Further, note that the empty goal and the empty hiproof and tactic are both denoted by $\langle\rangle$; this gives rise to the identity $\langle\rangle \; ; \; \langle\rangle = \langle\rangle \; ; \; \langle\rangle$ where on the left we have a tactic applied to an empty goal, and on the right a hiproof applied to an empty goal. The small-step semantics therefore does not need an explicit rule for the empty case.

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

The theorems below show that restrictions do not limit the language, by relating it to the big-step semantics.

Example 6. Consider the tactic program from Ex. 5. We show the reduction of T_u applied to the goal γ_1. The steps name the major rule applied at each point.

$$\begin{array}{ll}
\gamma_1 \; ; \; T_u \;\Rightarrow & \\
\Rightarrow \gamma_1 \; ; \; (\textsf{assert}\ \gamma_3 \; ; \; T_m) \mid (T_l \; ; \; T_u) & \textsf{(S-Def)} \\
\Rightarrow (\gamma_1 \; ; \; \textsf{assert}\ \gamma_3 \; ; \; T_m) \mid (\gamma_1 \; ; \; T_l \; ; \; T_u) & \textsf{(S-Alt)} \\
\Rightarrow \ldots \gamma_1 \; ; \; ([l]\, b \; ; \; c \otimes \textsf{id}) \; ; \; T_u & \text{reduce on right, } \textsf{(S-Def)} \\
\Rightarrow \ldots ([l]\, \gamma_1 \; ; \; b \; ; \; c \otimes \textsf{id}) \; ; \; T_u & \textsf{(S-Enter)}
\end{array}$$

$$
\begin{array}{ll}
\Rightarrow \ldots ([l]\, b \mathbin{;} \gamma_2 \otimes \gamma_3 \mathbin{;} c \otimes \mathsf{id}) \mathbin{;} T_u & \text{(S-Atomic)} \\
\Rightarrow \ldots ([l]\, b \mathbin{;} (\gamma_2 \mathbin{;} c) \otimes (\gamma_3 \mathbin{;} \mathsf{id})) \mathbin{;} T_u & \text{(S-Split)} \\
\Rightarrow \ldots ([l]\, b \mathbin{;} (c \mathbin{;} \langle\rangle) \otimes (\gamma_3 \mathbin{;} \mathsf{id})) \mathbin{;} T_u & \text{(S-Atomic)} \\
\Rightarrow \ldots ([l]\, b \mathbin{;} (c \mathbin{;} \langle\rangle) \otimes (\mathsf{id} \mathbin{;} \gamma_3)) \mathbin{;} T_u & \text{(S-Id)} \\
\Rightarrow \ldots ([l]\, b \mathbin{;} (c \otimes \mathsf{id}) \mathbin{;} \gamma_3) \mathbin{;} T_u & \text{(S-Join)}, \langle\rangle \otimes \gamma_3 \equiv \gamma_3 \\
\Rightarrow \ldots ([l]\, b \mathbin{;} c \otimes \mathsf{id}) \mathbin{;} \gamma_3 \mathbin{;} T_u & \text{(S-Exit)} \\
\Rightarrow \ldots ([l]\, b \mathbin{;} c \otimes \mathsf{id}) \mathbin{;} \gamma_3 & \\
\qquad \mathbin{;} (\mathsf{assert}\ \gamma_3 \mathbin{;} T_m) \mid (T_l \mathbin{;} T_u) & \text{(S-Def)} \\
\Rightarrow \ldots \mathbin{;} (\gamma_3 \mathbin{;} \mathsf{assert}\ \gamma_3 \mathbin{;} T_m) \mid (\gamma_3 \mathbin{;} T_l \mathbin{;} T_u) & \text{(S-Alt)} \\
\Rightarrow \ldots \mathbin{;} (\gamma_3 \mathbin{;} T_m) \mid (\gamma_3 \mathbin{;} T_l \mathbin{;} T_u) & \text{(S-Assert)} \\
\Rightarrow \ldots \mathbin{;} (\gamma_3 \mathbin{;} [m]\, c) \mid (\gamma_3 \mathbin{;} T_l \mathbin{;} T_u) & \text{(S-Def)} \\
\Rightarrow \ldots \mathbin{;} ([m]\, \gamma_3 \mathbin{;} c) \mid (\gamma_3 \mathbin{;} T_l \mathbin{;} T_u) & \text{(S-Enter)} \\
\Rightarrow \ldots \mathbin{;} ([m]\, c \mathbin{;} \langle\rangle) \mid (\gamma_3 \mathbin{;} T_l \mathbin{;} T_u) & \text{(S-Atomic)} \\
\Rightarrow \ldots \mathbin{;} ([m]\, c) \mathbin{;} \langle\rangle \mid (\gamma_3 \mathbin{;} T_l \mathbin{;} T_u) & \text{(S-Exit)} \\
\Rightarrow \ldots ([l]\, b \mathbin{;} c \otimes \mathsf{id}) \mathbin{;} ([m]\, c) \mathbin{;} \langle\rangle & \text{(S-Sel-L)} \\
\Rightarrow ([l]\, b \mathbin{;} c \otimes \mathsf{id}) \mathbin{;} ([m]\, c) \mathbin{;} \langle\rangle & \text{(S-Sel-R)}
\end{array}
$$

