Hierarchical Proof transformation in HOL Light

Konrad Stojanowski

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh
2017
Abstract

A notion of a hierarchical proof (hiproof) has been introduced recently to capture a notion of a structured proof tree that preserves the partial ordering of the tactics and allows to put additional structure on top of that to make a proof more human readable. *PrQL* is an expressive hiproof query and transformation language that allows to inspect a hiproof to better understand the underlying formal proof and improve on the overall structure of the hiproof. The aim of this project is to implement the hiproof transformation subset of *PrQL* for *HOL Light* interactive theorem prover that allows to transform hiproofs recorded using the *HipCam* patch to *HOL Light*. The correctness of the implementation is backed with a relevant test suite.
Acknowledgements

I wish to express my sincere thanks to my project supervisor, Dr. David Aspinall, who has supported me with his thorough knowledge of the subject and who devoted his time to help me better understand and deliver this project.

I would like to express my gratitude to my partner, Dawid, for his continuous support and encouragement throughout these two years.

I would also like to acknowledge my younger brothers, Damian and Rafal, for their help and support throughout this year.
Contents

1 Introduction 9
  1.1 Motivation ............................................ 9
  1.2 Contribution of the project .......................... 11
  1.3 Structure of the report ................................ 11

2 Background 13
  2.1 Logic for Computable Functions ....................... 13
  2.2 HOL Light .............................................. 14
    2.2.1 Example of a proof ............................... 15

3 Hiproofs 17
  3.1 Towards the definition of a hiproof ................... 18
  3.2 Hiproof_2 representation .............................. 19
    3.2.1 Child ordering relation .......................... 20
  3.3 Formal definition of an ordered hiproof ............... 20
    3.3.1 Danglers ......................................... 20
    3.3.2 Labelling ......................................... 20
  3.4 Operations ............................................ 21
  3.5 Fixing the graft operation ............................ 23

4 PrQL 25
  4.1 Structural clause ..................................... 26
  4.2 Transformation clause ................................ 27
  4.3 Performing the transformations ...................... 28
    4.3.1 Boxing a subhiproof .............................. 28
    4.3.2 Unboxing a subhiproof ............................ 29
    4.3.3 Renaming a node .................................. 29
    4.3.4 Interpretation of transformations ................. 29
    4.3.5 Putting it all together .......................... 30

5 Investigation and Design 31
  5.1 Design decisions ..................................... 31
    5.1.1 Scope of the project ............................. 31
    5.1.2 Adjusting PrQL to HOL Light ....................... 31
  5.2 Evaluation of the current PrQL implementation ....... 32
    5.2.1 Lexer and parser .................................. 32
CONTENTS

D PrQL 12 details
   D.1 Derived forms ........................................... 75
   D.2 Name and goal matching .................................. 75

E PrQL type system ............................................. 77

Bibliography ..................................................... 79
Chapter 1

Introduction

1.1 Motivation

In the age of the internet of things, self-driving cars, and mobile phones, computers become inextricable companions of our everyday lives. It is easy to forget, that the computer is quite a recent offspring of an ancient branch of science that has been developing with human civilization for thousands of years, namely logic. All in all, the fathers of the modern computer, Alan Turing and Alonzo Church, were both logicians and mathematicians.

As computer science stems from logic, it is obvious that apart from performing numerical tasks, one can employ computers in solving logical puzzles.

*Theorem proving* is a branch of computer science that deals with verification of logical and mathematical statements using computers. These are usually formalized using a suitable logical framework and fed into a software tool called a *theorem prover* together with an accompanying recipe that guides the tool towards establishing or refuting correctness of the original statement.

Due to its high expertise and resource requirements, theorem proving had not been used in practice as much as other branches of computer science until recently. This started to change as the benefits of formal verification became quite obvious also outside the field of mathematics. For example, software verification is a promising field of study that may completely change the way we perceive the issues of software correctness and security. Its abilities go way beyond what is available to the programmer through current tools, including type analysis or testing. It allows to inspect the program holistically as a model and verify our belief about what it should do (the *specification*) against what does it actually do.

The tools that allow us to achieve these results are called *theorem provers*. A *theorem prover* is a piece of software that, being logically correct itself\(^1\), can

\(^1\)Correctness of a theorem prover is ideally established and confirmed through research, see [9, 14] for details concerning the correctness of *HOL Light*. 
Chapter 1. Introduction

A formal proof can be represented by its derivation tree describing the order of proof rules or tactics (subproofs and logical equivalences) used to reach the conclusion of the proof. Natural deduction proofs, of which example is shown in Figure 1.1, may be an example of a derivation tree in the land of Peano Axioms. However, this representation of a formal proof is specific to the theorem prover or logical system and shows how the machine represents the proof in its own terms. Such structure can be different from the way a human being looks at the proof, and taking into consideration the size and complexity of formal proofs nowadays, it can be completely unintelligible for a human to even read proofs in such a form.

To enhance the user experience with formal proofs and theorem provers, a notion of hiproof has been introduced by E. Denney et al. in 2006 [5]. A hiproof is an inverted tactic tree structure that allows to abstract from that derivational structure of a proof and overlay it with a more user-friendly structure, in particular, to add hierarchy nesting so that the user can hide zoom in on a subproof. Moreover, hiproofs may be applied to proof planning thanks to the fact that they allow incomplete hiproofs through the danglers.

What is unique about hiproofs is that they allow passing goals from arbitrarily deep level of nesting not only to deeper levels, but also to shallower levels of nesting [5]. In other words, a hiproof box may enclose a partial subtree ‘exporting’ goals. An example of a hiproof is presented in Figure 1.2.
A natural complement of any data structure (e.g. XML) is a mechanism of data extraction and manipulation (e.g. XPath, XQuery). Proof Query Language, or PrQL for short, is an SQL-like language that has been designed to query and transform hiproofs [1, 2].

PrQL has been introduced in 2012 by D. Aspinall et al. [1] as a language to inspect hiproofs. The language has been extended in 2013 to allow for proof transformations [2]. To differentiate between these versions where needed, the former will be referred to as PrQL12 and the latter as PrQL13.

1.2 Contribution of the project

The aim of this project is to explore the design of PrQL12 and PrQL13 by providing an experimental implementation of PrQL13 for the HOL Light interactive theorem prover. This work will extend further the existing implementation of PrQL for HOL Light covering the PrQL12 subset of the language that has been delivered in the first stage of the project [18]. This stage of the project aims at delivering the hiproof transformation functionality as introduced in PrQL13 [2]. The project will rely on the HipCam hiproof recording tool for HOL Light for hiproof generation implemented by S. Obua [15] and improved in the first stage of the report. The implementation will be preceded by an investigation phase where relevant algorithms and query satisfaction rules will be designed or revisited if necessary.

The existing test suite arguing correctness and soundness of the implementation will be extended to take into consideration hiproof transformations.

1.3 Structure of the report

The report is divided into eight chapters, each describing a further stage of the project. The report has been structured in such a way that it introduces the reader to the world of theorem proving, then discusses the theoretical side of hiproofs, and describes the design decisions behind the implementation, and finally presents the reader with an overview and evaluation of the implementation with an outlook towards future way of improving and extending it.

Chapter 1 Motivation and aim of the project.

Chapter 2 Introduction to the HOL Light theorem prover to let the reader familiarize themselves with the background of the project.

