Querying Proofs (Work in Progress)

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

[1] LFCS, School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland
[2] SGT, NASA Ames Research Center
Moffett Field, CA 94035, USA
[3] Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany

Abstract. We motivate and introduce the basis for a query language designed for inspecting electronic representations of proofs. We argue that there is much to learn from large proofs beyond their validity, and that a dedicated query language can provide a principled way of implementing a family of useful operations.

1 Motivation

Increasingly, automated proof tools and interactive theorem provers are called upon to produce evidence of their claims, in the form of representations of proofs that may be independently checked or, perhaps, imported into another system. These electronic proofs must connect together atomic rules of inference and axioms in a sound way according to an underlying logic. Checking that this has been done correctly is comparatively straightforward (though not without difficulty [15]), although producing the proofs in the first place may have been extraordinarily difficult.

Real proofs can be very large, perhaps consisting of tens or hundreds of thousands of atomic rules of inference. There are many things that are interesting to know about such objects, beyond the basic fact that they are correctly constructed. For example, some natural questions when *inspecting* a proof are:

- What is the high-level structure of this proof, (how) can we break it down into pieces to understand it?
- Given a proof of a property which exploits a set of domain-specific axioms, which axioms actually occurred in the proof? (Or, in a purely logical setting, does a proof rely on axioms of classical logic?)
- Given a problem statement which contains some existential propositions as sub-formulae, which, if any, witnesses were found to make them true?
- Does a large proof contain duplicated parts that could be abstracted (or generalised) into a separate lemma, using a cut-like rule to reduce the size of the proof?

When the user is trying to understand the proof construction process, particularly if using interactive or semi-interactive provers, there are further natural questions which relate the constructed proof back to the procedures that produced it. For example, if *tactics* are our notion of proof producing procedure, there are natural questions to ask which *relate* the proof and the tactics that produced it:

- Given a set of tactics and a proof, which tactics were invoked in producing the proof and how were they invoked (i.e., which subgoals were solved by which tactics)?
- Were any tactics used repeatedly in this proof, perhaps with similar or identical inputs?
- Did some tactics get invoked but do no useful work?
- Given a failed proof (represented as a proof with unproved portions), which tactics were tried on the unproved portions?

These kind of questions are not idle curiosities: we believe that they are useful for practical *proof engineering*, when managing and maintaining sets of properties, proofs and programs which check and create them. For example, one of us (Denney) routinely resorts to low-level scripted tools to perform these kind of examinations when building large safety cases supported by formal proofs [10,5].[4]

One can approach this in a more principled manner with the hope of enabling more general tools with clear foundations. In this short paper, we introduce some first ideas for a *query language* designed for querying proofs.

Hierarchical proofs. We build on the foundation of *hiproofs* [11,2], which provide a simple abstract notion of proof tree. Hiproofs represent proof trees by composing atomic rules of inference from an unspecified underlying logic. Going beyond ordinary trees, they have a notion of *hierarchy*, provided by a way to nest and label a subtree. This simple addition provides a precise and useful notion of *structure* in the proof which can be used, for example, for noting where a lemma was applied, or where a particular tactic or external proof tool produced a subtree.

Sub-proof labelling, when it is present in a proof, immediately allows us to address the first questions above concerning overall proof structure and the application points of tactics. Subtrees provide structuring that can give hints to understanding the constructed proof object. Labels act as reference points to connect back to the proof-producing program. Note, however, that labels are not enough to completely capture the story of how a proof was produced since they only record *success* points, not points where some sub-procedure was attempted but failed to produce a proof. So some forms of query may refer to a proof and

[4] Of course, manipulating proof objects to *change* them is also interesting, although it might sometimes be better done on the *input* to systems that constructs proofs. In this preliminary study we restrict ourselves to queries which return pieces of the queried objects, without further manipulation.

its construction procedure together, or, equivalently, return results by querying the search tree that was explored during its construction.

