Querying Proofs

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 a query language *PrQL* designed for inspecting machine representations of proofs. PrQL natively supports *hiproofs* which express proof structure using hierarchical nested labelled trees. The core language presented in this paper is *locally structured*, with queries built using recursion and patterns over proof structure and rule names. We define the syntax and semantics of locally structured queries, demonstrate their power, and sketch some implementation experiments.

1 Introduction

Automated proof tools and interactive theorem provers are increasingly required to produce evidence of their claims as formal proof objects that may be independently checked or, perhaps, imported into other systems or transformed in particular ways. Proofs 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 essentially straightforward, although producing a proof in the first place may be 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?
- Given a problem statement which contains some existential propositions as sub-formulae, which, if any, witnesses were found to make them true?

* Research supported by EPSRC grant EP/J001058/1.
** Research supported by NASA contract NNA10DE83C.
*** Research supported by BMBF grants 01IS09044B (IGEL) and 01IW10002 (SHIP).

- Does a large proof contain duplicated parts that could be abstracted into a lemma, to reduce the size of the proof?

When the user is trying to understand the proof construction process, there are natural questions which relate the constructed proof back to the procedures that produced it. If tactics are our notion of proof producing procedure, some questions *relating* the proof to the tactics that produced it are:

- Given a set of tactics and a proof, which tactics were invoked in producing the proof and what subgoals did they solve?
- Were any tactics used recursively?
- Does one particular tactic always lead to another being invoked?
- Did some tactics get invoked but do no useful work?

These sort of questions are not idle curiosities: they are useful for practical *proof engineering*, when managing and maintaining sets of properties, proofs and programs which create and check them. 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.

We consider querying proofs here in a rigorous, generic manner with the hope of enabling general tools with clear foundations. In this paper, we introduce the basis of a query language *PrQL* designed specifically for querying proofs.

Hierarchical structured proofs. The foundation we start from is *hiproofs* [1,2], which provide a simple abstract notion of proof tree by composing atomic rules of inference from an unspecified underlying logic. Going beyond ordinary trees, they have a notion of *hierarchy*, by allowing labelling and nesting of subtrees. 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.

Contributions and paper outline. This paper contributes towards generic foundational aspects of theorem proving systems. Query languages for tree and graph structured data have been studied over the last decade or so, but have rarely been applied to formal proofs. We design a new core proof query language from first principles, directly connected with a precise abstract notion of proof. With motivating examples and implementation experiments, we establish its utility.

The rest of this paper is structured as follows. Section 2 introduces the foundation of hiproofs used in the rest of the paper. Section 3 describes the design decisions we took for our query language, and introduces it with a sequence of informal examples and their intended meanings. Section 4 describes the meaning of queries formally, so one can check that example queries indeed have the desired meanings; it also provides a baseline decidability result. In Section 5 we sketch a simple prototype implementation, which we use to validate our language design; full-scale experiments on large proofs remain as future work. We mention some of our future plans and discuss some related work in the concluding Section 6.

2 Hiproofs

Hiproofs add structure to an underlying *derivation system*, a simple form of logical framework. We give a brief recap here, for fuller details please see [1,2].

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

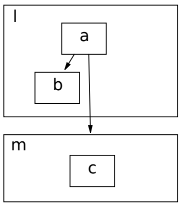
Informally and graphically, we draw hiproofs as inverted trees with a nested structure. Denotationally, a hiproof can be understood as a pair of an ordered tree and a forest with the same set of nodes, subject to some well-formedness conditions. Syntactically, a hiproof can be written as a term s in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

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

Fig. 1 shows an example hiproof term and its graphical representation in the middle. Boxes indicate nestings and have labels in their top corners, indicating the tactic which gave rise to the contents in the box; unlabelled boxes contain atomic rules. Tensor $\otimes$ places hiproofs side-by-side and sequencing ; builds "wiring" to connect hiproofs 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. The empty proof $\langle\rangle$ is useful when building proofs programmatically.

Valid hiproofs. A hiproof is called *valid* if it corresponds to a real proof tree in the underlying derivation system. The hiproof term in Fig. 1 validates the proof tree shown on the right-hand side, where an input goal γ_1 is proved using the atomic inference rules a, b and c. Validity extends naturally to arbitrary hiproof terms that have more than one input goal; such a term corresponds to a finite sequence of proof trees. We write $s \ \vdash \ g_1 \longrightarrow g_2$ if s is valid in this more

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



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

Fig. 1. A hiproof, its graphical representation and a proof it validates.

$$\dfrac{\frac{\gamma_1\cdots\gamma_n}{\gamma}\,a \text{ is an atomic inference}}{a \vdash \gamma \longrightarrow [\,\gamma_1,\ldots,\gamma_n\,]} \qquad \dfrac{}{\mathsf{id} \vdash \gamma \longrightarrow \gamma} \qquad \dfrac{s \vdash \gamma \longrightarrow g}{[l]\, s \vdash \gamma \longrightarrow g} \qquad \dfrac{}{\langle\rangle \vdash [\,] \longrightarrow [\,]}$$

$$\dfrac{s_1 \vdash g_1 \longrightarrow g \qquad s_2 \vdash g \longrightarrow g_2}{s_1 \,;\, s_2 \vdash g_1 \longrightarrow g_2} \qquad \dfrac{s_1 \vdash g_1 \longrightarrow g_1' \qquad s_2 \vdash g_2 \longrightarrow g_2'}{s_1 \otimes s_2 \vdash g_1 {}^\wedge g_2 \longrightarrow g_1' {}^\wedge g_2'}$$

Fig. 2. Validation of hiproofs (the symbol $^\wedge$ stands for list append).

general sense, taking a list of input (proven) goals g_1 to produce a list of output (unsolved) goals g_2. This relation is defined by the rules in Fig. 2.