Chapter 3 Introduction to the notion of a hierarchical proof tree transitioning from an intuitive description to the formal definition of a hiproof. Explanation of the hiproof transformation operations and their formal definition.
Chapter 4 Introduction of the proof query language beginning with a more informal discussion and examples with more technical details towards the end of the chapter.

Chapter 5 Discussion of the design decisions and principles for the hiproof transformation engine and reasons behind some changes to the Part 1 code.

Chapter 6 Details of the implementation.

Chapter 7 Evaluation of the implementation through examples and discussion of the test suite.

Chapter 8 Conclusion and summary of the project. Discussion of the possible future extensions to the project.

The report is complemented by five Appendices (A-E), that introduce more detail on HOL Light’s theorem proving (Appendix A), discuss the theory behind hiproofs (Appendices B and C) and PrQL (Appendix D), and the PrQL type system as introduced in part 1 of the report (Appendix E).
Chapter 2

Background

This chapter aims to introduce the reader into the world of theorem proving, especially the HOL Light interactive theorem prover, with a toy example of a proof using this ITP.

2.1 Logic for Computable Functions

The idea that computers could be used to prove theorems has accompanied computer science from its inception. In 1954, Martin Davies, from the Princeton University, implemented the algorithm for proving statements in the Presburger arithmetic and managed to prove that the sum of two even numbers is even [4]. Although this might not sound like an achievement today, the limitations of the early computers and the worse than exponential complexity of the algorithm could not yield finer results. In general, theorem proving remained more an experiment rather than a useful tool for a longer period of time.

Soon after that, in 1969, Dana Scott invented the Logic for Computable Functions (the name given by Robin Milner), a logical system based on the typed $\lambda$-calculus that was intended for reasoning about recursively defined functions [7, 17]. This can be perceived as a departure from then-prevalent mathematical flavour of automated reasoning, towards more computation oriented one.

The theory behind Logic of Computable Functions has been set to practice by Robin Milner who implemented the Stanford LCF theorem prover in 1972. Later on, the Meta Language (ML) functional programming language has been invented at the University of Edinburgh by Milner and his team for the sake of implementation of the second generation LCF-based theorem prover, the Edinburgh LCF [7].

Robin Milner’s work on theorem proving had a major impact on computer science as a whole. A large family of programming languages has sprung from the ML programming language, with Standard ML and OCaml being some examples of members of that family. Moreover, the LCF-based approach to theorem proving
gave rise to another family, the family of LCF-based theorem provers, with HOL Light and Isabelle/HOL being the most popular ones [7, 19].

Of course, the range of available theorem provers is wider than the LCF family. Other important theorem provers that had a significant impact on the field are Coq, Mizar, and NuPRL, each of them exhibiting different approaches to theorem proving.

2.2 HOL Light

The HOL Light theorem prover belongs to the HOL family of theorem provers which is a branch of the larger family of LCF-based theorem provers. The HOL theorem prover, which gave rise to the whole HOL family, has been created out of the need for a theorem prover that could facilitate hardware verification [7].

Significantly, HOL Light has been created with the aim to provide hardware verification just as the original HOL. However, since its creation, HOL Light has been applied to a wide range of problems, from the verification of the floating-point algorithms at Intel to proving the Kepler conjecture by Thomas Hales et al. [8, 10].

The HOL Light theorem prover has been implemented in OCaml, a programming language that evolved from ML through Standard ML and has been developed at Inria [10, 13]. One can notice a striking parallel between Edinburgh LCF being implemented in ML and the HOL Light that has been implemented in OCaml, what shows these ideas are constantly evolving and being improved on.

HOL Light is renowned for its relatively small kernel consisting of 12 operations that allow to manipulate logical statements, the terms. The kernel has been proven to be sound and correct through extensive research. The process of verifying HOL Light’s kernel included the design and implementation of CakeML programming language. What is significant, CakeML’s compiler has been verified to behave accordingly with its specification [12, 14].

To remove any possibility of introducing falsehood to any proof, the OCaml’s type system is used to lock down the ability to manipulate the terms to be confined only to the kernel [10].

The operations implemented in the kernel can be chained and composed together to form tactics which are larger building blocks for the proofs. On top of this, a tactics that are parametrized by another tactics are called tacticals. A statement to be proven is called the goal and HOL Light allows for both constructive and destructive (forward and backward) proofs, and the HOL Light’s flexible proof style allows for mixing both of these approaches [11].

Moreover, it is significant that HOL Light does not keep the history of a proof, only the goal. Hence, the HipCam module for HOL Light is used extensively in this project to record and visualise (hi)proofs [7, 15].
2.2.1 Example of a proof

In *HOL Light*, the main way of interacting with the theorem prover is through *OCaml’s read-eval-print loop* and each interaction is a well-formed *OCaml* expression evaluated by the language shell.

As said before, *HOL Light*’s logical system is based on the λ-calculus, hence it allows to prove statements about computable functions. As a toy example, let us consider a language with the following grammar:

\[
\begin{align*}
  s & \rightarrow \text{Num int} \\
  s & \rightarrow \text{Add } s s
\end{align*}
\]

A statement in this language consists either of a number, constructed using *Num* or a sum of expressions, constructed using *Add*. Although *HOL Light* allows for algebraic data types, *HOL Light* data types are a separate notion and can be declared using the *define_type* function. The abstract syntax tree of such a language can be expressed as *HOL Light*’s datatype defined in the following way:

```ocaml
let exp_INDUCT, exp_RECURSION =
  define_type "exp = Num num | Add exp exp" ;;
```

The *define_type* function returns two theorems that facilitate reasoning about the newly defined data type: the induction (*exp_INDUCT*) and the recursion (*exp_RECURSION*) [11]. Let us now define function *eval* that will evaluate an expression of type *expr* to a numerical value:

```ocaml
let EVAL = define
  `(\a. eval (Num a) = a) /\ (\e0 e1. eval (Add e0 e1) = (eval e0) + (eval e1))` ;;
```

As both the data type and the function have now been embedded in *HOL Light*, some properties of the implementation can be proven. For example, we may want to check if a statement expressing a sum of two given numbers is correctly evaluated to the sum of the two numbers, that is

\[
\forall n, m. \text{eval} (\text{Add} (\text{Num } n) (\text{Num } m)) = n + m.
\]

To prove the above statement, it is necessary to express it in the *HOL Light* syntax. A statement to be proven is called a *goal* and has the *OCaml* type *term*. *HOL Light* provides a parser extension for *OCaml* especially for expressing *goals* which are enclosed in backticks (e.g. `‘2 + 2’`). When discussing *hiproofs*, a distinction will be made between an *incoming goal* (an *ingoal*) of a tactic that is its input, and an *outgoing goal* (an *outgoal*) that is the result of applying a tactic on an *ingoal*.

Moreover, a sequence of tactics that will form the backward proof of the statement needs to be identified. That can usually be done using the *interactive* mode and the details of this, together with the exact meaning of the proof, have been discussed in Appendix A. The *prove* function exposed by *HOL Light*, takes
exactly a statement and the sequence of tactics and attempts at proving the statement using that sequence of tactics. The function returns a theorem if it succeeds to prove the statement and fails otherwise:

```plaintext
# let EVALSUM =
  prove (\! n m. eval (Add (Num n) (Num m)) = n + m',
  INDUCT_TAC THEN INDUCT_TAC THEN REWRITE_TAC[EVAL]);;

val EVALSUM : thm = |- !n m. eval (Add (Num n) (Num m)) = n + m
```

The grammar and the eval function discussed above are tiny examples of how verification can be facilitated using HOL Light. However, this example is enough for the reader to understand the principles behind the theorem proving using HOL Light. The rest of the report will concentrate on hiproofs which are in a way separate from the theorem prover and no further knowledge of HOL Light is necessary.
Chapter 3

Hiproofs

Every formal proof can be visualised as a proof tree that resembles the derivational structure of the proof, which means that it shows the partial ordering of tactics, subtactics, and other building blocks of the proof from the point of view of the theorem prover.

