# A Semantic Basis for Proof Queries and Transformations

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 & Universität Bremen
Bremen, Germany

**Abstract.** We add *updates* to the query language *PrQL*, designed for inspecting machine representations of proofs. PrQL natively supports *hiproofs* that express proof structure using hierarchically nested labelled trees, which we claim is a natural way of taming the complexity of huge proofs. Query-driven updates allow us to change this structure, in particular, to transform proofs produced by interactive theorem provers into forms that are easier for humans to understand, or that could be consumed by other tools. In this paper we motivate and define basic update operations, using an abstract denotational semantics of hiproofs and queries. This extends our previous semantics for queries based on syntactic tree representations. We define update operations that add and remove sub-proofs or manipulate the hierarchy to group and ungroup nodes. We show that these basic operations are well-behaved and hence can form a sound core for a hierarchical transformation language. Our study here is firmly in language design and semantics; implementation strategies and study of sub-languages of our query language with good complexity will come later.

## 1 Introduction

We are interested in ways to exploit machine representations of proofs constructed by interactive or automated theorem provers. These proof representations are produced so that they can be independently checked or imported into other systems. We believe that they can be exploited beyond this. For example, system inputs such as proof scripts are rarely given at the lowest level of detail, even with interactive theorem provers. Therefore it can be useful for proof developers to understand how the system has found a proof: which inference rules have been used, which axioms, which instantiations for existential variables, and

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

so on. More complex questions are also interesting. For example, whether a proof contains unnecessary detours or replicated sub-proofs.

To this end, we recently introduced PrQL [2], a *proof query language* which treats a large formal proof as an object that can be examined in a systematic way. We are currently developing practical prototypes to experiment with proof queries, so far based on exporting from Isabelle [2] and HOL Light [21]. But it is clear already that as well asking questions, we also want to be able to *transform proofs* to alter their structure in various ways. This may be used to aid understanding (human or machine), by hiding certain kinds of details. Or it could be used for optimisation or adaptation, to change proofs to more efficient forms, or for consumption by different systems such as proof commentary tools or machine learning tools. This paper is a study of a rigorous foundation for such transformations, introducing *update* extensions for PrQL.

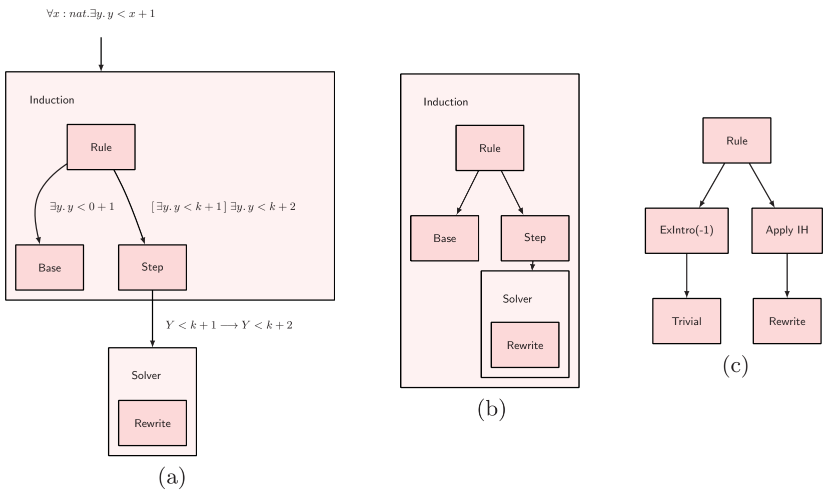
To study the foundations of updates, we need to have the right data model for hiproofs and define operations that preserve the hiproof structure. Some transformations may also preserve theoremhood of proved statements. This is why we design our own query and transformation language, rather than immediately encoding our concepts into a more general graph or tree model (such as XML) with an existing query and transformation language (such as XQuery Update [10] or XDuce [20]) that could make arbitrary dissections and rearrangement.

When it comes to implementing our query and update language, it is obviously desirable to reuse existing systems which have looser semantics but optimised implementations for query language fragments in good complexity classes. We may consider for example, graph databases, other tools in the "NoSQL" family or perhaps even SPARQL. We are conducting some early experiments in parallel with the work described here.

*Contributions and paper outline.* This paper contributes towards generic foundational aspects of theorem proving systems, in particular, the novel aspects of querying and transforming the proof objects which can be recorded by proof tools. Moreover, we contribute to the study of a *structured* representation for these objects. Sect. 2 introduces the idea of proof transformations that we are studying, with some informal examples and motivations. Sect. 3 recaps the technical background of hiproofs and PrQL. Sect. 4 introduces a revised denotational semantics for hiproofs; this extends previous work, connecting the syntactic strand of [3] with the previous denotational semantics of [14]. The new extensions add explicit orderings among subtrees and the ability to model *open*, i.e., incomplete, proofs. Sect. 5 gives a new denotational semantics to our query language. This interpretation provides two advances: (1) the ability to return locations in the hiproof where a query is satisfied, and (2) a close connection to a graph model that we can use to encode hiproofs. Sect. 6 builds on top of this to define our four kinds of update operations. We show that these operations are well-behaved and preserve proofs in certain senses. Finally, we give a more detailed comparison to related work in the concluding Sect. 7.

## 2 Querying and transforming hierarchical trees

We start from *hiproofs* [3, 14], which provide an abstract, generic notion of proof tree with hierarchical structure. Hiproofs are composed from atomic rules of inference from an unspecified underlying logic, but additionally provide a notion of *hierarchy*, by allowing labelling and nesting of subtrees inside boxes. This succinct notion of *structuring* in a proof can be used, for example, for noting where a lemma was applied, or where a particular tactic or external proof tool produced a subtree. The hierarchical structure of hiproofs and its interaction with the proof-tree is more complex than the straightforward tree structure, in particular because hiproofs allow nesting of partially completed proofs.



(a)

(b)

(c)

**Fig. 1.** Different hiproof structures on the same underlying proof

The picture shown in Fig. 1(a) is an example hiproof, shown at a certain level of abstraction. It corresponds with an ordinary (but upside-down) natural deduction style proof tree: the theorem being proved, $\forall x.\exists y.y < x + 1$ is shown at the top, and then the proof outline shows how the proof is achieved by decomposing the *goal* theorem into pieces. The labelled boxes correspond to *tactics* which have been applied to do this. Notice how the Induction box encapsulates an incomplete proof; it has the dangling edge which is passed into the Solver box. We suppose that boxes such as Base may contain further details, perhaps right down to atomic inferences in the underlying system; the diagram only hints at the full hiproof. Fig. 1(a) shows the statements being proved along edges. In a visualisation tool (such as the web-based HipCam [21]) the goals may be shown in pop-ups so as not to clutter the display, and boxes such as Base can be opened and closed dynamically.