In practice, of course, practical proof tools already have mechanisms to allow these sort of features. For example, externally invoked procedures may have their results (and perhaps justifications) grafted into an overall proof, or at least recording that they were applied [12]. Noteworthy sub-trees may be represented using names for reference (and then shared to create a dag structure) as in TPTP [16]. In many systems, switches may be used to turn on debugging output for proof procedures to create a lengthy log, which explains where things were tried and failed. Similarly, although there do exist proof assistants which support notions of proof with hierarchical aspects, for example, the proof data structure implemented in Omega [6], these typically include implementation features and are therefore less abstract than hiproofs.

2 Hiproofs

Hiproofs add structure to an underlying *derivation system*, a simple form of logical framework. We give a brief recap here, the reader is referred to [2,11] for full details.

A hiproof is built from (inverted) atomic inference rules a in the underlying derivation system: it maps input goals $[\gamma_1, \ldots, \gamma_n]$ to output subgoals $[\gamma'_1, \ldots, \gamma'_m]$. A nested hiproof, appearing immediately inside a labelled box, always has a single input goal which is the root of the tree at that level.

Informally and graphically, we draw hiproofs as inverted trees with a nested structure. Denotationally, a hiproof can be understood as a pair of a tree and a forest with the same set of nodes, subject to some well-formedness conditions. Syntactically, a hiproof can be written as a term s in the grammar below:

$$\begin{array}{lll} s ::= a & & \text{atomic} \\ \quad \mid \ \mathsf{id} & & \text{identity} \\ \quad \mid \ [l]\, s & & \text{labelling} \\ \quad \mid \ s_1 \ ; \ s_2 & & \text{sequencing} \\ \quad \mid \ s_1 \otimes s_2 & & \text{tensor (juxtaposition)} \\ \quad \mid \ \langle\rangle & & \text{empty} \end{array} \tag{1}$$

Fig. 1 shows the graphic representation of an example hiproof and its term equivalent. Boxes indicate nestings and have labels in their top corners; unlabelled boxes contain atomic rules. Tensor places things side-by-side and sequencing builds "wiring" to connect things together, using identity to create wires where a goal is not manipulated. In the example, id exports the second subgoal from the atomic rule a outside the box labelled l.

Validation. A hiproof is called *valid* if it corresponds to a real proof tree in the underlying derivation system. For example, the hiproof in Fig. 1 validates the proof tree:

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

where we have some input and output goals that can be proved with the atomic inference rules a, b and c. We write $s \vdash g_1 \longrightarrow g_2$ if s is a valid hiproof that takes input goals g_1 and produces output goals g_2.

A valid hiproof can be seen, then, as a nested labelling applied to a flat proof. In [2] we introduced a kernel tactic language which extends hiproofs with the well-known procedural tactic mechanisms for computing proofs: recursion for repetition, alternation for trying one thing or backtracking to another, and testing subgoals to introduce decision points. In [18] this is taken further by providing a declarative tactic language.

3 Queries

One design option would be to take an existing query language for graph (or semi-structured) data models (e.g., see [1] for models and [3] for web query languages), and then map from hiproofs into the existing language and use queries there. We prefer instead to start from queries written in a native query language closer to hiproofs, and give a direct semantics for them. This gives us a clearer idea of what queries we need and helps keeps the semantics precise; to establish bounds on performance (or perhaps for practical implementation) we may consider a translations as secondary.

To begin with, we want a simple query to be able to inspect and return parts of a hiproof. We defer relating proofs to their production mechanisms, the second category of examples in the introduction, for later. Thus queries may return atomic rule names a, labels l, or sub-hiproofs s. These will be selected by *paths* that match the hiproof tree and pick out pieces. Queries are then constructed by *generating* sets of paths using path expressions, and *filtering* with simple propositions to select those of interest.

3.1 Paths

We use hiproof constructors to build up paths. A path navigates down through the structure, choosing left and right branches of tensors, and entering boxes, until hitting a chosen point. So the hiproof constructors themselves can serve as labels.

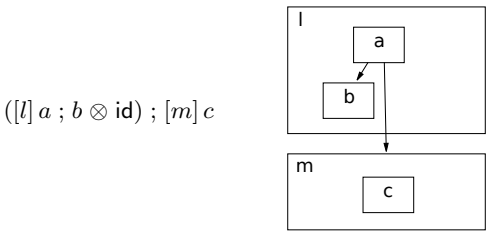

Fig. 1. A hiproof and its graphical representation.