A hierarchical proof, or a hiproof, is a proof tree that grows from a root at the top downwards and allows for hierarchical nesting of nodes. Hiproofs are very flexible in their structure as they allow leaving open (unproven) goals and passing goals between nodes at arbitrary levels of nesting. In other words, a hiproof is a proof tree that has been extended with an additional layer of hierarchy [5]. This additional layer allows for grouping and nesting nodes of the proof to achieve higher levels of abstraction.

There are a few characteristics of hiproofs that set them apart from current approaches to the tree-like representations of a proof. First of all, hiproofs concentrate on tactics and the relations between them, not on the goals. For example, a natural deduction tree, which is also a kind of a proof tree, treats goals as the first-class citizens, and makes tactic labelling a mere addition that can be omitted at will.

Moreover, theorem provers usually represent the proofs as directed acyclic graphs [3]. Hiproofs are abstracting away from this representation as well. They exhibit a structure of a tree what is closer to the approach taken by the pure formal logic than the particular implementation.

This chapter presents an overview of the theory behind hiproofs from [5] and [2] to give the reader an intuition about what hiproofs are. Appendix B complements this chapter by discussing the formal definitions behind ordered hiforests and Appendix C discusses the technical details of operations on ordered hiforests.
3.1 Towards the definition of a hiproof

One could define a generic proof tree as a tree \( \langle V, L, \rightarrow_s, r \rangle \) where \( V \) is the set of vertices, \( L \) is a labelling function \( L : V \rightarrow \Lambda \) that maps any node to its label, \( \rightarrow_s \) is the function composition relation, and \( r \) is the root [5]. This structure is definitely flat and does not exhibit any interesting features except showing the order of composition of the tactics.

As it has been said before, a hiproof is a proof tree with an additional layer of abstraction that allows for nesting nodes at arbitrary levels. Let us augment the above definition of a proof tree with an additional partial order relation \( \leq_i : V \times V \) called the inclusion relation that whenever two nodes \( v \leq_i w \) one says that \( v \) is inside \( w \). A complement of the inclusion relation is the proper containment relation \( >_i : V \times V \), where \( w >_i v \) when \( w \) contains \( v \).

There are a few constraints imposed on the structure of a hiproof governing how exactly does the newly introduced inclusion relation interact with the original sequencing relation:

1. the arrows always target the outer nodes,
2. the arrows always emanate from the inner nodes,
3. inclusion and sequence are mutually exclusive - a node cannot have children with respect to both \( \leq_i \) and \( \rightarrow_s \),
4. there is a single root of a hiproof nested within a node [5].

![](image.png)

Figure 3.1: Composition of two nodes on two different levels of nesting.

The hiproof allows for arbitrary levels of nesting, what means that any two nodes that are in a sequence \( u \rightarrow_s v \) can be at two different levels of nesting, and admits a situation where the former is at a higher level of nesting (deeper) than the latter as depicted in Figure 3.1 (an arrow points from a node \( u \) inside of a nested box and targets a node \( v \) outside of that nested box).

Let us inspect such a situation where the targeted node is at the zeroth level of nesting (\( \forall u \in V. v \not\leq_i u \)). As node \( v \) does not have any parents with respect to the \( \leq_i \) relation, it becomes a root of another tree within the \( \leq_i \) relation. In the above example there are no nodes inside of or below \( v \), hence it becomes a single-node tree.

Generalizing the intuitive reasoning above, one can say that the inclusion relation \( \leq_i \) forms a forest on the set of nodes \( V \) rather than a single tree, because any node that is at the \( i^{th} \) level of nesting becomes a root of such a tree on its own.

As a consequence, the sequencing relation \( \rightarrow_s \) breaks into a forest under the inclusion relation as well. To get an intuition about this, let us consider vertices \( u, v, \) and \( w \) such that \( u \rightarrow_s v \rightarrow_s w \). To box node \( v \) inside a newly introduced
node, \( r \), it is required to break the \( u \rightarrow s v \) path into \( u \rightarrow s r \) and \( v \leq_i r \), with the relation \( v \rightarrow s w \) left intact. In this case, \( v \) becomes a root of another tree with respect to the sequencing relation.

Hence, a **hiproof** can be loosely defined so far as a tree-like structure \( H = \langle V, L, \leq_i, \rightarrow_s, r \rangle \) that consists of:

- a set of vertices \( V \),
- a labelling function \( L : V \rightarrow \Lambda \) that maps the vertices to their labels,
- the inclusion relation \( \leq_i : V \times V \) that creates a forest \( i = \langle V, \leq_i \rangle \),
- the sequencing relation \( \rightarrow_s : V \times V \) that also creates a forest \( s = \langle V, \rightarrow_s \rangle \),
- the root vertex \( r \).

This representation of a hiproof is equivalent to the **type one hiproof** (\( \text{Hiproof}_1 \)) as described in [5]. Section 3.2 presents another, equivalent approach to describing the hiproof structure.

### 3.2 Hiproof$_2$ representation

In [5] the authors propose a representation of the hiproof alternative to the **type one hiproof**, called **Hiproof$_2$**. This representation, although equivalent to **Hiproof$_1$**, is more implementation-oriented and the **HipCam** hiproof recording module for HOL Light that will be used in the implementation later on employs a data structure based on this representation.

The **Hiproof$_1$** representation consists of two forests created by \( \leq_i \) and \( \rightarrow_s \) and is defined as \( H_1 = \langle V, L, \leq_i, \rightarrow_s, r \rangle \). The **Hiproof$_2$** representation glues the two forests back into a single tree by introducing the **connectedness relation** \( \rightarrow \) and the notion of the **inclusion level function** \( l : V \rightarrow \mathbb{N} \). It is important to differentiate the composition relation \( \rightarrow_s \) used in hiproof of type one from the **connectedness relation** \( \rightarrow \) used in the hiproof of type two that combines composition and inclusion.

The inclusion level function maps the node to its **level of nesting** or **inclusion level** which basically conveys the inclusion relation \( \leq_i \) in a different form. The \( l \) function is subject to three conditions [5]:

1. \( l(r) = 0 \), a node that has no parents with respect to the \( \leq_i \) relation is at the 0$^{th}$ level of inclusion,
2. if \( v \rightarrow v' \), then \( l(v') \leq l(v) + 1 \), meaning that if two nodes are connected, then they are either in a sequence (\( v \rightarrow_s v' \) when \( l(v') < l(v) \)) or \( v' \) is nested within \( v \) (\( v' \leq_i v \) when \( l(v') = l(v) + 1 \)),
3. if \( v \rightarrow v_1, v \rightarrow v_2 \), and \( l(v_1) = l(v) + 1 \), then \( v_1 = v_2 \), which forbids a node from having both nodes within and nodes following it$^1$.