*Variations of hierarchy.* Further right in Fig. 1 we see some alternative structuring of this simple inductive proof. Fig. 1(b) shows the complete step case being enclosed by the induction box; whereas Fig. 1(c) shows just the induction rule itself being boxed. These pictures motivate our main kind of desirable transformations: to alter and introduce hierarchical structure. For example, when an inductive proof appears in the proof tree, we might like to give it the uniform structure on the left so it can be easily picked apart. However, the proofs which arise by a naive labelling of tactics in HOL Light without hiproof adaptation [21], for example, have the form in Fig. 1(c).

*Basic transformations.* Generally, the life cycle of data management is captured by functions to *create*, *read*, *update*, and *delete*. We already have mechanisms to create proof objects: abstractly, via the syntax for hiproofs reviewed in Sect. 3, and in practice by functions for exporting proof objects from systems like Isabelle [2] and HOL Light [21]. To inspect proof objects, PrQL provides a language of structured queries, reviewed further below. To manipulate existing hiproofs, we need to add update and delete operations. But we want to do this in a way that respects the proof structure, rather than as arbitrary edits to a tree or graph. This motivates the following four types of operation.

**Introduce hierarchy** is used moving from Fig. 1(c) to Fig. 1(a): we introduce a nested hiproof called Base for the two steps ExIntro and Trivial, which hides the detail. We also push in the children of Rule into the Induction box.

**Remove hierarchy** is the opposite transformation. Visualisation tools perform this reversibly under user control, but here we want to permanently transform the underlying structure by pulling out individual pieces, such as when moving from Fig 1(b) to Fig 1(a).

**Remove subproof** deletes part of a hiproof. This is a radical operation, and will change what is being proved, popping out an unproved subgoal to the top level. For example, if we remove the Solver tactic in Fig. 1(a), the proof is left unfinished with the subgoal $Y < k + 1 \Longrightarrow Y < k + 2$ remaining.

**Complete subproof** is the inverse operation, and grafts on a new subtree. This can resolve a previously unproved subgoal, or generate new subgoals.

### 2.1 Finding somewhere to transform

First, to apply a transformation, we need to know *where* in a target hiproof it should be applied. A natural way to find a transformation point is to search for a node satisfying some properties: this is where *queries* enter the picture. (Similarly, update languages that have been defined elsewhere for semistructured data and graphs also use queries to position updates; see Sect. 7.)

We have already designed PrQL, a query language for hiproofs, so it is natural to reuse it. PrQL is a structured query language which combines property queries (that look at local properties on nodes) with structuring operations (that combine queries across connected nodes, decomposing the tree). These can be defined with recursion and logical connectives, giving a powerful language that can encode search in queries. For example, the PrQL query

$$\textbf{somewhere}\ (\textbf{atomic}\ \textsf{ExIntro}\ \textbf{then}\ \textbf{atomic}\ \textsf{Trivial})$$

is satisfied by the hiproof in Fig. 1(c). The **atomic** operator examines a label on a bottom-most nested node. The **then** operator decomposes the target graph across the proof tree sequence. Similarly, we can decompose sibling hiproofs with **beside** and nested hiproofs with **inside**, building up patterns. Patterns may contain *match variables* that get instantiated with names of rules or box labels. Using recursion we can define operators like **somewhere** (finds a match in any subtree) and **nearby** (finds a match in any subtree at the same nesting depth). See Sect. 3.1 for more details of PrQL.

However, so far there is not yet a notion of *where* a query is satisfied; we do not have a way to describe where ExIntro or Trivial rules were actually found. To pick out specific nodes in a hiproof, we extend the query language to return positions: a new type of match variable standing for a (sub)hiproof where a query is satisfied. We add the new query term "**at** $X$" which matches $X$ against the "currently examined" node in the tree. So

$$\textbf{somewhere}\ \textbf{inside}\ \textsf{Induction}\ \textbf{nearby}\ (\textbf{at}\ X\ \wedge\ \textbf{atomic}\ \textsf{Trivial})$$

returns locations $X$ where Trivial appears immediately inside an Induction box.

Unlike labels for boxes and atomic rule names, nodes in our proof trees are abstract: we do not need user-level syntax for writing their identities. So **at** can only locate a position by properties; it cannot pick out a specific node concretely. But the query language is precise enough that, for any specific node in the tree, there is a query which picks out that node uniquely (see Prop. 1 in Sect. 5).

### 2.2 Updating proofs

Now we have a way to specify transformation points, we can show how our update operations are written. Several language design choices are possible. We have followed an SQL-like paradigm, matching positions then using one-shot operations which can update a large proof in-place, based on the selected positions. A more ambitious choice would be to design a hybrid query and update language, with looping and branching to build up complex transformations. But we first want to understand the update combinators that are common to both.

As a first example, to turn Fig. 1(c) into Fig. 1(b) we use a transformation which adds a box around a given subtree, called **box**:

$$\begin{aligned}&\textbf{box}\ X\ \textbf{to}\ Y\ Z\ \textbf{as}\ \textsf{Induction}\ \textbf{where}\\&\quad(\textbf{at}\ X \wedge \textbf{atomic}\ \textsf{Rule})\ \textbf{then}\ (\textbf{seq}\ Y\ \textbf{beside}\ \textbf{seq}\ Z)\end{aligned} \tag{1}$$

where the recursive query **seq** $X$ picks out a sequence ending at $X$:

$$\textbf{seq}\ X \overset{def}{=}\ \mu Q.\ *\ \textbf{then}\ Q \vee (\textbf{at}\ X \wedge \neg(*\ \textbf{then}\ *))$$

Besides adding boxes, we can remove them with **unbox**:

$$\textbf{unbox}\ X\ \textbf{where at}\ X \land \textbf{inside}\ \textsf{Solver}$$

which removes the Solver box around the result of an automatic tactic. Instead we could rename it, simply writing: **rename** Solver **as** Auto.

So far, these operations have not changed what is proved in the hiproof. Other updates change the underlying proof tree, but maintain its validity. For example, maybe we are not interested in a particular subtree of a proof

$$\textbf{deletetree}\ X\ \textbf{where inside}\ \textsf{Meson}\ \textbf{at}\ X$$

then this removes the subtree generated by an automatic procedure, just leaving the name of the procedure. In the hiproof structure, we do not forget that something is unproved; the subtree leaves a *dangling* edge.

Dually, we can fill in a proof for such a dangling edge; this is a *refinement* operation in the sense that it extends the proof:

$$\textbf{refine}\ X\ \textbf{with}\ s\ \textbf{where at}\ X \land \textbf{unproved}\ \gamma$$

Here, $s$ is a literal term in the syntax for hiproofs, which proves the goal $\gamma$.

Finally, it can be useful to use a more general replacement transformation which is defined using **deletetree** then **refine**. For example, to find useless detours in a proof tree, we use the query:

$$\textbf{useless}\ X\ Y\ \overset{def}{=} (\textbf{at}\ X \land \textbf{goal}\ G)\ \textbf{then nearby}\ (\textbf{at}\ Y \land \textbf{goal}\ G)$$

this identifies a path from $X$ to $Y$ where we hit the same goal $G = \gamma$. It might even be a tactic which is worse than useless, in that it has transformed a goal $\gamma$ into several more goals to prove including $\gamma$ again. Now the **replace** update

$$\textbf{replace}\ X\ \textbf{by}\ Y\ \textbf{where}\ (\textbf{useless}\ X\ Y)$$

removes this detour.