Validity checking can be seen as a way of adding goals to a hiproof; correspondingly, a valid hiproof can be seen as a nested labelling applied to a flat proof. A hiproof thus represents the outcome of a proof process rather than the method by which it was obtained, and is independent of the direction (forwards from axioms or backwards from conjecture) of construction. In this paper we restrict our attention to valid hiproofs and we assume that the goals are uniquely determined by the validated hiproof.

3 Local structured queries

How should we express queries on proofs such as those in Sect. 1? One design choice would be to take an existing query language for graph (or semi-structured) data models (e.g., see surveys [3,4]), and then map from hiproofs into the existing language and use queries there. The drawback with that approach is that we immediately lose connection with our particular source language. Since our initial aim is to understand the concepts and constructs specific to querying proofs, rather than more general objects, we start from queries written in a minimal native query language, and investigate a direct semantics for them.

Our queries follow the hiproof structure, matching on leaves with atomics, structured proofs using labels, or on input or output goals of subproofs. In this paper, we consider queries that specify structure *locally*, in the sense that they cannot directly compare one part of the tree with another, or measure absolute position within the global proof. This restriction arises intentionally, because we use only first-order variables that refer to names and goals, not to subtrees or paths. Despite this, the language is still rather expressive and captures our desired queries fairly succinctly, so it is a good candidate core query language.

To introduce the language, we begin with constructs for matching leaves, boxes and goals in proofs, and then build up following the hiproof syntax.

Matches. We build *matches* inside queries using wildcards and match variables, constants (atoms, sets and predicates) and negation (to construct the complement of a match). Let Var_N be a set of schematic variables standing for names, ranged over by N in general and A when we suggest an atomic rule name or L a label name. Let Var_G be a set of variables standing for lists of goals. The name matches and goal matches are given by:

$$\begin{array}{l}
\mathit{nm} ::= a \mid l \mid * \mid \xi \mid N \mid \neg \mathit{nm} \\
\mathit{gm} ::= [\psi_1, \ldots, \psi_n] \mid G \mid \neg \mathit{gm}
\end{array}$$

where ξ stands for a logic-dependent predicate on names, and ψ stands for a logic-dependent predicate on goals used to check some structural property of the goal term. For example we might have a predicate that checks whether a goal γ is in the form of a horn clause, when $\phi_{hornclause}(\gamma)$ holds. Most simply, we suppose that we always have a predicate to check for equality with any specific goal γ and we overload γ to stand for that predicate.

We use matches to build up the *basic queries* that specify local structure. Informally, a basic query may hold for a given hiproof and a substitution of variables the query contains; we will define the result of a query to be the set of variable instantiations that make it true. As (merely) a matter of style, we use a verbose SQL-like textual notation:

$$\begin{array}{rll}
q ::= & * & \text{anything non-empty} \\
\mid & \textbf{atomic}\ \mathit{nm} & \text{atomic rule match} \\
\mid & \textbf{nothing} & \text{nothing (matches only identity)} \\
\mid & \textbf{inside}\ \mathit{nm}\ q & q \text{ satisfied inside box with label matching} \\
\mid & q_1\ \textbf{then}\ q_2 & q_1 \text{ and } q_2 \text{ satisfied by successive nodes in } ; \\
\mid & q_1\ \textbf{beside}\ q_2 & q_1 \text{ and } q_2 \text{ satisfied by adjacent nodes in } \otimes \\
\mid & \textbf{ingoals}\ \mathit{gm} & \text{goals into sub-proof match} \\
\mid & \textbf{outgoals}\ \mathit{gm} & \text{goals out of sub-proof match}
\end{array}$$

Basic queries are almost the same language as the hiproof syntax itself, omitting empty proofs and adding the ability to match on goals within. Thus, phrases act as structural patterns matching against an implicit hiproof subject.

For the hiproof given in Fig. 1, the following queries are each satisfied (the alignment around **then** matches the vertical split):

$$\begin{array}{rl}
(\textbf{inside}\ \mathsf{l}\ *) & \textbf{then}\ (\textbf{inside}\ \mathsf{m}\ *) \\
(\textbf{inside}\ *\ *\ \textbf{then}\ *\ \textbf{beside nothing}) & \textbf{then}\ * \\
(\textbf{inside}\ L_1\ *) & \textbf{then}\ (\textbf{inside}\ *\ \textbf{atomic}\ A)
\end{array}$$

The first two are purely structural, matching the form of the tree. The first matches the outer structure consisting of the box labelled l followed by the box labelled m. The second examines the shape inside the first box. The final query is satisfiable with the unique instantiation $\{L_1 \mapsto \mathsf{l}, A \mapsto \mathsf{c}\}$.