The final value is as claimed in Ex. 5.

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 of tactic expressions.

Theorem 2 (Completeness of small-step semantics). *If $\langle g, t\rangle \Downarrow \langle s, g'\rangle$, then $g \mathbin{;} t \Rightarrow^* s \mathbin{;} g'$*

Proof. Straightforward induction on big-step derivation.

Theorem 3 (Soundness of small-step semantics). *If $g \mathbin{;} t \Rightarrow^* s \mathbin{;} g'$ then $\langle g, t\rangle \Downarrow \langle s, g'\rangle$.*

Proof. By induction on the length of the derivation, using Lemma 1.

Lemma 1 (Structure preservation).

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

Proof. Each part by induction on the lengths of sequences involved. For each constructor, a major rule is the base case and congruence rules are step cases.

Theorems 1 and 3 give correctness also for the small step semantics.

Corollary 1 (Correctness of small-step semantics). *If $g \mathbin{;} t \Rightarrow^* s \mathbin{;} g'$ then $s \vdash g \longrightarrow g'$.*

4 Tactic Programming

Tactics as above are procedures which produce hiproofs. To help with writing tactics, most theorem provers provide *tacticals* (tactic functionals or higher-order tactics), which combine existing tactics into new ones. The simplest examples of tacticals are the alternation and sequencing operations for tactics. Theorem provers like the original LCF, Isabelle, HOL or Coq provide more advanced patterns of applications; we concentrate on two illustrative cases here.

We will write tacticals as a meta-level notion, i.e., the following equations are meant as short-cuts defining one tactic for each argument tactic t:

$$\begin{aligned}\mathtt{ALL}\ t &\overset{def}{=} (t \otimes \mathtt{ALL}\ t) \mid \langle\rangle \\ \mathtt{ID} &\overset{def}{=} \mathtt{ALL}\ \mathsf{id} \\ \mathtt{REPEAT}\ t &\overset{def}{=} (t\ ;\mathtt{REPEAT}\ t) \mid \mathtt{ID}\end{aligned}$$