## 3 Syntactic Hiproofs and PrQL queries

This section introduces previous material as background. We are as concise as possible and refer the reader to previous papers for more details [2, 3, 14].

Hiproofs add structure to an underlying *derivation system*, a simple kind of logical framework. A derivation system is given by a set $\mathcal{G}$ of *goals* (intuitively: possibly provable sequents or judgements), ranged over by $\gamma$, and a set of atomic inference rules ranged over by $a$. Atomic rules are composed to give hiproofs, which have a functional reading: a hiproof maps a finite list of input goals $g_1 = [\gamma_1, \ldots, \gamma_n]$ to a list of output subgoals $g_2 = [\gamma'_1, \ldots, \gamma'_m]$.

Informally, we draw hiproofs as inverted trees with a nested structure. Formally, a hiproof is given by two forests on the same set of nodes, as explained in Sect. 4. Syntactically, a hiproof can be written as a term:

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

where $l \in \mathcal{L}$, an arbitrary set of names and $a \in \mathcal{A}$ for some special subset $\mathcal{A} \subset \mathcal{L}$. We think of labels as standing for names of tactics or proof rules, or atomic steps; they have no semantic content. For example, the proofs in Fig. 1 are written syntactically as

$$([\mathsf{Induction}]\,\mathsf{Rule} \ ; \mathsf{Base} \otimes \mathsf{Step}) \ ; [\mathsf{Solver}]\,\mathsf{Rewrite} \tag{2}$$

$$[\mathsf{Induction}]\,\mathsf{Rule} \ ; \mathsf{Base} \otimes (\mathsf{Step} \ ; [\mathsf{Solver}]\,\mathsf{Rewrite}) \tag{3}$$

$$\mathsf{Rule} \ ; (\mathsf{ExIntro} \ ; \mathsf{Trivial}) \otimes (\mathsf{ApplyIH} \ ; \mathsf{Rewrite}) \tag{4}$$

### 3.1 Structured queries in PrQL

The definition of PrQL starts with *matches* built from 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:

$$nm ::= a \mid l \mid \bullet \mid \xi \mid N \mid \neg nm \qquad\qquad gm ::= \gamma \mid \psi \mid G \mid \neg gm$$

where $\xi$ stands for a logic-dependent predicate on names, and $\psi$ 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 $\gamma$ is in the form of a horn clause, when $\phi_{horn}(\gamma)$ holds. The special name $\bullet$ is used to label unproved goals; the name $* = \neg\bullet$ serves as a wildcard.

We use matches to build up queries, $q$, as below. The extension to PrQL to locate vertices uses a set of match variables $Var_H$, ranged over by $X$.

$$\begin{array}{lll} q ::= * & & \text{anything non-empty} \\ \quad \mid & \mathbf{at}\ X & \text{matches at node } X \\ \quad \mid & \mathbf{atomic}\ nm & \text{atomic rule match} \\ \quad \mid & \mathbf{inside}\ nm\ q & q \text{ satisfied inside box with label matching } nm \\ \quad \mid & q_1\ \mathbf{then}\ q_2 & q_1 \text{ and } q_2 \text{ satisfied by successive nodes} \\ \quad \mid & q_1\ \mathbf{beside}\ q_2 & q_1 \text{ and } q_2 \text{ satisfied by adjacent nodes} \\ \quad \mid & \mathbf{goal}\ gm & \text{proved goal matches } gm \\ \quad \mid & q_1 \wedge q_2 \mid q_1 \vee q_2 \mid \neg q & \text{compound queries} \\ \quad \mid & \mu Q.q & \text{recursive query} \end{array}$$

Queries are built from schematic hiproof terms. They are posed against an implicit hiproof subject, instantiating the match variables and testing goals. Compound queries are built using logical connectives and recursion. This core language allows many useful derived forms, like the search operator **somewhere**. We can examine gaps in proofs too; to assert that the hiproof has $\gamma$ as an unsolved goal we write:

$$\textbf{unproved}\ \gamma \overset{def}{=} \textbf{goal}\ \gamma \wedge \textbf{atomic}\ \bullet$$

This works because we model 'dangling' edges as empty boxes labelled with $\bullet$.

## 4 Denotational hiproofs

A hiproof consists of two forests on the same set of nodes, with a distinguished root, satisfying some conditions [14]. To relate to a derivation system (where premises of inference rules have an ordering), we add a left-to-right ordering among siblings. To relate to the syntax, we use a more general forest notion first, then restrict to hiproofs. To model incomplete (partial) proofs, we add nodes corresponding to unproved goals. Lastly, we extend labelling to attach to each node the goal it validates, as shown on edges entering nodes in Fig. 1(a).

Given a forest $F$ defined by a relation $R$ on a set of vertices, we write $\mathit{siblings}_R(v, v')$ if $v$ and $v'$ are children of the same $R$-parent. Given a vertex $v$, we write $\mathit{isroot}_R(v)$ for the assertion that $v$ is a root wrt $R$, i.e., $\forall v'. v'\ R\ v \Longrightarrow v = v'$, and $\mathit{isleaf}_R(v)$ for the dual, i.e., $\forall v'. v\ R\ v' \Longrightarrow v = v'$.

**Definition 1 (Ordered Hiforest).** *An* ordered hiforest $H = \langle V, L, \leq_{\mathsf{i}}, \to_{\mathrm{s}}, \lesssim\rangle$ *consists of a finite set of vertices $V$ with a labelling function $L : V \to (\mathcal{L} \cup \{\bullet\}) \times \mathcal{G}$ and three relations on $V \times V$. The relations are an* inclusion *order $\leq_{\mathsf{i}}$ (which captures the nesting of vertices; $>_i$ is proper containment), a* sequencing *relation $\to_{\mathrm{s}}$ (which captures the functional composition of nodes) and a* child *order $\lesssim$. These are subject to the following conditions:*

0. *$\langle V, \leq_{\mathsf{i}}\rangle$ and $\langle V, \to_{\mathrm{s}}\rangle$ each form forests; $\leq_{\mathsf{i}}$ and $\lesssim$ are partial orders.*
1. *arrows target outer nodes: $v \to_{\mathrm{s}} w$ and $v' >_i w \Longrightarrow v' >_i v$.*
2. *arrows emanate from inner nodes: $v \to_{\mathrm{s}} w$ and $v' \leq_{\mathsf{i}} v \Longrightarrow v = v'$.*
3. *inclusion & sequence are mutually exclusive: $v \leq_{\mathsf{i}} w$ and $v {\to_{\mathrm{s}}}^* w \Longrightarrow v = w$.*
4. *boxes have unique roots:*
   *$\mathit{siblings}_{\leq_{\mathsf{i}}}(v, v') \wedge \mathit{isroot}_{\to_{\mathrm{s}}}(v) \wedge \mathit{isroot}_{\to_{\mathrm{s}}}(v') \Longrightarrow v = v'$.*
5. *children or top-level roots are totally ordered:*
   *$\mathit{siblings}_{\to_{\mathrm{s}}}(v, v') \vee (\mathit{isroot}_{>_i}(v) \wedge \mathit{isroot}_{>_i}(v')) \Longrightarrow v \lesssim v' \vee v' \lesssim v$.*
6. *only leaves (wrt. sequencing and inclusion) may have $\bullet$ label:*
   *$L(v) = (\bullet, \gamma) \Longrightarrow \mathit{leaf}_{\to_{\mathrm{s}} \cup >_i}(v)$.*

Each node in a hiforest is given a name and a goal. The goal is the theorem proved at that node. The unproved parts are the 'dangling' holes labelled by $\bullet$. An ordered hiforest proves a sequence of top-level goals, whereas a hiproof proves just one.