Connectives. We allow propositional logical connectives to build compound queries, with familiar intended meanings:

$$q \ ::= \ldots \mid q_1 \wedge q_2 \mid q_1 \vee q_2 \mid \neg q$$

Search and check. Two important quantifier combinators on queries allow us to search within a proof for somewhere that a query is satisfied, or check that a query is satisfied everywhere.

$$\begin{array}{lll} q ::= \ldots & & \\ \quad \mid \ \textbf{somewhere}\ q & & q \text{ holds in some subproof} \\ \quad \mid \ \textbf{everywhere}\ q & & q \text{ holds in every subproof} \end{array}$$

With a syntactic interpretation, the natural domain of quantification is by subterm; because any subterm of a valid hiproof is also valid, this makes sense and we take "subproof" to mean subterm. The scope of **somewhere** and **everywhere** extends as far right as possible. These queries might be added directly to the language, but we will define them instead using recursion (introduced below).

The **somewhere** combinator is used in many of our examples. For example, a proof uses a tactic tac if the query

$$\textbf{somewhere inside}\ \mathsf{tac}\ *$$

is satisfied. As another example, we use a match on a goal-list variable G to find the goals passed into a tactic. The query

$$(\textbf{somewhere inside}\ \mathsf{m}\ \textbf{ingoals}\ G) \vee (\textbf{somewhere atomic}\ \mathsf{b} \wedge \textbf{ingoals}\ G)$$

can be read as "tell me the goals that are input to tactic m or the atomic rule b". The result should be the pair of instantiations $\{G \mapsto [\gamma_2]\}, \{G \mapsto [\gamma_3]\}$ for the hiproof in Fig. 1.

When is **everywhere** useful? Not for anything that requires a fixed structure, but with a goal-matching assertion that checks the format of the goals, for example, the check **everywhere outgoals** $[\phi_{hornclause}]$ requires that every goal appearing in the tree must have that certain form. With conditional queries, we use it to specify that goals appearing in certain places must have some property.

Recursive queries. Just as with tactics we can allow recursively defined queries. Recursively defined queries allow us to build up regular patterns and are defined using query variables Q:

$$q \ ::= \ldots \ \mid \mu Q.q$$

where q is a query in which Q can appear free. An example recursive pattern is:

$$\mu Q.\,(\textbf{atomic}\ \mathsf{a}\ \textbf{then}\ (\textbf{ingoals}\ [\gamma_2]\ \textbf{beside}\ Q)) \vee (\textbf{inside}\ \mathsf{m}\ *)$$

which is satisfied by proofs that repeatedly apply the atomic rule a, until reaching a box named m.

Using recursion we can define the searching and checking quantifiers:

$$\begin{aligned} \textbf{somewhere}\ q \overset{def}{=} \mu Q.\,q \vee\ & (\textbf{inside}\ *\ Q) \vee (Q\ \textbf{then}\ *) \vee (*\ \textbf{then}\ Q) \vee \\ & (Q\ \textbf{beside}\ *) \vee (*\ \textbf{beside}\ Q) \\ \textbf{everywhere}\ q \overset{def}{=} \mu Q.\,q \wedge\ & (\textbf{atomic}\ *\ \vee \textbf{nothing} \vee (\textbf{inside}\ *\ Q) \vee \\ & (Q\ \textbf{then}\ Q) \vee (Q\ \textbf{beside}\ Q)) \end{aligned}$$

these ensure that q holds at one (or every) node following the structure of the proof; notice that exactly one of the disjuncts must hold in the recursive cases. Later on we will show that these definitions have the intended meaning.

Derived forms. Using this core, we can readily add more derived forms:

$$\begin{array}{rlrl}
q_1 \;\mathbf{when}\; q_2 & \overset{def}{=} \neg q_2 \vee q_1 & \mathbf{provesgoal}\;\gamma & \overset{def}{=} \mathbf{ingoals}\;[\gamma] \wedge \mathbf{outgoals}\;[] \\
\mathbf{isthen} & \overset{def}{=} * \;\mathbf{then}\; * & \mathbf{axiom}\; nm & \overset{def}{=} \mathbf{atomic}\; nm \wedge \mathbf{outgoals}\;[] \\
\mathbf{isbeside} & \overset{def}{=} * \;\mathbf{beside}\; * & \mathbf{islabel}\; nm & \overset{def}{=} \mathbf{inside}\; nm\; *
\end{array}$$

$$\begin{array}{rl}
\mathbf{whenin}\; nm\; q & \overset{def}{=} \mathbf{inside}\; nm\; q \;\mathbf{when}\;\mathbf{islabel}\; nm \\
\mathbf{somewherebeside}\; q & \overset{def}{=} \mu Q.\, q \vee (Q \;\mathbf{beside}\; *) \vee (* \;\mathbf{beside}\; Q) \\
\mathbf{nearby}\; q & \overset{def}{=} \mu Q.\; q \vee (Q \;\mathbf{then}\; *) \;\vee\; (* \;\mathbf{then}\; Q) \\
 & \qquad\qquad \vee (Q \;\mathbf{beside}\; *) \;\vee\; (* \;\mathbf{beside}\; Q) \\
\mathbf{separately}\; q_1 \;\mathbf{and}\; q_2 & \overset{def}{=} \mu Q.\; (\mathbf{inside}\; *\; Q) \\
 & \qquad\quad \vee (\mathbf{somewhere}\; q_1 \;\mathbf{then}\;\mathbf{somewhere}\; q_2) \\
 & \qquad\quad \vee (\mathbf{somewhere}\; q_1 \;\mathbf{beside}\;\mathbf{somewhere}\; q_2)
\end{array}$$

The **when** conditional combinator is satisfied if q_1 is satisfied whenever q_2 is; by convention, the scope of q_1 and q_2 extend as far as possible. The last three combinators again use recursion to expand the scope of the local structure specifications. The query **somewherebeside** q is satisfied if q is satisfied in a $\otimes$-list of hiproofs; **nearby** q is an adjusted version of **somewhere** which restricts to the same level, without descending into boxes. The query **separately** q_1 **and** q_2 requires that q_1 and q_2 hold on disjoint portions of the proof.