$\mathtt{ALL}$ t applies t to as many atomic goals as available; in particular, $\mathtt{ID}$ is the 'polymorphic identity', which applied to any goal $g : n$ reduces to id^n, the n-fold tensor of id. $\mathtt{REPEAT}\ t$ applies t as often as possible. An application of this is a tactic to strip away all implications in the logic MPL (Example 3), defined as $\mathit{stripImp} \overset{def}{=} \mathtt{REPEAT}\ \mathit{impI}$. To see this at work, here it is used in an actual proof:

$$\begin{aligned}&\vdash A \Longrightarrow (B \Longrightarrow A)\ ;\mathit{stripImp}\\ &\Rightarrow \vdash A \Longrightarrow (B \Longrightarrow A)\ ;(\mathit{impI}\ ;\mathtt{REPEAT}\ \mathit{impI}) \mid \mathtt{ID}\\ &\Rightarrow^*\ (\{A\} \vdash B \Longrightarrow A\ ;\mathtt{REPEAT}\ \mathit{impI}) \mid (\vdash A \Longrightarrow (B \Longrightarrow A)\ ;\mathtt{ID})\\ &\Rightarrow^*\ (\mathsf{id}\ ;\{A, B\} \vdash A) \mid \ldots \mid \ldots\\ &\Rightarrow^*\ \mathsf{id}\ ;\{A, B\} \vdash A\end{aligned}$$

The last goal is easily proven with the atomic tactic *ax*.

4.1 Deterministic semantics

The big- and small-step semantics given above are non-deterministic: a tactic t applied to a goal g may evaluate to more than one hiproof s (and remaining goals g'). This may result in many 'unwanted' reductions along with successful ones; e.g., $\mathtt{REPEAT}\ t\ ;g$ can always reduce to $\mathsf{id}\ ;g$. Non-determinism has its advantages: the tensor splitting allows a tactic such as $\mathtt{ALL}\ b \otimes \mathtt{ALL}\ c$ to solve the goal $\gamma_2 \otimes \gamma_2 \otimes \gamma_3$ by splitting the tensor judiciously: $\gamma_2 \otimes \gamma_2 \otimes \gamma_3\ ;\mathtt{ALL}\ b \otimes \mathtt{ALL}\ c \ \Rightarrow\ (\gamma_2 \otimes \gamma_2\ ;\mathtt{ALL}\ b) \otimes (\gamma_3\ ;\mathtt{ALL}\ c)$. However, this behaviour is hard to implement (it requires keeping track of all possible reductions, and selecting the right ones after the fact), and it is in marked contrast to the usual alternation tactical ($\mathtt{ORELSE}$ in the LCF family) which always selects the first alternative if it is successful, and the second otherwise.

To give a deterministic behaviour for our language, we can define a restricted small-step semantics, which includes a strict subset of the reductions of the non-deterministic one. Since the principal sources of non-determinism are the alternation and the tensor splitting rules, the deterministic small-step semantics has the same rules as the small-step semantics from Sect. 3.2, but replaces rules (S-Sel-L), (S-Sel-R) and (S-Split) with the following:

$$(s_1 \ ; g) \mid p_2 \ \Rightarrow_D \ s_1 \ ; g \qquad (\mathsf{TS}_D\text{-}\mathsf{Alt\text{-}L})$$

$$\frac{p_1 \ \not\Rightarrow_D \ s_1 ; h}{p_1 \mid (s_2 \ ; g) \ \Rightarrow_D \ s_2 \ ; g} \qquad (\mathsf{TS}_D\text{-}\mathsf{Alt\text{-}R})$$

$$\frac{g_1 : n \qquad t_1 : n}{g_1 \otimes g_2 \ ; t_1 \otimes t_2 \ \Rightarrow_D \ (g_1 \ ; t_1) \otimes (g_2 \ ; t_2)} \qquad (\mathsf{TS}_D\text{-}\mathsf{Split})$$

The right alternative is only chosen if the left alternative does not evaluate to any hiproof. Further, we only allow the argument goals to be split if the first component of the tactic has a fixed arity. This means that in the above example, $\gamma_2 \otimes \gamma_2 \otimes \gamma_3 \ ; \ \mathtt{ALL}\ b \otimes \mathtt{ALL}\ c$ does not reduce; to prove that goal under deterministic reduction, we need the tactic $\mathtt{ALL}\ (b \mid c)$.