**Definition 2 (Ordered Hiproof).** *An* ordered hiproof *is an ordered hiforest which satisfies the additional constraint:*

7. *Top-level roots are unique:* $\mathit{isroot}_{\rightarrow_{\mathrm{s}} \cup >_i}(v) \wedge \mathit{isroot}_{\rightarrow_{\mathrm{s}} \cup >_i}(v') \implies v = v'$.

We are mainly interested in *valid* hiproofs, which are those corresponding to a proof in the underlying derivation system.

**Definition 3 (Validity).** *A hiforest $H$ is* valid *if it corresponds to a sequence of (possibly incomplete) proof trees in the underlying derivation system; we write $H \models g_1 \longrightarrow g_2$ if this holds and where $g_1$ is the list of goals on the outermost roots of $H$, and $g_2$ is the list of unproved goals on the holes, as ordered by extending $\lesssim$ to the leaves of the tree.*

A *map* between two hiforests is a map between the vertices and the labels which preserves the orderings and the labelling. We say a hiforest $H_1$ *refines to* a hiforest $H_2$, $H_1 \sqsubseteq H_2$, if there is an inclusion from $H_1$ to $H_2$ which also preserves the roots wrt $>_i$.

We now define some operations on the two dimensions of hiforests which will form the semantic foundations of our transformations. For brevity, definitions are given informally here, and made precise in the appendix. Given two hiforests $H_1$ and $H_2$ such that $H_1 \models g_1 \longrightarrow g$ and $H_2 \models g \longrightarrow g_2$, we define a composition operation $\mathit{graft}(H_1, H_2)$ that 'grafts' the roots of $H_2$ into the dangling goals of $H_1$, such that $\mathit{graft}(H_1, H_2) \models g_1 \longrightarrow g_2$; it can be characterised at the smallest hiforest $H_3$ which refines $H_1$, $H_1 \sqsubseteq H_3$, for which there is a (necessarily injective) map $\alpha : H_2 \longrightarrow H_3$. This is an instance of a more general operation $\mathit{graft}(H_1, H_2, v_1, \ldots, v_m)$ which grafts the $m$ roots of $H_2$ into the specified danglers $v_1, \ldots, v_m$ of $H_1$, where $H_1$ may contain more than $m$ danglers.

Given a vertex $v \in V$ in hiforest $H$, we define $\mathit{cover}(v, H)$ as the hiproof containing the set of vertices in $H$ reachable from $v$ by $>_i$ or $\rightarrow_{\mathrm{s}}$, including $v$ itself. If $H \models g_1 \longrightarrow g_2$ then $\mathit{cover}(v, H) \models \gamma_v \longrightarrow g_v$ where $L(v) = (l, \gamma_v)$ and $g_2 = g_2' \mathbin{^\wedge} g_v \mathbin{^\wedge} g_2''$ (with $^\wedge$ denoting list concatenation). The operation $\mathit{chop}(v, H)$ removes exactly these vertices, replacing them with a hole. So $\mathit{chop}(v, H) \models g_1 \longrightarrow g_3$ where $g_3$ is the list $g_2' \mathbin{^\wedge} [\gamma_v] \mathbin{^\wedge} g_2''$. Together, these operations are inverse to grafting, i.e. $\mathit{graft}(\mathit{chop}(v, H), \mathit{cover}(v, H), v) = H$ (modulo some technical restrictions). The final operations are $\mathit{box}(l, H)$ and $\mathit{unbox}(H)$ which add and remove 'boxes' around the roots of $H$, where a box is a node (labelled $l$) including all the other nodes (below that root). These are inverse as well: $\mathit{unbox}(\mathit{box}(l, H)) = H$. These two operations preserve validity and input and output goal lists.

## 5 Semantics for queries

The query semantics we gave in [2] was based on querying syntax models directly. Since hiproofs are constructed syntactically, this is in a sense the most direct approach. However, syntactic representations are not canonical, because a particular underlying tree structure can be denoted by many terms in the syntax. E.g., the proof in Fig.1(c) can be expressed as in (4) or as

$$\mathsf{Rule}\ ;\ (\mathsf{ExIntro} \otimes \mathsf{ApplyIH})\ ;\ (\mathsf{Trivial} \otimes \mathsf{Rewrite})$$

For the definition of boolean satisfaction of a query given in [2], this is not problematic as we can close under the syntactic equivalence given by the algebraic structure of hiproofs. But to define updates it is more delicate, since we need a firm notion of *focus* in the hiproof to anchor changes; e.g., example (1) does not work with the syntactic form above. We could use normal forms for syntactic terms, but the denotational model is more direct and also fits well with parallel work on implementation using graph databases, building on [21].

The definition of query satisfaction in the denotational semantics uses a substitution to instantiate variables: $\sigma : (\mathit{Var}_N \rightharpoonup \mathcal{L}) \uplus (\mathit{Var}_G \rightharpoonup G) \uplus (\mathit{Var}_H \rightharpoonup V)$, where $V$ is the set of vertices of the hiproof being queried. The base case for query satisfaction is for names and goals, treated very similarly:

$$\begin{array}{rll@{\qquad\qquad}rll}
n \models_\sigma n' & \text{iff} & n = n' & \gamma \models_\sigma \gamma' & \text{iff} & \gamma = \gamma' \\
\xi \models_\sigma n & \text{iff} & \xi(n) & \psi \models_\sigma \gamma & \text{iff} & \psi(\gamma) \\
N \models_\sigma n & \text{iff} & \sigma(N) = n & G \models_\sigma \gamma & \text{iff} & \sigma(G) = \gamma \\
(\neg N) \models_\sigma n & \text{iff} & \neg(N \models_\sigma n) & (\neg G) \models_\sigma \gamma & \text{iff} & \neg(G \models_\sigma \gamma)
\end{array}$$

For a relation $R$ and distinct $a, b$, we write $a\ R^1\ b$ if $a\ R\ b$ and there is no intermediate $c$ such that $a\ R\ c$ and $c\ R\ b$.