3.1 Examples

We show some of our motivating examples relating proofs and tactics. First, the tactic tac occurs recursively in a hiproof if the query

$$\mathbf{somewhere}\;\mathbf{inside}\;\mathsf{tac}\;\mathbf{somewhere}\;\mathbf{islabel}\;\mathsf{tac}$$

is satisfied. The tactic inner always occurs whenever the tactic outer is invoked if this query is satisfied:

$$\mathbf{everywhere}\;\mathbf{whenin}\;\mathsf{outer}\;\mathbf{somewhere}\;\mathbf{islabel}\;\mathsf{inner}.$$

More elaborately, a tactic named base always appears alongside a tactic named step inside the tactic induct:

$$\begin{array}{l}
\mathbf{everywhere}\;\mathbf{whenin}\;\mathsf{induct}\;\mathbf{somewhere}\;(\mathbf{somewherebeside}\;\mathbf{islabel}\;\mathsf{base}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\;(\mathbf{somewherebeside}\;\mathbf{islabel}\;\mathsf{step}).
\end{array}$$

Examples returning results are given in Sect. 4.1 after introducing the semantics.

4 Semantics

We will define the semantics of queries using a satisfaction relation $s \models_\sigma q$. This denotes satisfaction of a query on a hiproof s with respect to a substitution σ for match variables. The substitution maps variables N to names for atomic tactics and labels, and variables G to lists of the form $[\gamma_1, \ldots, \gamma_n]$.

Two base satisfaction relations define matching on names and goal lists:

$$
\begin{array}{rcll}
* & \models_\sigma n & & \text{always} \\
n' & \models_\sigma n & \text{iff} & n = n' \\
\xi & \models_\sigma n & \text{iff} & \xi(n) \\
N & \models_\sigma n & \text{iff} & \sigma(N) = n \\
(\neg N) & \models_\sigma n & \text{iff} & \neg(N \models_\sigma n)
\end{array}
\qquad
\begin{array}{rcll}
[\psi_1, \ldots, \psi_n] & \models_\sigma g & \text{iff} & \exists \gamma_1 \cdots \gamma_n .\, g = [\gamma_1, \ldots, \gamma_n] \\
 & & & \quad \text{and } \psi_1(\gamma_1) \cdots \psi_n(\gamma_n) \\
G & \models_\sigma g & \text{iff} & \sigma(G) = g \\
(\neg G) & \models_\sigma g & \text{iff} & \neg(G \models_\sigma g)
\end{array}
$$

Before giving the main relation, we consider hiproof terms in more detail. Terms s in the hiproof grammar denote tree-based models in the denotational semantics of hiproofs [1]. Under the denotational interpretation, certain terms are equivalent. We will give our interpretation over the syntax, considering valid hiproofs modulo the following equations generating this equivalence:

$$
\begin{array}{lll}
s \,;\, \mathsf{id} = s & \mathsf{id} \,;\, s = s & \mathsf{id} \text{ is an identity for sequencing} \\
s \otimes \langle\rangle = s & \langle\rangle \otimes s = s & \langle\rangle \text{ is an identity for juxtaposition} \\
\multicolumn{2}{c}{s \,;\, \langle\rangle = s} & \langle\rangle \text{ is a right-identity for sequencing} \\
\multicolumn{2}{c}{s_1 \,;\, (s_2 \,;\, s_3) = (s_1 \,;\, s_2) \,;\, s_3} & ; \text{ is associative} \\
\multicolumn{2}{c}{s_1 \otimes (s_2 \otimes s_3) = (s_1 \otimes s_2) \otimes s_3} & \otimes \text{ is associative} \\
\multicolumn{2}{c}{(s_1 \,;\, s_2) \otimes (s_3 \,;\, s_4) = (s_1 \otimes s_3) \,;\, (s_2 \otimes s_4)} & ; \text{ and } \otimes \text{ can be exchanged}
\end{array}
$$

It is easy to confirm that the equations preserve validity on the same lists of input and output goals for the rules in Fig. 2. We will write $s = s'$ if two terms are equal in the theory generated by these equations (i.e., closing under congruence).