$^1$As discussed in Appendix B, inclusion and sequence are mutually exclusive.
Hence, the hiproof of type two can be defined as $H_2 = \langle V, L, \rightarrow, l, r \rangle$. This representation is advantageous for implementation as the $\rightarrow$ relation, which forms the backbone of the data structure, can be implemented using a single tree. This backbone is augmented by the $l$ function that informs us of the character of the edges between nodes in the tree.

### 3.2.1 Child ordering relation

The hiproof structure can be augmented with the child ordering relation $\preceq$ that imposes a total left-to-right ordering on children nodes of a single parent and a partial order on all nodes of a hiproof in general. The direction of the ordering is arbitrary, however to be consistent with the convention, let us assume the left-to-right direction of children as the default from now on.

This property will turn to be useful for both defining the operations on hiproofs as well as for the implementation.

### 3.3 Formal definition of an ordered hiproof

Appendix B gives the formal definition of an ordered hiforest and an ordered hiproof as introduced in [2]. Although this detailed definition is crucial for implementation, the high level discussion in this chapter should be sufficient to get an intuition about hiproofs in general.

#### 3.3.1 Danglers

Hiproofs support incomplete proofs using the notion of a dangler. The dangler is a node that denotes an unproven (or open) goal. Such nodes are labelled as $\bullet$ in [2], and as specified in the 6th condition of the definition (see Appendix B), only the leaves may have the $\bullet$ label.

#### 3.3.2 Labelling

Let us consider the labelling function $L : V \rightarrow \Lambda$ and the set of labels $\Lambda$ itself. The actual structure of elements in $\Lambda$ could be arbitrary, but let us define $\Lambda$ to be at least a tuple $\Lambda = (\lambda \cup \{\bullet\}, \Gamma)$ where:

- $\lambda$ is the set of possible names of tactics used in the hiproof,
- the dot $\bullet$ is a special label that describes an unproven goal,
- $\Gamma$ is the set of goals that appear in that proof.
3.4 Operations

The initial notion of a hiproof has been refined into the ordered hiproof by adding the child ordering relation \( \preceq \) to reflect the derivational system’s left to right ordering of inference rules and to allow to formalize operations on hiproofs which will preserve their validity.

There are five basic operations that can be performed on a hiproof: cover, chop, graft, box, and unbox. This section introduces the operations by describing them briefly to give the reader a basic understanding of how they work. Appendix C discusses the technical details behind these operations as necessary for the implementation.

Each of the operations will be followed by a graphical example of the operation being applied on a hiproof \( H \) presented in Figure 3.2a.

Let us assume a valid hiproof \( H = \langle V, L, \leq, \rightarrow, \preceq \rangle \) with \( H \models g \rightarrow g' \) where \( g \) is the set of incoming goals (ingos) and \( g' \) is the set of outgoing goals (outgos) and in this specific case, danglers. All operations below are defined for a valid hiproof and guarantee to yield a valid hiproof as well.

Given a node \( v \in V \), the cover operation returns all nodes within and below node \( v \), including \( v \) itself. If the label on \( v \) is \( L(v) = (l, \gamma_v) \), then the cover of \( v \) is \( H_v \models \gamma_v \rightarrow g'' \) where \( g'' \) is a subset of \( g' \). A cover of node \( D \) from the exemplary hiproof cover \( (H, D) \) is presented in Figure 3.2b.

The chop operation removes a cover of a node \( v \) and labels that node within the original hiproof as a dangler. Hence, if the original hiproof is \( H \models g \rightarrow g' \) and the label on the node \( v \) is \( L(v) = (l, \gamma_v) \), then the resultant hiproof is \( H_r \models g \rightarrow g' \cup \gamma_v \) (assuming there are no danglers in the cover) and \( L(v) = (\bullet, \gamma_v) \). An example of
Chapter 3. Hiproofs

(a) \textit{box}(H) \hspace{1cm} (b) An example of boxing a subhiproof.

Figure 3.3: Different approaches to boxing.

applying the \textit{chop} operation on the hiproof \(H\) at node \(D\) as \(\text{chop}(H, D)\) has been presented in Figure 3.2c.

Grafting is an operation that reverses chopping, as given two hiforests \(H_1\) and \(H_2\), grafting \(H_2\) onto \(H_1\) connects back the roots in hiforest \(H_2\) with danglers within \(H_1\) hiforest. Given two hiforests \(H_1 \models g \rightarrow g'\) and \(H_2 \models g' \rightarrow g''\), grafting \(H_2\) onto \(H_1\) will result in a hiproof \(H \models g \rightarrow g''\).

The chop and cover operations are inverse to cover as a hiproof that has been chopped, can have the chopped subhiproof grafted again:

\[ \text{graft}(\text{chop}(H, v), \text{cover}(H, v)) = H \]

Let us consider the hiproof \(H\) as shown in Figure 3.2a. Figure 3.2b presents the cover of node \(D\) expressed as \(\text{cover}_D = \text{cover}(H, D)\) and Figure 3.2c shows the result of \(\text{chop}_D = \text{chop}(H, D)\). By grafting \(\text{cover}_D\) back on \(\text{chop}_D\) using the graft operation \(\text{graft}(\text{chop}_D, \text{cover}_D)\) we will get back to the initial hiproof \(H\) from Figure 3.2a.

There are also two operations that allow for introducing and removing additional levels of hierarchy: \textit{box} and \textit{unbox}. Given a hiproof \(H\), the \textit{box} operation will create a new node and nest the hiproof \(H\) within it which can also be explained as ‘drawing a box’ around the hiproof \(H\). Boxing the whole hiproof \(H\) in a box \(G\) is shown in Figure 3.3a.

However, adding a box around the whole hiproof is not the most useful operation one could imagine. Section 4.3.1 introduces \textit{transformations} derived from the basic \textit{operations} that allow to box subhiproofs. An example of boxing nodes \(D\), \(E\), and \(F\), into a box \(G\) expressed as \(\text{addboxat}(H, G, D, [\])\) is presented in Figure 3.3b. The \textit{addboxat} operation also allows for limiting the extent of the box from the below, for example leaving the \(F\) node out of the box.
3.5 Fixing the graft operation

When working on the project an error has been spotted in the definition of the graft operation as presented in the original paper introducing hiproof transformations [2]. The graft operation has been defined as:

\[
\text{graft}(H, H', v_1, ..., v_n) = \langle V - \{v_1, ..., v_n\} \cup V', L|_{V' - \{v_1, ..., v_n\}} \cup L', \lesssim''_{i, \rightarrow''_{s, \lesssim''}} \rangle
\]

The definition of the child ordering relation \(\lesssim''\) in the resultant hiforest has been incorrectly defined as:

\[
v \lesssim'' w \text{ iff either } \begin{cases} 
v \lesssim w \land w \notin \{v_1, ..., v_n\} \\
(v \lesssim v_i \land v_{ri} = w) \lor (v = v_{ri} \land v_i \lesssim' w) 
\end{cases}
\]

The error lies in the second case for \(v \lesssim'' w\) which reads incorrectly as follows:

\[
v \lesssim'' w \text{ if } (v \lesssim v_i \land v_{ri} = w) \lor (v = v_{ri} \land v_i \lesssim' w)
\]

The second clause in this case is equivalent to saying: ‘\(v\) is a root in \(H'\) and its corresponding dangler \(v\) in \(H\) is left of \(w\) in \(H'\),’ which is incorrect due to the fact that \(v_i\) belongs to \(V\), not to \(V'\), from the definition. The error lies in the redundant \('\) above the \(\lesssim\) which is obviously a typo. The second case of the \(v \lesssim'' w\) should read as follows:

\[
v \lesssim'' w \text{ if } (v \lesssim v_i \land v_{ri} = w) \lor (v = v_{ri} \land v_i \lesssim w)
\]
Chapter 4

PrQL

Hiproofs are complex structures, and as a size of a single proof can grow to hundreds of thousands of lines of code [8], browsing through such a huge hiproof can be quite tiresome. Proof query language, or PrQL, has been designed as a language to extract and refine hiproofs.