**Definition 4 (Query satisfaction).** *Let $H$ be an ordered hiforest with vertices $V$ and $q$ a query. Satisfaction of $q$ for $H$ at a vertex $v \in V$ wrt a substitution $\sigma$ is defined as the least relation $v \models_\sigma q$ satisfying the following clauses:*

$$\begin{array}{lll}
v \models_\sigma * & \textit{always} & \\
v \models_\sigma \textbf{at}\ X & \textit{iff} & \sigma(X) = v \\
v \models_\sigma \textbf{goal}\ gm & \textit{iff} & gm \models_\sigma \gamma \textit{ where } L(v) = (l, \gamma) \textit{ for some } l \\
v \models_\sigma \textbf{inside}\ nm\ q & \textit{when} & nm \models_\sigma l \textit{ where } (v) = (l, \gamma) \textit{ for some } \gamma \\
 & & \qquad\qquad \textit{and } \forall w.\, w \leq_{\mathrm{i}}^1 v \Longrightarrow w \models_\sigma q \\
v \models_\sigma q_1\ \textbf{beside}\ q_2 & \textit{when} & v \models_\sigma q_1 \textit{ and } \exists w. v \lesssim^1 w \textit{ with } w \models_\sigma q_2 \\
v \models_\sigma q_1\ \textbf{then}\ q_2 & \textit{when} & v \models_\sigma q_1 \textit{ and } \exists w. v \to_{\mathrm{s}}^1 w \textit{ with } w \models_\sigma q_2 \\
v \models_\sigma q_1 \wedge q_2 & \textit{when} & v \models_\sigma q_1 \textit{ and } v \models_\sigma q_2 \\
v \models_\sigma q_1 \vee q_2 & \textit{when} & v \models_\sigma q_1 \textit{ or } v \models_\sigma q_2 \\
v \models_\sigma \neg q & \textit{when} & \neg(v \models_\sigma q) \\
v \models_\sigma \mu Q.q & \textit{when} & v \models_\sigma q[\mu Q.q/Q]
\end{array}$$

*A query $q$ is satisfied by a substitution $\sigma$ on a hiforest $H$, written $H \models_\sigma q$, if it is satisfied on each outermost root vertex of $H$, i.e., $\forall v. \mathit{isroot}_{\to_{\mathrm{s}} \cup >_i}(v) \Longrightarrow v \models_\sigma q$.*

Def. 4 works by navigating in a fixed hiproof $h$ to find satisfying vertices $v$. Because a vertex determines a sub-hiproof, this is equivalent to a structural definition as given in [2], which works by decomposing the subject hiproof during navigation, defining a relation $s \models_\sigma q$. Note that in this model **atomic** is defineable as an empty box: **atomic** $nm =$ **inside** $nm\ (\neg *)$.

**Definition 5 (Query interpretation).** *Let $H$ be an ordered hiforest and $q$ a query. Then we define the interpretation of $q$ in $H$ as the set of satisfying substitutions:* $[\![q]\!]_H = \{\, \sigma \mid H \models_\sigma q \,\}$.

Our language is expressive but queries can be expensive. In [2] we gave a naive algorithm for $[\![q]\!]$ using unification to instantiate variables, which is exponential in the number of match variables. Recursion and match variable unification unavoidably affect the data complexity of our queries (see basic results e.g., [1, 12, 18]). For large proofs, we would want a fragment that is more feasible but captures most desirable examples. The following proposition is the denotational counterpart of a similar proposition in [2].

**Proposition 1.** *Given a hiproof $H$, one of its vertices $v$ and a variable $X$, there is a query $Q(v, X)$ which locates $v$ at $X$, i.e.,* $[\![Q(v, X)]\!]_H = \{\sigma\}$ *with* $\sigma(X) = v$.

## 6 Transformations and their semantics

We now introduce the core update operations formally. Note that we do *not* want to allow arbitrary "tree surgery" of the hiproof structure; we want update operations to preserve semantic validity. Updates have the syntax:

| | |
|---|---|
| $u ::=$ **box** $X_r$ **to** $X_1 \ldots X_n$ **as** $l$ | add nested box around $X_r \ldots X_1 \ldots X_n$ |
| **unbox** $X$ | unfold nested box at $X$ |
| **rename** $X$ **as** $l$ | change label on box at $X$ |
| **refine** $X$ **with** $s$ | add a new sub-hiproof at $X$ |
| **deletetree** $X$ | delete subtree at $X$ |
| **replace** $X$ **by** $Y$ | replace subtree at $X$ by that at $Y$ |

The **box** operation is the most interesting. It introduces a nested box, whose contents are nodes in the partial subtree with $X_r$ as root and $X_1 \ldots X_n$ as leaves. This allows us to gather to an arbitrary depth, using a query to select either end of the path; this is useful to package up repeated applications of rules. The other update operations are straightforward to understand. An update is *applied* by combining with a query to instantiate node variables in a hiproof, written as **update** $u$ $q$. This matches $q$ to the root of the hiproof; a more common pattern is to search the hiproof for matches, as seen in the examples in Sect. 2.2. This is written and defined as $u$ **where** $q =$ **update** $u$ (**somewhere** $q$).

### 6.1 Interpretation of transformations

We can specify positions in a hiproof, but we still need to solve a well-known problem with tree and graph updates. Suppose a query picks out several nodes and a transformation changes the structure; then simultaneous updates may *overlap*. The result may be ill-defined, or may depend on the execution order. The semantics as given here is based on single-valued answers to queries; where a query has several answers, there may be several update results, representing applying the operation to different positions in the tree. To have a global effect, the update results may be *merged* if they do not conflict, or we may simply repeatedly apply a query and update. We are not yet investigating implementation in detail, so making any such choices for PrQL could be premature; we prefer to first pin down an accurate semantics. Later on, we plan to extend the language to allow more efficient constructs, avoiding multiple passes and using type systems to ensure safety; we will relate back to the present, intended semantics.

To interpret updates, we use the operations in Sect. 4 and extra definitions:

(i) A combinator to transform a subforest of $H$ with a function $f$:

$$\mathit{at}(H, v, f) = \mathit{graft}(\mathit{chop}(H, v), f(\mathit{cover}(H, v)), v)$$

(ii) The *box* operator specialised to box only down to vertices $v_1, \ldots, v_n$:

$$\begin{aligned} \mathit{addbox}(H, l, v_1, \ldots, v_n) = \mathit{graft}(\ & \mathit{box}(l, \mathit{chop}_n(H, v_1, \ldots, v_n)), \\ & \mathit{cover}_n(H, v_1, \ldots, v_n), v_1, \ldots, v_n) \end{aligned}$$

where $\mathit{chop}_n(H, v_1, \ldots, v_n)$ and $\mathit{cover}_n(H, v_1, \ldots, v_n)$ are the obvious generalisations of *chop* and *cover* to $n$ arguments.

(iii) To add or remove boxes at the subforest given by $v_r$:

$$\begin{aligned} \mathit{addboxat}(H, v_r, v_1, \ldots, v_n) &= \mathit{at}(H, v_r, \lambda H.\mathit{addbox}(H, l, v_1, \ldots, v_n)) \\ \mathit{unboxat}(H, v_r) &= \mathit{at}(H, v_r, \mathit{unbox}) \end{aligned}$$

(iv) To change the label of a vertex: let $H = \langle V, L, \leq_{\mathrm{i}}, \to_{\mathrm{s}}, \lesssim \rangle$, $v \in V$ and $l \in L$, then $L'$ is defined as $L'(v') = (l, \gamma)$ for $v' = v$, where $L(v) = (l', \gamma)$ and $L'(v') = L(v)$ otherwise. Then $\mathit{relabel}(l, H, v) = \langle V, L', <_i, \to_{\mathrm{s}}, \lesssim \rangle$.