Theorem 4 (Deterministic small-step semantics). *The deterministic semantics is a restriction of the non-deterministic one:*

$$\textit{if } t \ ; g \ \Rightarrow_D^* \ s \ ; g \textit{ then } t \ ; g \ \Rightarrow^* \ s \ ; g$$

Proof. A simple induction on the derivation of $\Rightarrow_D^*$. The rules (TS_D-Alt-L), (TS_D-Alt-R) and (TS_D-Split) are admissible in the non-deterministic small-step semantics.

Corollary 2. *(Soundness of the deterministic semantics) If $g \ ; t \ \Rightarrow_D^* \ s \ ; g$, then $\langle g, t\rangle \Downarrow \langle s, g'\rangle$.*

Theorem 4 implies that the deterministic semantics is weaker. In fact, it suffices for our simple example tacticals but some others (e.g., violating the arity restriction) are not covered. To recover more of the expressivity of the non-deterministic semantics, one could introduce a notion of backtracking, and treat the tensor split in a demand driven way to avoid the restriction of fixed arity. However, these extensions are beyond the scope of the present paper.

5 Related work and conclusions

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.

Our work with hierarchical structure in the form of hiproofs is unique, although there are many related developments on tactic language semantics and structured proofs elsewhere. A full survey is beyond our scope (more references can be found in [1]), but we highlight some recent and more closely connected developments.

Traditional LCF-style tactic programming uses a full-blown programming language for defining new tactics, as is also done in the modern HOL systems. The direct way of understanding such tactics is as the functions they define over proof states, suggesting a denotational fixed point semantics such as studied by Oliveira et al. [2]. 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.

Kirchner [7] 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 a similar semantic framework. He defines a judgement $e/\tau \rightarrow e'/\tau'$, 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.

Tinycals [3] is a recent small-step tactic language, implemented in the Matita system. The main motivation is to allow stepping inside tactics in order to extend Proof General-like interaction. In Coq and other systems of which we are aware, single-stepping defined tactics using their source is not possible, at best they can be *traced* 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 is also reminiscent of Isabelle's declarative proof language Isar [8], but there is no naming for the nested structure in either case.

One system that bears a closer structural resemblance to the hiproof notion is NuPrl's *tactic tree* [9], which extends LCF-style tactics by connecting them to proof trees, as would be done by combining our big-step semantics with the validation check which links a hiproof to an underlying tree. NuPrl allows navigating the tree and expanding or replacing tactics at each node.

Future work. Work still remains to fully describe the formal properties of the calculus and its extensions, including type systems for arity checking as hinted at in Sect. 3, and further deterministic evaluation relations. One important result for the small-step semantics is to characterise the normal forms. This requires a careful analysis of the "stuck" states (such as when an atomic tactic does not match a goal) that can be reached, and we have some preliminary results on this. Isolating failure points in stuck states will be important to help debugging.

The calculus we have presented here represents an idealised simple 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. Clearly, larger examples should be explored, but this will require mechanical assistance in some form. Moreover, there is clearly a diversity of concepts and constructs which are used to guide proof search in real systems. For example, many proof assistants allow goals to depend on each other via a meta-variable mechanism. More generally, we can envision interdependencies between each of tactics, goals, and proofs, and this leads us to speculate on the possibility of a "tactic cube" (analogous to Barendregt's Lambda Cube) of tactic languages.

On the practical side, the use of a generic tactic language offers hope that we will be able to write tactics that can be ported between different systems. We plan to investigate this and other issues with an implementation in Proof General. In associated work at Edinburgh, a graphical tool is being developed for displaying and navigating in hiproofs. Finally, one of us is developing a system which uses auto-generated formal proofs as evidence for the certification of safety-critical software. In this regard, the explicitly structured proofs which result from applications of tactics are likely to prove more useful than the unstructured proofs which are generated by most present theorem provers.

Acknowledgements. The authors would like to thank Gordon Plotkin, John Power, and Alan Bundy for useful discussions.