Definition 1 (Query satisfaction). *Let s be a valid hiproof and q a query in the minimal query language. The satisfaction of q for s with the substitution σ is defined as the least relation $s \models_\sigma q$ satisfying:*

$$
\begin{array}{rl@{\quad}l@{\quad}l}
s & \models_\sigma * & \textit{when} & s \neq \langle\rangle \\
a & \models_\sigma \textbf{atomic}\ nm & \textit{when} & nm \models_\sigma a \\
\mathsf{id} & \models_\sigma \textbf{nothing} & & \\
[l]\, s & \models_\sigma \textbf{inside}\ nm\ q & \textit{when} & nm \models_\sigma l \textit{ and } s \models_\sigma q \\
s_1 \,;\, s_2 & \models_\sigma q_1\ \textbf{then}\ q_2 & \textit{when} & s_1 \models_\sigma q_1 \textit{ and } s_2 \models_\sigma q_2 \\
s_1 \otimes s_2 & \models_\sigma q_1\ \textbf{beside}\ q_2 & \textit{when} & s_1 \models_\sigma q_1 \textit{ and } s_2 \models_\sigma q_2 \\
s & \models_\sigma \textbf{ingoals}\ gm & \textit{when} & gm \models_\sigma g \textit{ where } s \vdash g \longrightarrow h \\
s & \models_\sigma \textbf{outgoals}\ gm & \textit{when} & gm \models_\sigma h \textit{ where } s \vdash g \longrightarrow h \\
s & \models_\sigma q_1 \wedge q_2 & \textit{when} & s \models_\sigma q_1 \textit{ and } s \models_\sigma q_2 \\
s & \models_\sigma q_1 \vee q_2 & \textit{when} & s \models_\sigma q_1 \textit{ or } s \models_\sigma q_2 \\
s & \models_\sigma \neg q & \textit{when} & \neg(s \models_\sigma q) \\
s & \models_\sigma \mu Q.q & \textit{when} & s \models_\sigma q[\mu Q.q/Q] \\
s & \models_\sigma q & \textit{when} & \exists s'.\, s' \models_\sigma q \textit{ and } s' = s.
\end{array}
$$

Recursive queries $\mu Q.q$ are interpreted using unfolding; this suffices since we query only finitely deep trees. More precisely, we can define satisfaction using an auxiliary relation $\models_n$ indexed by the maximum depth of the number of unfoldings of a recursive query, where $\mu_n Q.q$ can be unfolded at most n times. Then $\models$ is defined as the union of all finite unfolding relations $\models_n$. The definition works for singly recursive queries where we do not need to interpret queries with free query variables, but can be extended for mutually recursive queries.

Proposition 1. *Let s be a valid hiproof. Then*

1. $s \models_\sigma$ **somewhere** q *iff* $\exists s'.s'$ *is a subterm of* s *and* $s' \models_\sigma q$,
2. $s \models_\sigma$ **everywhere** q *iff* $\forall s'.s'$ *is a subterm of* s *and* $s' \models_\sigma q$.

(where quantification ranges over non-empty terms, and s is a subterm of itself).

Thus these important derived forms have the intended meanings.

How precise are our queries? The following proposition establishes, as intended, that every term can be characterised up to equality by a query. Thus, we can use queries to describe finite sets of hiproofs.

Proposition 2. *Given any hiproof s not containing $\langle\rangle$, there is a query $Q(s)$ which characterises s precisely.*

Proof. *Let $Q(s)$ be given by the embedding:*

$$\begin{aligned}
Q(a) &= \textbf{atomic}\ a \\
Q(\mathsf{id}) &= \textbf{nothing} \\
Q([l]\, s) &= \textbf{inside}\ l\ Q(s) \\
Q(s_1 \mathbin{;} s_2) &= Q(s_1)\ \textbf{then}\ Q(s_2) \\
Q(s_1 \otimes s_2) &= Q(s_1)\ \textbf{beside}\ Q(s_2)
\end{aligned}$$

Now we claim that whenever $s' \models_\sigma Q(s)$ for some s', we must have $s = s'$.

Using a simple normal form, Prop. 2 can be extended to cover all hiproofs.

4.1 Examples and their results

Now we demonstrate the remainder of our motivating queries; meanings can be calculated using the semantics above to show that they are correct. The invocation of a query to get some results can be written in SQL style as:

$$\textbf{select}\ e\ \textbf{from}\ s\ \textbf{where}\ q$$

which denotes the set of expressions $\sigma(e)$ for all substitutions σ that satisfy the query (see Sect. 5 on how this can be implemented). That is:

$$\{\sigma(e) \mid s \models_\sigma q\}.$$

The kind of expressions e chosen here depends on what we want to do with query results. We don't consider a general transformation language for query results here, but one could easily allow expressions that combine pieces of query results in arbitrary ways. Our examples below restrict to simple query variables.

- To find all the axioms in a valid hiproof s:

  $$\begin{aligned} \mathit{Axioms}(s) = {} & \textbf{select } A \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere axiom } A \end{aligned}$$

  Applied to $s = ([l]\, a \,;\, b \otimes \mathsf{id}) \,;\, [m]\, c$, this query returns $\{A \mapsto \mathsf{c}, A \mapsto \mathsf{b}\}$.
- To find the existential witnesses inside a valid hiproof s, we can find uses of the existential introduction rule:

  $$\begin{aligned} \mathit{Wit}(s) = {} & \textbf{select } A \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere atomic } A \wedge \textbf{atomic } \mathit{ExI}_t \end{aligned}$$

  Here, the *ExI* rule is annotated by the witness t that is chosen as part of its name, and we use ExI_t to denote the predicate selecting all such rule names.
- Which tactics are used in a proof?

  $$\mathit{Tactics}(s) = \textbf{select } L \textbf{ from } s \textbf{ where somewhere inside } L \, *$$

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

  $$\begin{aligned} \mathit{Input}(\mathsf{tac}, s) = {} & \textbf{select } G \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere inside } \mathsf{tac} \textbf{ ingoals } G \\ \mathit{Output}(\mathsf{tac}, s) = {} & \textbf{select } G \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere inside } \mathsf{tac} \textbf{ outgoals } G \end{aligned}$$