**Definition 6 (Interpretation of transformations).** *Let $H$ be a hiproof and $q[X_1 \ldots X_n]$ a query with match variables instantiated by $\sigma$. The meaning of an update wrt $\sigma$ is a partial function, defined when the RHS is defined:*

$$\begin{aligned} [\![\mathbf{box}\ X_r\ \mathbf{to}\ X_1 \ldots X_n\ \mathbf{as}\ l]\!]_H^\sigma &= \mathit{addboxat}(H, l, \sigma(X_r), \sigma(X_1), \ldots, \sigma(X_n)) \\ [\![\mathbf{unbox}\ X]\!]_H^\sigma &= \mathit{unboxat}(H, \sigma(X)) \\ [\![\mathbf{rename}\ X\ \mathbf{as}\ l]\!]_H^\sigma &= \mathit{relabel}(H, \sigma(X), l) \\ [\![\mathbf{refine}\ X\ \mathbf{with}\ s]\!]_H^\sigma &= \mathit{graft}(H, [\![s]\!], \sigma(X)) \\ [\![\mathbf{deletetree}\ X]\!]_H^\sigma &= \mathit{chop}(H, \sigma(X)) \\ [\![\mathbf{replace}\ X_1\ \mathbf{by}\ X_2]\!]_H^\sigma &= \mathit{graft}(\mathit{chop}(H, \sigma(X_1)), \mathit{cover}(H, \sigma(X_2)), \sigma(X_1)) \end{aligned}$$

$$[\![\mathbf{update}\ u\ q]\!]_H = \{\, [\![u]\!]_H^\sigma \mid \sigma \in [\![q]\!]_H \textit{ and } [\![u]\!]_H^\sigma \textit{ is defined} \}$$

Def. 6 gives a non-deterministic semantics; the result may be empty (if operations are undefined) or there may be several results (for different instantiations). We do not say anything here about how to combine several results into one, as this may depend on the implementation; as hinted above, an implementation may encode our core operations using a more general update language. In this setting, a better alternative would be to give criteria which guarantee a deterministic result. For the same reason, we do not yet investigate complexity results.

## 7 Related work and conclusions

This paper introduced an update extension of PrQL, a query language for hiproofs. We interpret queries and transformations using denotational semantics of hiproofs, which are graph-like structures subject to well-formedness constraints. We showed that the basic operations are enough to capture desirable transformations, and that they preserve well-formedness and the connection to underlying proof trees.

*Connections in theorem proving.* As larger proof developments are being constructed, people are starting to explore ways to investigate them. Besides PrQL, a query language has been proposed for OmDoc proofs [22]. The *Proviola* tool [24] provides another means for proof understanding, by recording the output issued by an interactive proof during its execution development; impressively, it has been used to annotate source code of large proofs in both Coq (the Feit-Thompson proof [17]) and HOL Light (Hales's Flyspeck proof [23]). However, Proviola sheds no light on a proof that proceeds in a single tactic execution step. A hiproof-based tool would allow more dynamic exploration, by zooming into proof objects to look at the fine detail — although the practical details of managing such large proof objects will be challenging. Other researchers have used proof as the subject for search and machine learning (e.g., [19, 25]). Again this work might be usefully adapted to proof trees.

Conversely, we hope that our work can be adapted to transforming proof scripts. Rather than altering the extracted proof trees for HOL Light, we might want to impose the structural changes on the input proofs themselves, where possible. Work has been started on tools and foundations for *proof refactoring* towards this [5, 15, 27], but it is challenging: it requires understanding the meaning of input proof scripts, and how to transform them. By contrast, it is much easier to manipulate recorded output proof structures.

*Update languages for structured data.* There is a large body of work from the last decade on query and update languages for general forms of structured data. PrQL was inspired by, among others, UnQL [7] and Graph Logic [9]; the latter was extended to Context Logic to consider updates [8] and the former extended to a language of functional transformations [11], in the setting of XML Update. The approach taken by the W3C to extend XQuery [10] has a more SQL-like flavour, similar to our approach.

*Transformations and hierarchy.* To study PrQL updates and extensions further, fundamental results on tree queries [18], transformation operations [16] and complexity [4] should be possible to adapt. However, without restricting our language we are unlikely to improve on earlier complexity results [2], so instead we want to focus on translation into an efficient underlying XML or graph-based system. Having worked out the language design and semantics, we need to use the right level of abstraction before translation, taking *hierarchy* as a native construct. Hierarchical graphs have recently been studied in another setting, for structuring *safety cases* in a hierarchical way, providing a tool that performs transformations like those studied here [13]. Related ideas for managing hierarchy in understanding provenance have recently been proposed [6].

*Future and ongoing work.* Several extensions to our update language are desirable; at the least, to add constructs for composing and iterating transformations. Before pursuing that, we want to extend our practical experiments to transformations. Taking the implementation of hiproofs in HOL Light [21], we can output them in a form suitable for a graph database system such as Neo4j [26], which can store and process very large structures on disk. Some of our queries and transformations can be captured in Neo4j's query and update language *Cipher*, although it remains to investigate how efficient the encoding is; alongside practical experiments, we need to give a further theoretical analysis.

*Acknowledgements.* The authors thank James Cheney and Domagoj Vrgoc for helpful discussions.

## References

[1] Alfred V. Aho. Algorithms for finding patterns in strings. In: *Handbook of theoretical computer science (vol. A)*. Ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990, pp. 255–300.

[2] David Aspinall, Ewen Denney, and Christoph Lüth. Querying proofs. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. 2012, pp. 92–106.

[3] David Aspinall, Ewen Denney, and Christoph Lüth. Tactics for Hierarchical Proof. In: *Mathematics in Computer Science* 3.3 (2010), pp. 309–330.

[4] Pablo Barceló Baeza. Querying graph databases. In: *Proceedings of the 32nd symposium on Principles of database systems*. 2013, pp. 175–188.

[5] Timothy Bourke et al. Challenges and Experiences in Managing Large-Scale Proofs. In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. LNCS 7362. Springer, 2012, pp. 32–48.

[6] Peter Buneman, James Cheney, and Egor V. Kostylev. Hierarchical models of provenance. In: *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*. 2012, pp. 10–10.

[7] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. In: *The VLDB Journal* 9.1 (2000), pp. 76–110.

[8] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '05. New York, NY, USA: ACM, 2005, pp. 271–282.

[9] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A Spatial Logic for Querying Graphs. en. In: *Automata, Languages and Programming*. Ed. by Peter Widmayer et al. LNCS 2380. Springer, 2002, pp. 597–610.

[10] Donald D. Chamberlin et al. *XQuery Update Facility 1.0 (W3C Recommendation)*. 2011.