Initially, PrQL had been created only as a language to extract information such as labels or goals from hiproofs [1]. Since then, it has been extended to allow hiproof transformations [2]. The first part of the project dealt only with information extraction [18]. As this part of the project deals with hiproof transformation, this chapter will present the PrQL language in full, however discussing it more from the hiproof tranformation angle.

PrQL queries exhibit a structure similar to SQL, as they are declarative in style. A typical PrQL query consists of two clauses: the structural and the transformation clauses in the form of $u$ where $q$ with $q$ being the former and $u$ the latter.

The right hand side clause, called the structural clause, describes what information from the hiproof should be extracted. The left-hand side clause of the query, called the transformation clause, describes what transformation should be performed on the hiproof given the data from the right-hand side clause. An example of a query removing a subhiproof with a label ‘ARITH_TAC’ would take form as follows:

\[
\text{deletetree } X \text{ from } h \text{ where somewhere atomic 'ARITH_TAC' and at } X
\]

The second line of the query describes the nodes of interest where the transformation deletetree should be applied. This clause of the query is looking for all nodes which have label ‘ARITH_TAC’ on them and assigns all such nodes to

---

1This name has been chosen in the Part 1 report to distinguish between the structural, name, and goal matching, but can be applied here as well to distinguish the two clauses of the query [18].

2The distinction between the structural and transformation clauses has been introduced by this report.
Chapter 4. PrQL

$q ::= * \quad \text{anything non-empty}
| \text{atomic } nm \quad \text{label on a node}
| \text{inside } nm \ q \quad \text{nesting}
| q_1 \text{ then } q_2 \quad \text{sequencing}
| q_1 \text{ beside } q_2 \quad \text{child ordering}
| \text{ingoals } gm \quad \text{in-goals of a node}
| \text{outgoals } gm \quad \text{out-goals of a node}
| \text{at } nm \quad \text{pointer to a sub-proof}
| \text{nothing} \quad \text{empty hiproof}
| \mu Q.q \quad \text{recursive query}
| q_1 \land q_2 \quad \text{conjunction}
| q_2 \lor q_2 \quad \text{disjunction}
| \neg q \quad \text{negation}

Figure 4.1: PrQL’s basic queries for the structural clause [1, 2].

variable $X$. The first line of the query describes what should be done with these nodes. In this case, the query will remove the cover of a node under the $X$ variable. Section 5.6 discusses thoroughly the interpretation of the transformations, especially in cases when the structural query is ambiguous and yields more than one result.

This chapter gives an overview of PrQL concentrating on the transformation queries as introduced in [2]. The unique contribution of this chapter is the discussion of the PrQL inner workings at length.

4.1 Structural clause

This section discusses the very basics of the structural clause of a PrQL query. The structural clause has been introduced in [1] and implementation of this part of the language was dealt with in part 1 of the project. Since this project does not go into much detail with the structural clause, further details of PrQL 12 have been presented in Appendix D.

All basic queries for the structural clause are presented in Figure 4.1. These queries let the user describe the structure that is to be found within the hiproof.

The atomic query allows inspecting a label on a node without any structure within, while the inside query allows to inspect a node that has both a label and its nested subhiproof.

Using the then and beside queries one can express structural connection between two subqueries, while $\land$ and $\lor$ (and and or in the implementation) are logical connectives. The ingoals and outgoals queries inspect the incoming and outgoing goals of a node.

The at query is in a way the PrQL counterpart of the at combiner as discussed in
Transformation clause

As discussed before, the transformation clause of the PrQL query describes the transformation that should be performed on the hiproof basing on the data fed from the structural clause.

There are six transformation queries as shown in Figure 4.2, and one projection query that allows to pass the values of the variables back to the user.

The box operation introduces a nested box around a subhiproof. The subhiproof to be boxed is defined by two characteristics:

1. a root node ($X_r$) which will be the root node of the subhiproof within the new box,
2. a set of border nodes ($X_1, ..., X_n$) that will be the delimiting points where the box ends.

The unbox transformation, which is an inverse of box, simply unfolds a nested box from around a subhiproof. The transformation takes a pointer to the node which should be unfolded.

The third transformation, rename, changes the label on a set of given nodes ($X$) to a given name ($l$). Of course, the set of ingoals is left intact as allowing a user to change that on purpose could lead to the hiproof losing its validity.

The refine transformation is unique among all transformations as it operates on two hiproofs. Let us assume a situation where once a huge proof is done, a part of it is extracted and proven separately for some reason (e.g. a more concise way to prove it has been found). Ideally, we would like to be able to include its hiproof back to the old hiproof. The refine transformation allows to graft another (external) hiproof on an already existing hiproof (assuming the ingoals match) in defined by the user places.

A dangler can occur as a result of either the proof being still unfinished or a part of a hiproof being removed on purpose. The deletetree transformation simply removes the subhiproof being a cover of a single, given node and relabels the root node.
\[
\begin{align*}
\text{u} & := \text{box} \ X_r \text{ to } X_1, \ldots, X_n \text{ as } l \quad \text{add nested box} \\
\text{unbox} \ X & \quad \text{unfold nested box at } X \\
\text{rename } X \text{ as } l & \quad \text{rename box at } X \\
\text{refine } X \text{ with } s & \quad \text{add a new subhiproof } s \text{ at } X \\
\text{deletetree } X & \quad \text{delete subhiproof at } X \\
\text{replace } X \text{ by } Y & \quad \text{replace subhiproof at } X \text{ by that at } Y \\
\text{select } X_1, \ldots, X_n & \quad \text{projection of the structural clause result}
\end{align*}
\]

Figure 4.2: PrQL’s updates for the transformation clause [1, 2].

node of that subhiproof as a dangler. The open goal can be filled in by refinement or replacement operation later on.

The replace operation takes pointers to two nodes within the same hiproof and replaces a node with a copy of a cover of the other node given the ingoals are the same for both of them.

The last query, select, which has been introduced originally in [1], unlike the other six introduced afterwards in [2], acts simply as a projection and passes the result of the structural clause to the user. This is useful when exploring the hiproof to extract any information from the hiproof. It can be also used to debug the transformation queries as no operation will be performed, but the user can see which nodes will be affected.

### 4.3 Performing the transformations

In Section 3.4 the five basic hiproof operations have been introduced: cover, chop, graft, box, and unbox, accompanied with the at combiner. This section discusses applying these operation to performed the transformations presented in Section 4.2.

Before moving on to defining the interpretation of the transformations, let us first derive three useful definitions that will simplify reasoning about the transformations.

#### 4.3.1 Boxing a subhiproof

The box operation as defined in Section 3.4, adds box around a whole cover of a given node. However, the transformation box allows to limit the extent of the box from below.