- Which tactics call themselves recursively? (shown earlier for fixed tac)

  $$\begin{aligned} \mathit{Rec}(s) = {} & \textbf{select } L \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere inside } L \textbf{ somewhere islabel } L \end{aligned}$$

- Which tactic uses atomic tactic a, i.e., inside which label does a occur? Using the **nearby** combinator defined in the last section, this query returns all labels L which contain a directly, i.e., labels which are the immediate surrounding parent of a, not a more distant ancestor.

  $$\begin{aligned} \mathit{Inside}(a, s) = {} & \textbf{select } L \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere inside } L \textbf{ nearby atomic } a \end{aligned}$$

- Are there steps in the proof which have no effect?

  $$\begin{aligned} \mathit{UselessTacs}(s) = {} & \textbf{select } L \textbf{ from } s \textbf{ where} \\ & \quad \textbf{somewhere inside } L \textbf{ ingoals } G \wedge \textbf{outgoals } G \end{aligned}$$

  This returns useless tactics that return the same goal that they were given (necessarily G is a single element list by the hiproof structure). Some tactics may be even worse and return the same goal that they were given and more besides! To catch those, we could add subset inclusion to goal matching.

- Are there duplicated subproofs inside a proof? We answer this by finding labelled subtrees that have the same input and output goals, using the **separately** operator introduced earlier:

$$\begin{aligned}\mathit{Duplicates}(s) = \textbf{select } & L_1, L_2, G_i, G_o \textbf{ from } s \textbf{ where} \\ & \textbf{separately inside } L_1\ q \textbf{ and inside } L_2\ q\end{aligned}$$

  where q abbreviates **ingoals** $G_i \wedge$ **outgoals** G_o.

In the last example, we might want to return (or replace) the actual duplicate subtrees. To do that we would need to add variables ranging over hiproofs (or paths in hiproofs) to the language; see Sect. 6 for remarks on this extension.

4.2 Query equivalence and decidability

Prop. 2 characterises proofs by queries. We can turn this around, and ask whether queries can be characterised by the proofs that satisfy them. This motivates a Leibniz-style equality between queries.

Definition 2. *Two queries p, q are* equivalent*, written $p \cong q$, if for all proofs s and substitutions σ, we have $s \models_\sigma q \Longleftrightarrow s \models_\sigma p$.*

We can now state a number of equations over queries. These are proven by expanding Def. 2 and using Def. 1. First, conjunction and disjunction commute over the basic queries; we write this as a family of equations:

$$\textbf{inside } nm\ (p \between q) \cong (\textbf{inside } nm\ p) \between (\textbf{inside } nm\ pq) \tag{2}$$

$$(p_1 \between p_2) \oplus q \cong (p_1 \oplus q) \between (p_2 \oplus q) \tag{3}$$

$$p \oplus (q_1 \between q_2) \cong (p \oplus q_1) \between (p \oplus q_2) \tag{4}$$

for $\between \in \{\wedge, \vee\}$ and $\oplus \in \{\textbf{then}, \textbf{beside}\}$. Negation distributes over the basic queries variously. E.g., the query **ingoals** gm is not satisfied by s iff the goals of s do not match gm, whereas the query **atomic** am is not satisfied by s iff either s is an atom that does not match am, or if it is not an atom. We give three equations, and omit similar ones for **outgoals**, **inside**, **then**, and **nothing**:

$$\neg(\textbf{ingoals } gm) \cong \textbf{ingoals } (\neg gm) \tag{5}$$

$$\begin{aligned}\neg(\textbf{atomic } am) \cong\ & \textbf{atomic } (\neg am) \vee (\textbf{islabel } *) \\ & \vee \textbf{nothing} \vee \textbf{isbeside} \vee \textbf{isthen}\end{aligned} \tag{6}$$

$$\begin{aligned}\neg(p \textbf{ beside } q) \cong\ & ((\neg p) \textbf{ beside } *) \vee (* \textbf{ beside } (\neg q)) \\ & \vee (\textbf{atomic } *) \vee \textbf{nothing} \vee (\textbf{islabel } *) \vee \textbf{isthen}\end{aligned} \tag{7}$$

Finally, we have the usual laws of propositional logic: De Morgan equalities, double negation, commutativity and distributivity of conjunction and disjunction. By reading our equations as rewrite rules from left to right, we get a decision procedure for equivalence of queries, as long as they do not contain any recursion.

Definition 3 (DNF). *A query q is in disjunctive normal form (DNF), if it is of the shape $\bigvee_{i=1\dots n} \bigwedge_{j=1\dots m_i} \phi_{i,j}$ where $\phi_{i,j}$ are basic queries, or in other words a disjunction of conjunctions of basic queries.*

Proposition 3. *For each recursion-free query q there is an equivalent query q' in DNF, denoted as $DNF(q)$.*

The *size* of *DNF*(q) is exponential in the size of q. Most equations are linear in the query argument (that is, the query arguments occur once on each side of the equation), and hence only introduce a constant size increase when applied left to right, but (3) and (4) and similarly distributivity for $\vee$ and $\wedge$ contain the query argument q and p twice on the right-hand side. Thus, each of **then**, **besides** or $\wedge$ may double the size, leading to exponential increase. Of course, the size of the resulting $DNF(q)$ will usually be much smaller; we can cut it further down by eliminating contradictory conjunctions such as **atomic** $a \wedge$ **isthen**.