Definition 1 (Path). *A path is denoted as follows:*

$$p ::= \bullet \mid [-]\,p \mid p \otimes - \mid - \otimes p \mid p\,;- \mid -\,;p$$

A path selects a part of a hiproof (Def. 3 below), if the shape of the hiproof fits with the path; in this case we say that the path is well-formed.

Definition 2. *Given a hiproof s, the set $\mathbf{P}(s)$ of all paths well-formed wrt s is defined recursively as follows:*

$$\begin{aligned}
\mathbf{P}(a) &= \{\bullet\} \\
\mathbf{P}(\mathsf{id}) &= \{\} \\
\mathbf{P}([l]\,s) &= \{\bullet\} \cup \{[-]\,q \mid q \in \mathbf{P}(s)\} \\
\mathbf{P}(s_1 \otimes s_2) &= \{\bullet\} \cup \{p \otimes - \mid p \in \mathbf{P}(s_1)\} \cup \{- \otimes p \mid p \in \mathbf{P}(s_2)\} \\
\mathbf{P}(s_1\,;\,s_2) &= \{\bullet\} \cup \{p\,;- \mid p \in \mathbf{P}(s_1)\} \cup \{-\,;p \mid p \in \mathbf{P}(s_2)\}
\end{aligned}$$

A well-formed path selects a sub-hiproof of a given hiproof, called its *target*, defined in the obvious way as follows:

Definition 3 (Selection). *For a hiproof s and a path $p \in \mathbf{P}(s)$, the* target *of p is defined as a selection of s as follows:*

$$\begin{aligned}
\mathit{sel}(\bullet, s) &= s \\
\mathit{sel}([-]\,p, [l]\,s) &= \mathit{sel}(p, s) \\
\mathit{sel}(p \otimes -, s_1 \otimes s_2) &= \mathit{sel}(p, s_1) \\
\mathit{sel}(- \otimes p, s_1 \otimes s_2) &= \mathit{sel}(p, s_2) \\
\mathit{sel}(p\,;-, s_1\,;\,s_2) &= \mathit{sel}(p, s_1) \\
\mathit{sel}(-\,;p, s_1\,;\,s_2) &= \mathit{sel}(p, s_2)
\end{aligned}$$

A simple but worthwhile observation is that selection preserves validity.

Lemma 1 (Validity Preservation). *Given a validated hiproof $s \vdash g \longrightarrow_1 g_2$, for all $p \in \mathbf{P}(s)$, there are goals h_1, h_2 such that $\mathit{sel}(p, s) \vdash h_1 \longrightarrow h_2$.*

This is proven by inspecting the validation of the hiproof. Given a validation, we can extend $\mathit{sel}(p, s)$ to return the concrete lists of goals h_1 and h_2 discharged and recharged by s. In particular, this allows us to inspect subgoals inside the proof, or check the *arity* of a sub-proof or atomic rule. An atomic rule a has an *input arity* n given by its number of premises, written $a : n$. Axioms have zero input, so $a : 0$ says that a is an axiom.

Operations and propositions on paths give us the *path algebra*.

Definition 4 (Path concatenation). *For two paths p and q, their concatenation $p +\!\!+ q$ is defined in the obvious way:*

$$\begin{aligned} \bullet +\!\!+ q &= q \\ [-]\, p +\!\!+ q &= [-]\, (p +\!\!+ q) \\ p \otimes - +\!\!+ q &= (p +\!\!+ q) \otimes - \\ - \otimes p +\!\!+ q &= - \otimes (p +\!\!+ q) \\ p\,;\, - +\!\!+ q &= (p +\!\!+ q)\,;\, - \\ -\,;\, p +\!\!+ q &= -\,;\, (p +\!\!+ q) \end{aligned}$$

Concatenation is associative and has the empty path $\bullet$ as left and right unit.

3.2 Queries

A query is an operation which selects (interesting) pieces of a hiproof, given by one or more paths. Queries are built using comprehension schemes of first-order logic over an algebra of schemes and paths.