To be able to provide this functionality, let us define operation addbox as follows:

\[
\text{addbox}(H, l, v_1, \ldots, v_n) = \text{graft}(\text{box}(l, \text{chop}_n(H, v_1, \ldots, v_n)), \text{cover}_n(H, v_1, \ldots, v_n))
\]

In the above equation, the chop$_n$ and cover$_n$, are generalizations of the respective functions for $n$ vertices. What the addbox operation achieves is that it allows to
limit the extent of the box from below by firstly chopping these off the hiproof, then boxing the resultant subhiproof, and finally grafting back the previously chopped off subhiproofs back on the boxed subhiproof. This employs the characteristic of the box function which leaves the dangling nodes outside of the box.

The addbox operation lacks one final piece that would let us achieve functionality required for the box transformation: the ability to choose the root of the newly created box. Let us recall the at combiner which allows applying an operation at a given subhiproof:

\[ at(H, v, f) = graft(chop(H, v), f(cover(H, v))) \]

Finally, using the at operation, the addboxat operation can be defined that will fulfil all requirements of the box transformation:

\[ addboxat(H, l, v_r, v_1, ..., v_n) = at(H, v_r, \bigwedge_H.addbox(H, l, v_1, ..., v_n)) \]

### 4.3.2 Unboxing a subhiproof

The unbox operation is applied to the root of the hiproof, not to any chosen subhiproof. The problem here is identical to the one posed by the original box operation. To fix this, the at combiner should be applied as follows:

\[ unboxat(H, v) = at(H, v, unbox) \]

The newly defined unboxat operation will apply the unbox operation on a chosen subhiproof what fulfils the requirements posed by the unbox transformation.

### 4.3.3 Renaming a node

The initial five operations do not provide the ability to rename a node, since they deal only with the structure of a hiproof and renaming a node is not a structural operation per se.

Hence, let us define another operation, rename, as follows:

\[ rename(H, v, l) = \langle V, L\mid V - \{v\} \cup \{v \mapsto (l, \gamma) \mid L(v) = (l', \gamma)\}, \leq, \rightarrow_s, \preceq \rangle \]

The above operation relabels node \( v \) from \( L(v) = (l', \gamma) \) to \( L(v) = (l, \gamma) \). It can be generalized to \( rename_n \) just like the chop and cover before.

### 4.3.4 Interpretation of transformations

As all necessary operations have now been defined, the exact meaning of the PrQL transformations can be expressed. Let \( H \) be a hiproof, \( q \) be the structural
clause of the query on that hiproof satisfiable by a variable assignment \( \sigma \), and \( u \) being the transformation clause of the query. The notation \( \llbracket u \rrbracket_H^\sigma \) expresses the interpretation (result or meaning) of query \( u \) with respect to hiproof \( H \) and a variable assignment \( \sigma = \llbracket q \rrbracket_H \). Figure 4.3 presents the interpretation of the six PrQL hiproof transformations as defined in [2].

\[
\begin{align*}
\text{[box } X_r \text{ to } X_1, \ldots, X_n \text{ as } l]_H^\sigma &= \text{addboxat}(H, l, \sigma(X_r), \sigma(X_1), \ldots, \sigma(X_n)) \\
\text{[unbox } X]_H^\sigma &= \text{unboxat}(H, \sigma(X)) \\
\text{[rename } X \text{ as } l]_H^\sigma &= \text{rename}(H, \sigma(X), l) \\
\text{[refine } X \text{ with } s]_H^\sigma &= \text{graft}(H, [s], \sigma(X)) \\
\text{[deletetree } X]_H^\sigma &= \text{chop}(H, \sigma(X)) \\
\text{[replace } X \text{ by } Y]_H^\sigma &= \text{graft}(\text{chop}(H, \sigma(X)), \text{cover}(H, \sigma(Y)), \sigma(X))
\end{align*}
\]

Figure 4.3: Interpretation of the queries in the transformation clause [2].

### 4.3.5 Putting it all together

The above discussion has defined both structural and transformation clauses of a PrQL query. It is important how these connect to create a PrQL query. Let \( q \) be a structural clause and \( u \) be the transformation clause. A PrQL query can be constructed using the where connective as follows:

\[ u \ \text{where} \ q. \]

For example, to remove the cover of every atomic node with label \( \text{EQ,MP} \), the following query needs to be issued:

```
deletetree $X$ (from s) where somewhere atomic 'EQ,MP' and at $X$.
```

The from \( s \) part of the query has been parenthesised as it does not constitute an integral part of a query from the theoretical point of view, but will be used in the implementation to define which hiproof to perform the operation on.
Chapter 5
Investigation and Design

This chapter discusses all the necessary design decisions towards implementation of proof transformations for PrQL in HOL Light, including any additional theoretical work necessary to complete the project, and other decisions that have been made to make PrQL more adjusted to HOL Light.

5.1 Design decisions

Before more detailed investigation can be performed, it is crucial to frame the scope of the project, make some design decisions, and set some principles to be able to judge completeness of the implementation at the end of the project.

5.1.1 Scope of the project

The aim of this part of the project is to deliver the hiproof transformations extension to PrQL as described in [2] for the HOL Light automated theorem prover within the HipCam hiproof recording framework. This part of the project builds on top of the hiproof querying engine delivered in Part 1. The implementation will cover all six transformation queries and will be accompanied by a comprehensive test suite to check the implementation’s correctness.

5.1.2 Adjusting PrQL to HOL Light

PrQL is ITP-unaware (or ITP-neutral) by design [1, 2], what means that it abstracts away from the underlying derivational systems, which in this case would be HOL Light. Although this has been the initial goal of hiproofs in general, the implementation of PrQL should be made ITP-aware (without losing much generality).
A goal in HOL Light is expressed using backticks (e.g. $\forall x.x + 1 < 5$ becomes `!x.x + 1 < 5`), and this notation is used in this implementation as well (e.g. \texttt{ingoals} [\texttt{!x.x + 1 < 5}]). Although this decision has been made in Part 1, it is important to reiterate this due to the major changes that have been introduced to the lexer and parser in this part of the project.

As discussed in [18], labels on nodes, as created by HipCam, can take arbitrary values, hence to differentiate constants (strings) from variables, each variable will start with $\$, e.g. $\$X$ instead of just $X$. From now on, the report will follow this convention to disambiguate variables from constants.

A special case of the above are the danglers, which in the formal notation are labelled as $\bullet$. Using this character in the implementation would not be a reasonable thing to do, as the $\bullet$ character (or any similar one) does not appear on any widely used keyboard layout. Instead, a new query, \texttt{iscut} will be introduced:

$$
\texttt{iscut} := \texttt{atomic} \bullet
$$

One could be tempted to define the \texttt{iscut} query in terms of \texttt{inside} or \texttt{islabel} queries for the nested node as well, however as specified in the 6th constraint in Appendix B, the $\bullet$ label can be placed only on a leaf without any nested structure within (an \texttt{atomic} box).

### 5.2 Evaluation of the current PrQL implementation

Since a considerable amount of time has passed since the end of Part 1 of the project, it is worth to revisit and evaluate the existing codebase to identify possible areas for improvement before moving to the Part 2 implementation.

The investigation has been divided into two phases, the lexer and parser and the query satisfaction engine, as any changes applied to the latter will probably impact the modules downstream.

#### 5.2.1 Lexer and parser

When implementing Part 1 of the project the decision was made to write a custom lexer and parser for PrQL. The advantages of this solution were mainly the then perceived time efficiency and the ability to create custom error messages what would make that user-facing module more friendly towards the user.