Checking that a basic query q satisfies a given hiproof s is linear in the size of q, as we just traverse the structure of q and s. Hence, checking that a query q' in DNF satisfies a given hiproof s is also linear in the size of q', as we merely need to check each of the basic queries $\phi_{i,j}$ against s. Hence, because of the size of $DNF(q)$, satisfiability of recursion-free queries is decidable in exponential time.

Proposition 3 does not hold for queries containing the recursion operator. To check that a given recursive query q and substitution σ satisfy a hiproof s, we can unfold the recursion in q as often as needed, and then use the DNF of the unfolded term. The size of the hiproof s bounds the size of the unfolding, as a hiproof cannot be smaller than a basic query it satisfies, and $DNF(q)$ is always larger than q. Thus:

Proposition 4. *$s \models_\sigma q$ is decidable in exponential time over $size(s) + size(q)$.*

This straightforward argument establishes decidability. Better complexity bounds surely exist, as they are known for related query languages and various fragments (see e.g., [5]), but mappings into other languages are beyond our scope here.

5 Implementing queries

We have built a simple implementation of the query language in order to validate its design by running example queries on small proofs and checking the results. We directly use the semantics and turn Def. 1 into a function $sat(s, q)$ which implements the **select** statement from Sect. 4.1 and returns the (minimal) set of all substitutions which satisfy q.

Substitutions are given as partial functions $Var \rightharpoonup \mathcal{T}$, where $\mathcal{T}$ is the set of names or goal lists. Given two substitutions ρ and σ, their unification $unify(\rho, \sigma)$ is defined iff $\forall a \in dom\,\rho \cup dom\,\sigma. \rho(a) = \sigma(a)$, and it is defined pointwise to be $\sigma(a)$ if $\sigma(a)$ is defined, $\rho(a)$ if $\rho(a)$ is defined, and undefined everywhere else. To combine two sets Φ and Ψ of substitutions, as returned by recursive calls of the *sat* function, we define the combinator

$$\Psi \gg \Phi = \{unify(\rho, \sigma) \mid \rho \in \Psi, \sigma \in \Phi, unify(\rho, \sigma) \text{ is defined}\}$$

For the basic queries, there are simple functions sat_N and sat_G which return the set of substitutions matching a given name or goal match. Then sat can be recursively defined as follows (we only give some of the representative cases):

$$\begin{aligned}
sat(a, \textbf{atomic}\ nm) &= sat_N(a, nm)\\
sat([l]\, s, \textbf{inside}\ nm\ q) &= sat_N(l, nm) \gg sat(s, q)\\
sat(s_1\ ;\ s_2, q_1\ \textbf{then}\ q_2) &= sat(s_1, q_1) \gg sat(s_2, q_2)\\
sat(s, q_1 \wedge q_2) &= sat(s, q_1) \gg sat(s, q_2)\\
sat(s, q_1 \vee q_2) &= sat(s, q_1) \cup sat(s, q_2)
\end{aligned}$$

Note that when combining the results for a disjunctive query, we can just take the union of the results. We can show the correctness of this definition, namely that if $\sigma \in sat(s, q)$ then $s \models_\sigma q$, and also that if $s \models_\sigma q$ then there is $\rho \in sat(s, q)$ such that $\rho \subseteq \sigma$ (so sat returns a minimal set of substitutions).

Implementation. Using this definition, our prototype implements the query language for small experiments. It represents queries as an algebraic datatype `Q`, and in time-honoured fashion uses SML as both implementation platform and scriptable command-line interface. Hiproofs are represented modulo the equations in Sect. 4, following the denotational semantics in [1]. The implementation is a functor which is generic over the proofs in question, reflecting the generic nature of the query language.

We provide two instantiations of the generic implementation: one for the syntactic hiproofs, where we have a datatype `S` as in (1), and one which models Isabelle proof objects [6] as hiproofs. Taking existing proofs such as those in Isabelle as hiproofs, we need to derive the hierarchical structure. We use theorems to do this. That is, a box $[l]\, s$ is a theorem named l, together with its proof s. This leads to an interesting example: the query $Rec(s)$ applied to an Isabelle hiproof would return all theorems which are used in their own proof.

6 Related work and conclusions

This paper introduced *locally structured* proof queries in our proof query language, PrQL. These build up patterns of structure that are matched to a position in the implicit tree. Using logical connectives, variable substitution and structural recursion, queries can span and relate different portions of the tree and express many natural queries on proofs. But, in this locally structured fragment it is not possible to write a query that directly refers to (or returns) a position in the tree, or does any counting. This limitation can be lifted, e.g., by adding a notion of path to the language. In future work we will report on *globally structured* queries allowed by this, as well as a slightly different language where queries are defined directly over our semantic models.

Related work in theorem proving. The idea of a general query language for inspecting formal proofs appears novel, although there are many investigations into exploiting proofs in particular ad hoc ways. We can't survey all but mention a few. Researchers have connected decision procedures to theorem proving by grafting invocation records of decision procedures (with possible justifications) into an overall proof (e.g., [7]). Noteworthy sub-trees may be represented using names for reference (and then shared to create a dag structure) as in TPTP and its proof format TSTP [8]. Many systems use debugging output for proof procedures to create a lengthy log, which explains where things were tried and failed. Some tools use representations of proof trees in the first place which connect the proof-producing mechanism to the proof and are equipped with browsing and editing mechanisms, e.g., NuPrl [9]. Besides checking proofs [10], other researchers have made efforts to translate proofs between systems [11]; discover dependencies between parts of proofs [12] to help simplify or rearrange; and data-mine proofs to discover common patterns [13].