[11] James Cheney. FLUX: functional updates for XML. In: *SIGPLAN Not.* 43.9 (2008), pp. 3–14.

[12] Rance Cleaveland and Bernhard Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In: *Proceedings of the 3rd International Workshop on Computer Aided Verification*. CAV '91. London, UK, UK: Springer-Verlag, 1992, pp. 48–58.

[13] Ewen Denney, Ganesh Pai, and Iain Whiteside. Hierarchical Safety Cases. In: *NASA Formal Methods*. Ed. by Guillaume Brat et al. LNCS 7871. Springer, 2013, pp. 478–483.

[14] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A Hierarchical Notion of Proof Tree. In: *ENTCS* 155 (2006), pp. 341–359.

[15] Dominik Dietrich, Iain Whiteside, and David Aspinall. POLAR: A Framework for Proof Refactoring. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. 2013.

[16] Hartmut Ehrig. *Fundamentals of algebraic graph transformation*. English. Berlin; New York: Springer, 2006.

[17] Georges Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy et al. LNCS 7998. Springer, 2013, pp. 163–179.

[18] Martin Grohe and Nicole Schweikardt. Comparing the succinctness of monadic query languages over finite trees. In: *RAIRO - Theoretical Informatics and Applications* 38.4 (2004), pp. 343–373.

[19] Jónathan Heras and Ekaterina Komendantskaya. ML4PG in Computer Algebra Verification. In: *Intelligent Computer Mathematics*. Ed. by Jacques Carette et al. LNCS 7961. Springer, 2013, pp. 354–358.

[20] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. In: *ACM Trans. Internet Technol.* 3.2 (2003), pp. 117–148.

[21] Steven Obua, Mark Adams, and David Aspinall. Capturing Hiproofs in HOL Light. In: *Intelligent Computer Mathematics*. Ed. by Jacques Carette et al. LNCS 7961. Springer, 2013, pp. 184–199.

[22] Florian Rabe. A Query Language for Formal Mathematical Libraries. In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. LNCS 7362. Springer, 2012, pp. 143–158.

[23] Carst Tankink et al. Formal Mathematics on Display: A Wiki for Flyspeck. In: *Intelligent Computer Mathematics*. Ed. by Jacques Carette et al. LNCS 7961. Springer, 2013, pp. 152–167.

[24] Carst Tankink et al. Proviola: A Tool for Proof Re-animation. In: *Intelligent Computer Mathematics*. Ed. by Serge Autexier et al. LNCS 6167. Springer, 2010, pp. 440–454.

[25] Josef Urban et al. MaLARea SG1 - Machine Learner for Automated Reasoning with Semantic Guidance. In: *Automated Reasoning*. Ed. by Alessandro Armando et al. LNCS 5195. Springer, 2008, pp. 441–456.

[26] Chad Vicknair et al. A comparison of a graph database and a relational database: a data provenance perspective. In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE '10. New York, NY, USA: ACM, 2010, 42:1–42:6.

[27] Iain Whiteside et al. Towards Formal Proof Script Refactoring. In: *Intelligent Computer Mathematics*. Ed. by James H. Davenport et al. LNCS 6824. Springer, 2011, pp. 260–275.

## A Additional technical details

**Definition 7 (Grafting).** *Let $H = \langle V, L, \leq_{\mathsf{i}}, \rightarrow_{\mathsf{s}}, \lesssim \rangle$ be a valid hiforest with $H \models g_1 \longrightarrow g$. Let $v_1, \ldots, v_n$ be distinct vertices in $V$, with $L(v_i) = (\bullet, \gamma_i)$ (and hence $n \leq length(g)$). Let $H' = \langle V', L', \leq_{\mathsf{i}}', \rightarrow_{\mathsf{s}}', \lesssim' \rangle$ be another hiforest with $H \models g' \longrightarrow g_2$, so it has $n$ overall roots $\{v_{r1} \ldots v_{rn}\} \in V'$ ordered by $\lesssim'$ with $L(v_{ri}) = (l_i, \gamma_i)$. Suppose (wlog) $V \cap V' = \emptyset$.*

*Then we can define a new hiforest by*

$$graft(H, H', v_1, \ldots, v_n) = \langle V - \{v_1 \ldots v_n\} \cup V', L|_{V - \{v_1 \ldots v_n\}} \cup L', \leq_{\mathsf{i}}'', \rightarrow_{\mathsf{s}}'', \lesssim'' \rangle$$

*The relations $\leq_{\mathsf{i}}''$, $\rightarrow_{\mathsf{s}}''$ and $\lesssim'$ are defined by:*

$$v \leq_{\mathsf{i}}'' w \textit{ iff either } \begin{cases} v \leq_{\mathsf{i}} w \ \wedge \ w \notin \{v_1 \ldots v_n\} \\ v \leq_{\mathsf{i}}' v_{ri} \ \wedge \ v_i \leq_{\mathsf{i}} w \\ v \leq_{\mathsf{i}}' w \end{cases}$$

$$v \rightarrow_{\mathsf{s}}'' w \textit{ iff either } \begin{cases} v \rightarrow_{\mathsf{s}} w \ \wedge \ w \notin \{v_1 \ldots v_n\} \\ v \rightarrow_{\mathsf{s}} v_i \ \wedge \ v_{ri} \rightarrow_{\mathsf{s}}' w \\ v \rightarrow_{\mathsf{s}}' w \end{cases}$$

$$v \lesssim'' w \textit{ iff either } \begin{cases} v \lesssim w \ \wedge \ w \notin \{v_1 \ldots v_n\} \\ (v \lesssim v_i \ \wedge \ v_{ri} = w) \ \vee \ (v = v_{ri} \ \wedge \ v_i \lesssim' w) \\ v \lesssim' w \end{cases}$$

*If $H$ has exactly $n$ holes $v_1, \ldots, v_n$ (i.e., $g = [\, \gamma_1, \ldots, \gamma_n \,]$ and $L(v_i) = (\bullet, \gamma_i)$), then we write $graft(H, H')$ as an abbreviation.*

**Definition 8 (Cover).** *Given a hiforest $H = \langle V, L, \leq_{\mathsf{i}}, \rightarrow_{\mathsf{s}}, \lesssim \rangle$ and vertex $v \in V$, we define the* cover *of $v$ as all nodes below or inside $v$ by $V' = cover_{\rightarrow_{\mathsf{s}} \cup >_i}(v)$, where the cover of a relation $R$ is defined as $cover_R(x) = \{y \mid x \, R^* \, y\}$. and the labellings and orderings restricted accordingly:*

$$cover(H, v) = \langle V', L|_{V'}, \leq_{\mathsf{i}}|_{V' \times V'}, \rightarrow_{\mathsf{s}}|_{V' \times V'} \lesssim |_{V' \times V'} \rangle.$$

When defining the chopping operation, we do not take out the node $v$, but replace its label with $\bullet$ to make it a dangler:

**Definition 9 (Chopping).** *Given a hiforest (or hiproof) $H = \langle V, L, \leq_{\mathsf{i}}, \rightarrow_{\mathsf{s}}, \lesssim\rangle$ and vertex $v$, then we define a new hiforest without nodes below or inside $v$ by setting $V' = (V - \mathit{cover}_{\rightarrow_{\mathsf{s}} \cup >_i}(v)) \cup \{v\}$ and*

$$\begin{aligned}\mathit{chop}(H, v) = \langle\, & V',\, L|_{V - \mathit{cover}_{\rightarrow_{\mathsf{s}} \cup >_i}} \cup \{v \mapsto (\bullet, \gamma) \mid L(v) = (l, \gamma)\},\\ & \leq_{\mathsf{i}}|_{V' \times V'}, \rightarrow_{\mathsf{s}}|_{V' \times V'}, \lesssim|_{V' \times V'} \rangle\end{aligned}$$

We can generalise *chop* and *cover* to $n$ arguments. Chopping $n$ vertices removes them sequentially from $H$, whereas the cover of $n$ vertices is a hiforest with $n$ roots:

$$\begin{aligned}\mathit{chop}_1(H, v_1) &= \mathit{chop}(H, v_1)\\ \mathit{chop}_n(H, v_1, \ldots, v_n) &= \mathit{chop}_{n-1}(\mathit{chop}(H, v_1), v_2, \ldots, v_n)\\ \mathit{cover}_1(H, v_1) &= \mathit{cover}(H, v_1)\\ \mathit{cover}_n(H, v_1, \ldots, v_n) &= \mathit{cover}(H, v_1) \cup \mathit{cover}_{n-1}(H, v_2, \ldots, v_n)\end{aligned}$$

To avoid notational difficulties when dealing with more than one root simultaneously, we define boxing and unboxing only for hiproofs. The definitions extend easily to hiforests by boxing reach root of the forest separately (although that is not needed in this paper). Note how the danglers in $H$ are not included in the box introduced with $\mathit{box}(l, H)$.

**Definition 10 (Boxing and Unboxing).** *Given a non-empty hiproof $H = \langle V, L, \leq_{\mathsf{i}}, \rightarrow_{\mathsf{s}}, \lesssim\rangle$ with overall root $v_r$, i.e., $\mathit{isroot}_{\rightarrow_{\mathsf{s}} \cup >_i}(v_r)$, then the* boxing *of $H$ with a label $l$ is defined as*

$$\begin{aligned}\mathit{box}(l, H) = \langle\, & V \cup \{*\}, L \cup \{* \mapsto (l, \gamma) \mid L(v_r) = (l', \gamma)\},\\ & \leq_{\mathsf{i}} \cup \{(v, *) \mid v \in V, L(v) = (l, \gamma) \wedge l \neq \bullet\}, \rightarrow_{\mathsf{s}}, \lesssim \cup \{(*, *)\}\rangle\end{aligned}$$

*The* unboxing *removes such a box (if it exists): let $H = \langle V, L, \leq_{\mathsf{i}}, \rightarrow_{\mathsf{s}}, \lesssim\rangle$, then we define*

$$V' = \begin{cases} V - \{r\} & \mathit{isroot}_{\rightarrow_{\mathsf{s}} \cup >_i}(r), L(v) = (l, \gamma) \wedge l \neq \bullet\\ V & \mathit{otherwise}\end{cases}$$

*Then:*

$$\mathit{unbox}(H) = \langle V', L|_{V'}, \leq_{\mathsf{i}}|_{V'}, \rightarrow_{\mathsf{s}}, \lesssim|_{V'}\rangle$$

By careful inspection of the operation definitions we can show that the resulting hiforests indeed satisfy the conditions of Def. 1 and preserve semantic validity as stated earlier.

**Proposition 2 (Operations and validity).** *The semantic operations preserve the hiforest conditions and moreover, preserve semantic validity of hiproofs with the expected input-output goals.*

The final part of justifying our definitions is to show that the interpretation of updates is well-defined, when query results are given and refinement has the

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

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

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

right shape. Specifically, **refine** $X$ **with** $s$ requires that when $\sigma(X) = v$ and the subtree at $v$ has validity $\mathit{chop}(H, v) \;\models\; g_1 \longrightarrow g_2$, then the term given denotes a hiforest with the same input-output shape.

For this we need to show that syntactic hiproof terms denote valid tree structures. This is shown together with the definition of $[\![s]\!]$. Validity for syntactic hiproof terms is written as $s \;\vdash\; g_1 \longrightarrow g_2$, meaning that the hiproof $s$ takes a list of input (proven) goals $g_1$ to produce a list of output (unproved) goals $g_2$, and is defined by the rules in Fig. 2.

**Definition 11 (Interpretation of hiproof terms).** *The definition of* $[\![s]\!]$ *is by induction on the syntactic validity* $s \;\vdash\; g_1 \longrightarrow g_2$*, defining* $[\![s]\!]$ *and establishing at the same time that* $[\![s]\!] \;\models\; g_1 \longrightarrow g_2$*. The cases are:*

- $\boxed{a \;\vdash\; \gamma \longrightarrow [\,\gamma_1,\ldots,\gamma_n\,].}$ *Then* $[\![a]\!]$ *is the* $n+1$ *point hiforest with nodes* $a, x_1, \ldots, x_n$*. We set* $a {\rightarrow_{\mathrm{s}}} x_i$*,* $L(a) = (a, \gamma)$ *and each* $x_i$ *is a "dangler", so* $L(x_i) = (\bullet, [\gamma_i])$*.*
- $\boxed{\mathsf{id} \;\vdash\; \gamma \longrightarrow \gamma.}$ *Then* $[\![\mathsf{id}]\!]$ *is the hiforest with one "dangler" node* $*$*, where* $L(*) = (\bullet, [\gamma])$*.*
- $\boxed{[l]\,s \;\vdash\; \gamma \longrightarrow g_2.}$ *Then* $[\![[l]\,s]\!] = \mathit{box}(l, [\![s]\!])$ *since* $[\![s]\!]$ *has a unique top-level root.*
- $\boxed{s_1 \,;\, s_2 \;\vdash\; g_1 \longrightarrow g_2.}$ *Then* $[\![s_1 \,;\, s_2]\!] = \mathit{graft}([\![s_1]\!], [\![s_2]\!])$*. The premises of the validity rule and the induction hypothesis ensure that the grafting operation is well-defined.*
- $\boxed{s_1 \otimes s_2 \;\vdash\; g_1 {}^\wedge g_2 \longrightarrow g_1' {}^\wedge g_2'.}$ *Then* $[\![s_1 \otimes s_2]\!]$ *is the hiforest formed by disjoint union of* $[\![s_1]\!]$ *and* $[\![s_2]\!]$*, with the ordering relation* $\lesssim$ *extended on the roots and dangling nodes.*
- $\boxed{\langle\rangle \;\vdash\; [\,] \longrightarrow [\,]}$. $[\![\langle\rangle]\!]$ *is the empty hiforest.*

Note that denotational hiproofs are unique only up to the choice of node set $V$; two hiproofs which have the same structure and labelling but differ only on $V$ are isomorphic [14]. The definitions above work with particular hiproofs, but it can be verified that the choice of node names (but not labels!) is unimportant.