However, soon into the first phase of the project the solution was found to be inflexible and hard to maintain, but has been kept due to time constraints. Since the transformation part of PrQL introduces a few additional structures to the language, it is a good moment to revisit and evaluate the hand crafted lexer and
5.2. Evaluation of the current PrQL implementation

parser. As adding the new constructs to the already existing custom lexer and parser would make the already complex code even more complicated, the lexer and parser generators have been considered.

There are two main parser generators for OCaml, namely ocamlyacc and Menhir. The ocamlyacc parser generator comes with the standard installation of OCaml as it is used by the OCaml compiler itself. On the other hand, Menhir is a newer, alternative solution, that accepts more complex grammars and allows for customizing error messages. Furthermore, Menhir accepts a language for describing grammars which is derived from the one used by ocamlyacc, and is 90% compatible with ocamlyacc [13, 16]. However, a huge advantage of ocamlyacc over the other solution is availability of numerous tutorials and comprehensive documentation that Menhir lacks for the moment.

Both parser generators discussed above rely on ocamllex, the OCaml’s own lexing solution, for tokenizing the input. However, ocamllex is not the only lexer generator currently available. Sedlex is a newer tool of that kind whose main advantages are the lack of code generation stage (sedlex works as a OCaml’s ppx rewrite module) and its support of Unicode. It exposes the same API as ocamllex, hence can be easily replaced for ocamllex when necessary [6, 13].

The decision has been made to use the standard set of ocamllex and ocamlyacc, since these tools arrive with the OCaml distribution, can generate code that is independent of any additional libraries and modules, are well documented, and have stood the test of time. PrQL’s grammar is not too complex to be expressed using ocamlyacc hence the only advantage that would have remained from employing Menhir to perform this task would be the ability to create custom error messages. However, the transition from ocamlyacc to Menhir is quite easy and could be performed anytime if necessary for the project’s further development.

5.2.2 Query satisfaction engine

The current query satisfaction engine, as delivered in the Part 1 of the project, divided the queries into two categories: basic, which were implemented ‘natively’, and derived, which were desugared back into basic queries. This division has been introduced in [1] and the implementation has followed it for the simplicity that comes from following this approach: if the base queries are proved to be correct, and the translation from the derived queries to the basic ones is proved correct as well, the derived queries should be correct as well.

Even though desugaring had many theoretical advantages, including easier negation handling, it proved to be computationally expensive as a single derived query had to be translated into a more complex structure that caused many redundant function calls.

The decision has been made to implement as many of the derived queries as possible into native for the engine to improve the performance of the system. As shown in Chapter 7.5, this improves the overall performance of the system.
5.3 Type system

Part 1 of the project introduced types to PrQL’s structural queries which are shown in Appendix E. As this part of the project deals with a broader subset of the language, the type system needs to be extended to take into consideration the new transformation clauses as well.

In Part 1 of the project, the set of types has been defined as:

\[ \tau ::= \text{atomic} | \text{label} | \text{goal} | \text{goals} | \text{rec} \]

This does not take into consideration the representation of a node which is required for the \texttt{at} query and the transformation clauses. Hence, the set of types needs to be extended to take into consideration the type \texttt{node}:

\[ \tau ::= \ldots | \text{node} \]

Moreover, the type checking and type inference rules need to be extended as well to include the \texttt{at} operator and the \texttt{node} type. Figure 5.1 presents the type checking rule for the \texttt{at} operator, while Figure 5.2 shows the type inference rule to that operator. The \texttt{at} operator has been included in the Part 1 implementation as a preparation for the extension.

The type checking and type inference rules also need to be extended to the transformation clauses to ensure that the queries are meaningful before they reach the transformation engine.

Figure 5.3 shows the type checking rules that have been introduced for the transformation clauses. \( \Gamma \) denotes the set of variable-type substitutions and a general judgement \( \Gamma \vdash \$V : \tau \) tells us that the substitution \( \Gamma \) implies variable \( \$V \) is of type \( \tau \). It is important to note that all variables in the realm of the transformation clauses are of type \texttt{node}.

The type inference rules for the transformation clauses are presented in Figure 5.4. A general judgement \( q \implies \Gamma \) means that query \( q \) implies variable-type substitution \( \Gamma \). For a PrQL query in \texttt{u where q} form, both \texttt{u} and \texttt{q} must imply the same \( \Gamma \).
5.3. Type system

\[ V : \tau \in \Gamma \quad \Gamma \vdash V : \tau \quad \Gamma \vdash \text{unbox } X_r \]

\[ \Gamma \vdash X_r : \text{node} \quad \Gamma \vdash X_1 : \text{node} \quad \ldots \quad \Gamma \vdash X_n : \text{node} \]

\[ \frac{\Gamma \vdash X_r : \text{node} \quad \Gamma \vdash X_1 : \text{node} \quad \ldots \quad \Gamma \vdash X_n : \text{node}}{\Gamma \vdash \text{box } X \text{ to } X_1, \ldots, X_n} \]

\[ \Gamma \vdash X_r : \text{node} \quad \Gamma \vdash \text{refine } X_r \text{ with } s \quad \Gamma \vdash \text{replace } X_r \text{ with } X_s \]

\[ \Gamma \vdash X : \text{node} \quad \Gamma \vdash \text{rename } X \text{ as } l \quad \Gamma \vdash \text{deletetree } X \]

Figure 5.3: Type checking rules for the transformation queries.

\[ q \Rightarrow \Gamma \]

\[ q \Rightarrow \Gamma \quad u \Rightarrow \Gamma \quad \text{replace } X \text{ with } Y \Rightarrow X, Y : \text{node} \]

\[ \text{box } X_r \text{ to } X_1, \ldots, X_n \Rightarrow X_r, X_1, \ldots, X_n : \text{node} \]

\[ \text{refine } X \text{ with } s \Rightarrow X : \text{node} \quad \text{unbox } X_r \Rightarrow X_r : \text{node} \]

\[ \text{rename } X \text{ as } l \Rightarrow X : \text{node} \quad \text{deletetree } X \Rightarrow X : \text{node} \]

Figure 5.4: Type inference rule for the transformation queries.
type 'a hiproof =
   Hi_atomic of int * label * 'a * ('a varmap)
| Hi_id of 'a
| Hi_cut of 'a
| Hi_var of varname * 'a
| Hi_label of label * 'a * 'a hiproof * 'a info
| Hi_tensor of 'a hiproof list * 'a info
| Hi_sequence of 'a hiproof list * 'a info
;

Figure 5.5: The 'a hiproof data structure used by HipCam.

5.4 Hiproof data structure

This project builds on top of the HipCam hiproof recording module for HOL Light created by Steven Obua [15]. The hiproofs captured by HipCam are of a parametrized type 'a hiproof, where 'a is the type of the incoming goal (which in case of HOL Light is (term list * term)). Parametrizing this data structure makes it more portable and easier to test.

The constructs of the 'a hiproof data type are presented in Figure 5.5. The \( \rightarrow_s \) relation is expressed by the Hi_sequence construct. The Hi_sequence, contrary to the \( \rightarrow_s \) relation, is not a binary operator, but accepts a list of nodes that are in a sequence (and on the same level of nesting). The same holds for the \( \leq_i \) relation expressed by Hi_tensor. The Hi_atomic construct represents a node without any structure within, while Hi_label describes a nested box. The Hi_cut represents a dangler and Hi_id is an identity box used to preserve the left-to-right child ordering. The Hi_var can be employ to deduplicate a hiproof.