References

1. Denney, E., Power, J., Tourlas, K.: 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)
2. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: ArcAngel: a tactic language for refinement. Formal Aspects of Computing **15**(1) (2003) 28–47
3. Coen, C.S., Tassi, E., Zacchiroli, S.: Tinycals: Step by step tacticals. Electr. Notes Theor. Comput. Sci. **174**(2) (2007) 125–142
4. Pollack, R.: On extensibility of proof checkers. In Dybjer, P., Nordström, B., Smith, J.M., eds.: TYPES. LNCS 996, Springer (1994) 140–161
5. Appel, A.W., Felty, A.P.: Dependent types ensure partial correctness of theorem provers. Journal of Functional Programming **14** (January 2004) 3–19
6. Delahaye, D.: A tactic language for the system Coq. In: Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000, Reunion Island, France, 2000. LNCS 1955, Springer (2000) 85– 95
7. Kirchner, F.: Coq tacticals and PVS strategies: A small-step semantics. In Archer, M. et al., eds.: Design and Application of Strategies/Tactics in Higher Order Logics, NASA (September 2003) 69–83
8. Wenzel, M.: Isar — a generic interpretative approach to readable formal proof documents. In Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Thery, L., eds.: Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99. LNCS 1690, Springer (1999) 167– 184.
9. Griffin, T.G.: Notational Definition and Top-down Refinement for Interactive Proof Development Systems. PhD thesis, Cornell University (1988)

A Example Reductions

Here are the reductions for the examples from Section 4 in detail. We first show how `ALL` id reduces applied to an empty goal list.

$$\begin{array}{ll}
\langle\rangle \mathbin{;} \mathtt{ALL}\ \mathsf{id} & \\
\ \Rightarrow \langle\rangle \mathbin{;} (\mathsf{id} \otimes \mathtt{ALL}\ \mathsf{id}) \mid \langle\rangle & (\textsf{S-Def}) \\
\ \Rightarrow (\langle\rangle \mathbin{;} (\mathsf{id} \otimes \mathtt{ALL}\ \mathsf{id}) \mid (\langle\rangle \mathbin{;} \langle\rangle) & (\textsf{S-Alt}) \\
\ \Rightarrow \langle\rangle \mathbin{;} \langle\rangle & (\textsf{S-Sel-R})
\end{array}$$

We can use this to show how `ID` reduces for a goal of arity 2.

$$\begin{array}{ll}
\gamma_1 \otimes \gamma_2 \mathbin{;} \mathtt{ID} & \\
\ \Rightarrow^* \gamma_1 \otimes \gamma_2 \mathbin{;} (\mathsf{id} \otimes \mathtt{ALL}\ \mathsf{id}) \mid \langle\rangle & (\textsf{S-Def}) \\
\ \Rightarrow\ \ (\gamma_1 \otimes \gamma_2 \mathbin{;} \mathsf{id} \otimes \mathtt{ALL}\ \mathsf{id}) \mid (\gamma_1 \otimes \gamma_2 \mathbin{;} \langle\rangle) & (\textsf{S-Alt}) \\
\ \Rightarrow\ \ ((\gamma_1 \mathbin{;} \mathsf{id}) \otimes (\gamma_2 \mathbin{;} \mathtt{ALL}\ \mathsf{id})) \mid \ldots & (\textsf{S-Split}) \\
\ \Rightarrow\ \ ((\mathsf{id} \mathbin{;} \gamma_1) \otimes (\gamma_2 \mathbin{;} (\mathsf{id} \otimes \mathtt{ALL}\ \mathsf{id}) \mid \langle\rangle)) \mid \ldots & (\textsf{S-Id}),(\textsf{S-Def}) \\
\ \Rightarrow^* ((\mathsf{id} \mathbin{;} \gamma_1) \otimes (\gamma_2 \otimes \langle\rangle \mathbin{;} \mathsf{id} \otimes \mathtt{ALL}\ \mathsf{id}) \mid (\gamma_2 \mathbin{;} \langle\rangle)) \mid \ldots & (\textsf{S-Alt}),\ \gamma_2 \equiv \gamma_2 \otimes \langle\rangle \\
\ \Rightarrow^* ((\mathsf{id} \mathbin{;} \gamma_1) \otimes ((\gamma_2 \mathbin{;} \mathsf{id}) \otimes (\langle\rangle \mathbin{;} \mathtt{ALL}\ \mathsf{id}) \mid \ldots)) \mid \ldots & (\textsf{S-Split}) \\
\ \Rightarrow^* ((\mathsf{id} \mathbin{;} \gamma_1) \otimes ((\mathsf{id} \mathbin{;} \gamma_2) \otimes (\langle\rangle \mathbin{;} \langle\rangle) \mid \ldots)) \mid \ldots & (\textsf{S-Id}),\ \text{see above} \\
\ \Rightarrow^* ((\mathsf{id} \mathbin{;} \gamma_1) \otimes (\mathsf{id} \mathbin{;} \gamma_2)) \mid \ldots & (\textsf{S-Join}),\ \gamma_2 \otimes \langle\rangle \equiv \gamma_2,\ (\textsf{S-Sel-R}) \\
\ \Rightarrow^* \mathsf{id} \otimes \mathsf{id} \mathbin{;} \gamma_1 \otimes \gamma_2 & (\textsf{S-Join}),\ (\textsf{S-Sel-L})
\end{array}$$