To be precise, let Var_A, Var_L, Var_S and Var_P be disjoint, countably infinite sets of variables for atomics, labels, hiproofs and paths, respectively, ranged over by the indicated capital letters. The *hiproof expressions* are hiproofs built over atomic, label and hiproof variables, using the operations in (1) and the selection operation from Def. 3 above. The *path expressions* are built using paths, path variables and the path operation $+\!\!+$. The *path propositions* are expressions of first-order logic over equations between path expressions, hiproof expressions, or atomic propositions that constrain atomic goals or atomic rules.

An example of an expressible useful derived property is the prefix ordering between paths:

$$p \leq q \iff \exists r.\, p +\!\!+ r = q.$$

A simple query is defined as a set comprehension scheme

$$\{P \in \mathbf{P}(s) \mid \phi(P)\}$$

where s is the hiproof to query and $\phi(P)$ is a path proposition selecting the interesting paths, P. More complex queries can have multiple generating expressions. Most of our queries return atomic tactics or labels, though, so we allow the following extensions. For paths to return atomic tactics, we have

$$\{A \mid P \in \mathbf{P}(s), \phi(A, P, s)\} = \{sel(P, s) \mid P \in \mathbf{P}(s) \wedge \exists A\,.\, sel(P, s) = A \wedge \phi(A, s)\}$$

where $\phi(A, P, s)$ is a proposition over an atomic tactic A, a path P and the hiproof s. Thus, we can write a query which returns a set of atomic tactics which is a shortcut for a query which returns a set of paths guaranteed to select an atomic tactic.

For the examples we give below, we constrain atomic rules by specifying a particular subset we want to choose (for example, those that are axioms or those that prove existential statements). Other examples examine concrete goals that appear in the proof (hiproof validation). In the Hitac tactic language [2] we used a matching relation $\mathsf{assert}\ \phi$ as an abstract constraint on goals γ, similarly.

3.3 Example queries

Finally we illustrate our ideas with a few examples that show how to answer some of the questions posed in Section 1.

- To find all axioms in a valid hiproof s:

$$\mathit{Axioms}(s) = \{A \mid P \in \mathbf{P}(s).\ \mathit{sel}(P, s) = A \wedge A : 0\}.$$

- To find existential witnesses inside a valid hiproof s, we suppose that the introduction rule for the existential ex_I is a set of atomic tactics $ex_I = \{ex_{I\,t}\}_{t \in \mathcal{T}}$ indexed by a set of terms $\mathcal{T}$ (witnesses) in the underlying logic. The witness query returns the instantiated existential rules:

$$\mathit{Wit}(s) = \{A \mid P \in \mathbf{P}(s).\ \mathit{sel}(P, s) = A \wedge A \in ex_I\}$$

- Which goals are input to (or output from) a tactic called tac?

$$\mathit{Input}(\mathsf{tac}, s) = \{g \mid P \in \mathbf{P}(s).\ \exists S_1.\mathit{sel}(P, s) = [\mathsf{tac}]\, S_1 \wedge S_1 \ \vdash\ g \longrightarrow h\}$$

$$\mathit{Output}(\mathsf{tac}, s) = \{h \mid P \in \mathbf{P}(s).\ \exists S_1.\mathit{sel}(P, s) = [\mathsf{tac}]\, S_1 \wedge S_1 \ \vdash\ g \longrightarrow h\}$$

- Which tactics calls themselves recursively? Note how this query has two generating expressions $P \in \mathbf{P}(s)$ and $Q \in \mathbf{P}(s)$:

$$\begin{aligned}\mathit{Rec}(s) = \{L \mid\ & P \in \mathbf{P}(s), Q \in \mathbf{P}(s).\ P \leq Q \wedge \\ & \exists S_1.\ \mathit{sel}(P, s) = [L]\, S_1 \wedge \exists S_2.\ \mathit{sel}(Q, s) = [L]\, S_2\}\end{aligned}$$

  This returns labels l which label subtrees that contain the same label l again.
- Which tactic uses atomic tactic a, i.e., inside which label does a occur? This query returns all labels L which contain a directly, i.e., there are no other labels inside boxes containing labels in L.