To exploit a formal generic proof representation language like hiproofs, it is appealing to use a generic concrete representation like TSTP. A TSTP proof consists of the sequence of formulas output by an automated theorem prover along with their sources, and is hence a more "operational" format than hiproofs, which can be translated into TSTP in either forwards-style (deriving conclusions from axioms) or backwards-style (decomposing conjectures to back to axioms). Going in the opposite direction, although TSTP does not represent tactics, inference rules can be nested, giving a simple form of hierarchy. We could decompose the derivations in various ways thus deriving an implicit hierarchy, or extend the language with labels on sub-derivations to represent hierarchy explicitly. Proofs in the TSTP archive can be queried online [14] using a range of primitive and quantitative predicates, or by translation [15] into the Proof Markup Language (PML) [16], which serves as an interlingua representation for the justification of results produced by Semantic Web services. Queries in PML are simply partial proofs, rather than expressions in a separate query language (of course, PrQL also has close ties to its underlying proof language), and query evaluation seeks to return (possibly partial) proofs that "fill in the blanks" in the initial query. Our original motivation for developing a query language was to extract information from TSTP proofs in order to construct safety cases, and we plan to extend our prototype implementation to support this.

Query languages for structured data and programs. Away from theorem proving, query languages for trees and graphs have been studied for some time. Languages related to PrQL include those aimed at semi-structured (XML-like) models such as UnQL [5] which uses structural recursion on tree (and graph) representations, similarly to PrQL's recursive queries, and Graph Logic [17] which uses a separating conjunction to destruct the graph subject of queries. Checking for patterns in programs, ASTLog [18] is a Prolog variant for examining syntax trees and PQL [19] is a more general framework for querying programs at varying levels of abstraction. Establishing precise connections with PrQL would let us exploit known complexity results, existing algorithms and their implementations.

Acknowledgements. We would like to thank Geoff Sutcliffe for help with TPTP.

References

1. Denney, E., Power, J., Tourlas, K.: Hiproofs: A hierarchical notion of proof tree. Electr. Notes Theor. Comput. Sci. **155** (2006) 341–359
2. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. Mathematics in Computer Science **3**(3) (2010) 309–330
3. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40** (February 2008)
4. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and semantic web query languages: A survey. In Eisinger, N., Maluszynski, J., eds.: Reasoning Web. LNCS 3564., Springer (2005) 35– 133
5. Buneman, P., Fernandez, M., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. The VLDB Journal **9**(1) (March 2000) 76–110
6. Berghofer, S., Nipkow, T.: Proof terms for simply typed higher order logic. In Harrison, J., Aagaard, M., eds.: TPHOLs'00, LNCS 1869. Springer (2000) 38–52
7. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. Journal of Automated Reasoning **21** (1998) 279–294
8. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP language for writing derivations and finite interpretations. In Furbach, U., Shankar, N., eds.: Automated Reasoning. LNCS 4130. Springer (2006) 67–81
9. Allen, S.F., Bickford, M., Constable, R.L., Eaton, R., Kreitz, C., Lorigo, L., Moran, E.: Innovations in computational type theory using Nuprl. Journal of Applied Logic **4**(4) (December 2006) 428–469
10. Necula, G., Lee, P.: Proof generation in the Touchstone theorem prover. In McAllester, D., ed.: CADE-17. LNCS 1831. Springer (2000) 25–44
11. Denney, E.: A prototype proof translator from HOL to Coq. In Aagaard, M., Harrison, J., eds.: TPHOLs 00. LNCS 1869., Springer (2000) 108–125
12. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: Proc. User Interfaces for Theorem Provers, UITP'98. (1998)
13. Urban, J.: MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. J. Applied Logic **4**(4) (2006) 414–427
14. Sutcliffe, G., Suttner, C.: The TPTP problem library for automated theorem proving. `http://www.tptp.org` Homepage and online tools, visited November 2011.
15. da Silva, P.P., Sutcliffe, G., Chang, C., Ding, L., Rio, N.D., McGuinness, D.L.: Presenting TSTP proofs with inference web tools. In Konev, B., Schmidt, R.A., Schulz, S., eds.: PAAR/ESHOL. of CEUR Workshop Proceedings 373, CEUR-WS.org (2008)
16. da Silva, P.P., McGuinness, D., Fikes, R.: A proof markup language for semantic web services. Information Systems **31**(4-5) (June 2006) 381–395
17. Cardelli, L., Gardner, P., Ghelli, G.: A spatial logic for querying graphs. In Widmayer, P., Ruiz, F.T., Bueno, R.M., Hennessy, M., Eidenbenz, S., Conejo, R., eds.: ICALP. LNCS 2380., Springer (2002) 597–610
18. Crew, R.F.: ASTLOG: A language for examining abstract syntax trees. In: DSL, USENIX (1997)
19. Jarzabek, S.: Design of flexible static program analyzers with PQL. IEEE Trans. Software Eng. **24**(3) (1998) 197–215

*This work is dedicated fondly to the memory of Kostas Tourlas,
one of the originators of hiproofs.*