Finally, here is the full reduction from Section 4.

$$\begin{array}{ll}
\vdash A \Longrightarrow (B \Longrightarrow A) \mathbin{;} \mathit{stripImp} & \\
\ \Rightarrow\ \ \vdash A \Longrightarrow (B \Longrightarrow A) \mathbin{;} (\mathit{impI} \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid \mathtt{ID} & (\textsf{S-Def}) \\
\ \Rightarrow\ \ (\vdash A \Longrightarrow (B \Longrightarrow A) \mathbin{;} \mathit{impI} \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid (\vdash A \Longrightarrow (B \Longrightarrow A) \mathbin{;} \mathtt{ID}) & (\textsf{S-Alt}) \\
\ \Rightarrow\ \ (\{A\} \vdash B \Longrightarrow A \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid \ldots & (\textsf{S-Atomic}),\ (\textsf{S-Seq-L}) \\
\ \Rightarrow\ \ (\{A\} \vdash B \Longrightarrow A \mathbin{;} (\mathit{impI} \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid \mathtt{ID}) \mid \ldots & (\textsf{S-Def}) \\
\ \Rightarrow\ \ (\{A\} \vdash B \Longrightarrow A \mathbin{;} \mathit{impI} \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid (\{A\} \vdash B \Longrightarrow A \mathbin{;} \mathtt{ID}) \mid \ldots & (\textsf{S-Alt}) \\
\ \Rightarrow\ \ (\{A,B\} \vdash A \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid \ldots \mid \ldots & (\textsf{S-Atomic}),\ (\textsf{S-Seq-L}) \\
\ \Rightarrow\ \ (\{A,B\} \vdash A \mathbin{;} ((\mathit{impI} \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid \mathtt{ID})) \mid \ldots \mid \ldots & (\textsf{S-Def}) \\
\ \Rightarrow\ \ (\{A,B\} \vdash A \mathbin{;} \mathit{impI} \mathbin{;} \mathtt{REPEAT}\ \mathit{impI}) \mid (\{A,B\} \vdash A \mathbin{;} \mathtt{ID}) \mid \ldots \mid \ldots & (\textsf{S-Alt}) \\
\ \Rightarrow^* (\mathsf{id} \mathbin{;} \{A,B\} \vdash A) \mid \ldots \mid \ldots & (\textsf{S-Id}),\ (\textsf{S-Sel-R}) \\
\ \Rightarrow^* \mathsf{id} \mathbin{;} \{A,B\} \vdash A & (\textsf{S-Sel-L}),\ (\textsf{S-Sel-L})
\end{array}$$

The last goal is easily proven with the atomic tactic *ax*.