$$\begin{aligned}\mathit{Inside}(a, s) = \{L \mid\ & P \in \mathbf{P}(s), Q \in \mathbf{P}(s), R \in \mathbf{P}(s). \\ & P \leq Q\ \wedge\ P \leq R\ \wedge\ R \leq Q\ \Rightarrow \\ & \exists S_1.\ \mathit{sel}(P, s) = [L]\, S_1\ \wedge \mathit{sel}(Q, s) = a \wedge \\ & \quad \neg\exists M, S_2.\ \mathit{sel}(R, s) = [M]\, S_2\}.\end{aligned}$$

4 Future Work

This brief paper introduces some of our ideas for proof query languages. Much remains to be done: we plan to first complete our study of the semantics for the query constructs, and then to introduce a more user-friendly language for actually writing queries, using the above comprehension schemes to give their denotation. Then we need to give an account of how queries are evaluated: this might be with a direct operational interpretation, or via translation to an auxiliary metalanguage. Further out, we want to set this work in the context of related query languages, perhaps by translations as suggested above. See, e.g., [4] for some expressivity and complexity results.

Meanwhile, we are also keen to explore moving the hiproof formalisation closer to usable implementations; not to replace incumbent systems with their large machinery and proof libraries, but to serve as an experimental platform for studying proof languages more precisely. See [18] for an example in this direction, describing a declarative language for hiproofs and also some *refactoring* operations to model changes undertaken in real proof developments. Such refactorings cause input changes to proof tools that don't change statements being proved, but may alter resultant proof objects or their structure.

It is the overall goal of our work to provide an abstract *metalanguage* which can be used to represent proofs and proof manipulation in a prover-independent way. The next step, therefore, will be to develop a mapping between our language and concrete proof frameworks, such as TPTP.

Related work. We believe that the idea of a dedicated query language for inspecting proofs is novel, although there are some related investigations on particular ways of exploiting proofs. These include, for example, efforts to *translate proofs* between systems [9]; ways to *discover dependencies* between parts of proofs [14] to help simplify or rearrange; and ways to *mine proofs* to discover common patterns [17]. Away from theorem proving, query languages have been introduced for other forms of structured data, including semi-structured (XML-like) models [7], and programs or their intermediate forms during compilation [8,13].

References

1. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40:1:1–1:39, February 2008.
2. D. Aspinall, E. Denney, and C. Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3(3):309–330, 2010.
3. J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and semantic web query languages: A survey. In N. Eisinger and J. Maluszynski, editors, *Reasoning Web*, volume 3564 of *Lecture Notes in Computer Science*, pages 95–95. Springer Berlin / Heidelberg, 2005.
4. P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In J. Paredaens and D. V. Gucht, editors, *PODS*, pages 3–14. ACM, 2010.

5. N. Basir, E. Denney, and B. Fischer. Deriving safety cases for hierarchical structure in model-based development. In *The 29th International Conference on Computer Safety, Reliability and Security (SafeComp '10)*, Vienna, Austria, 2010.
6. C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In *Proceedings of CADE-14*, volume 1249 of *LNAI*. Springer, 1997.
7. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610. Springer, 2002.
8. R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *DSL*. USENIX, 1997.
9. E. Denney. A prototype proof translator from HOL to Coq. In M. Aagaard and J. Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.
10. E. Denney and B. Fischer. A verification-driven approach to traceability and documentation for auto-generated mathematical software. In *Automated Software Engineering (ASE '09)*, 2009.
11. E. Denney, J. Power, and K. Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359, 2006.
12. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998. 10.1023/A:1006023127567.
13. S. Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Trans. Software Eng.*, 24(3):197–215, 1998.
14. O. Pons, Y. Bertot, and L. Rideau. Notions of dependency in proof assistants. In *Proc. User Interfaces for Theorem Provers, UITP'98*, 1998.
15. G. Sutcliffe, E. Denney, and B. Fischer. Practical proof checking for program certification. In *Proceedings of the CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR'05)*, July 2005.
16. G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP language for writing derivations and finite interpretations. In U. Furbach and N. Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 67–81. Springer Berlin / Heidelberg, 2006.
17. J. Urban. MizarMode—-an integrated proof assistance tool for the Mizar way of formalizing mathematics. *J. Applied Logic*, 4(4):414–427, 2006.
18. I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards formal proof script refactoring. In *Proceedings MKM 2011*, 2011. To appear.