It is crucial to understand how the 'a hiproof data type conveys the \( \rightarrow_s \) relation for two nodes on different levels of nesting. Let us return to the case discussed in Section 3.1 which is repeated in Figure 5.6 for convenience. The hiproof would be expressed using the 'a hiproof data type as:

\[
\text{Hi_sequence}([\]
   \text{Hi_label}(w, \)
   \text{Hi_atomic}(u, 1) \)
   \text{Hi_atomic}(v, 0) \]

where 1 and 0 are the number of outgoals of that atomic box. Although obviously it is true that \( u \rightarrow_s v \), this relation is not directly expressed in 'a hiproof what
is a major drawback of this structure. The fact that \( u \rightarrow_s v \) is conveyed indirectly using the left-to-right child ordering.

The \( \text{'a hiproof} \) data structure uses a list of nodes wherever a collection of nodes is in place. This way the left-to-right child ordering is enforced for nodes with the same parent. The fact that \( u \rightarrow_s v \) is derived from the fact that the hiproof within \( w \) has a single outgoal and the first node that can capture that outgoal is \( v \). This is true for any arbitrarily large \( \text{'a hiproof} \) structure. The issue gets more obvious for a larger hiproof, for an example the one presented in Figure 5.7. It is true that \( b \rightarrow_s d \) and \( c \rightarrow_s e \), but the \( \text{'a hiproof} \) data structure will not allow us to express it directly. Figure 5.7 shows how the hiproof in Figure 5.7 may be described using the \( \text{'a hiproof} \) data structure.

The data structure conveys only the information that \( b \) and \( c \) have 1 outgoal each, hence the number of outgoals of their immediate tensor is 2, and this information is propagated up so that the number of outgoals of the label is 2 as well. When the label is sequenced with the tensor containing \( d \) and \( e \), we can say that the tensor consumes the outgoals of the label as its ingoals and \( d \) and \( e \) consume a single outgoal from the label. The fact that \( \text{'a hiproof} \) preserves child ordering allows us to say that the outgoals are consumed in the same order as they appear (left-to-right), hence \( b \rightarrow_s d \) and not \( b \rightarrow_s e \).

This characteristic of the \( \text{'a hiproof} \) and its lack of unique pointers to nodes causes the necessity for an auxiliary data structure to avoid computation of child nodes for a node each time this information is required as discussed in Section 6.4.1.

### 5.5 HipCam constructive operations

The HipCam software that is being used to record hiproofs exposes a number of useful constructive operations that are being used in hiproof construction, but
can be also used for the transformations. The most useful functions are \texttt{sequence} and \texttt{tensor}.

The \texttt{sequence} takes a list of objects of type \texttt{a hiproof} and \texttt{sequences} them. The function not only wraps the list in the \texttt{Hi sequence} construct, but also ensures correctness of the sequence. For example, its does not allow to create an empty sequence or acts as an identity for a list of size 1. For a list of size 2 or more, the function also inspects the size of in and out arities of each element in the sequence to ensure the structure is sound.

The \texttt{tensor} function acts similarly to \texttt{sequence}, but establishes a child ordering relations amongst nodes instead of sequencing. It as well forbids construction of an empty tensor and acts as an identity for a single-element list.

These two constructive operations will be useful in re-composing hiproofs when implementing transformations to achieve greater confidence in validity of the result.

5.6 Interpretation of the transformation queries

It is worth recalling that a \textit{PrQL} query takes form \texttt{u where q} where \texttt{q} stands for the \textit{structural clause} that is answered with a \textit{set of variable substitutions} \( \Sigma = \bigcup_i \{ \sigma_i \} \) that is consumed by the \textit{transformation clause} describing the transformation that should be performed on the hiproof. Each substitution maps a variable to a particular result, be it a label or a node, for example \( \sigma(sX) \mapsto \text{Atomic Repeat} \) or \( \sigma(sX) \mapsto \text{Node u} \).

The \textit{PrQL} specification proposes that the meaning of \texttt{u where q} is interpreted as follows [2]:

\[
[u \ where \ q]_H = \{ [u]_H^\sigma \mid \sigma \in [q]_H \ and \ [u]_H^\sigma \ is \ defined \}.
\]

The above translates to a set of different hiproofs each of them being produced by applying the transformation with different variable substitution. However, the specification does not comment on how to compose a set of results into a single result and this section is aimed at addressing this issue.

Since a single \textit{structural clause} can lead to an arbitrary number of substitutions, it is necessary to define the outcome of a transformation when zero, one, or more than one possible substitutions arise.

A design decision has been made that in case when the set of substitutions is empty \( \Sigma = \emptyset \), the engine should yield no result to communicate the fact to the user. Returning the hiproof intact in such a case could be misleading therefore the implementation should be explicit about there being no outcome. Moreover, when there is a single substitution \( |\Sigma| = 1 \), the single possible transformation should be applied to the hiproof.
5.6. Interpretation of the transformation queries

The situation gets more complex when a query is ambiguous and there exists more than one possible result. Let $n$ be the size of $\Sigma$. First of all, the substitutions could be applied iteratively on a single hiproof so that the transformation is performed $n$ times and the resultant hiproof is the ‘sum’ of all transformations. Although this approach may seem valid, there is quite a significant flaw in it. For example, let us consider a `deletetree` $X$ query that yields two different substitutions $u$ and $v$ for $X$, $\Sigma = \{\{X \mapsto u\}, \{X \mapsto v\}\}$, where $u \in \text{cover}(v)$. Such an outcome raises a few issues. For example, if the transformations should be applied iteratively, do we start with removing $u$ then $v$ or the other way around? If $v$ is removed first, and then $u$ does not belong to the hiproof anymore, what should be an outcome of such a transformation?

Another example of such undefined behaviour may come from a query that attempts to draw boxes with the same root node, but different border nodes. Figure 5.9a shows a hiproof consisting of a sequence of three atomic boxes $u$, $v$, and $w$. Let us consider the following query on that hiproof:

\[
\text{box } X \text{ to } Y \text{ as } A \text{ where somewhere (atomic } u \text{ and at } X) \text{ then (atomic } v \text{ and at } Y) \text{ or (atomic } u \text{ and at } X) \text{ then } \ast \text{ then (atomic } w \text{ and at } Y)\]

The above query would yield the following set of substitutions:

\[
\Sigma = \{\{X \mapsto \text{Node } u, \ Y \mapsto \text{Node } v\}, \{X \mapsto \text{Node } u, \ Y \mapsto \text{Node } w\}\}.
\]

If the transformation based on the first substitution is applied, then a box is drawn around the first two nodes as shown in Figure 5.9b. Subsequently, applying the
transformation based on the second substitution would mean drawing a new box around the whole sequence from \( u \) to \( w \), what would result in the whole new box being pulled inside the already existing box as shown in Figure 5.9c.

The result would have been different had the transformations been applied in the reverse order, as a box would be drawn around all three nodes at first and then the second box would be drawn only around nodes \( u \) and \( v \), what would leave node \( w \) outside of the second box.

Another issue with this approach is that once a single transformation is applied and the overall structure of the hiproof is altered, the other substitution may not be a valid one with regard to the query anymore.

Since the approach of iterative application of transformations definitely raises many theoretical issues, a separate result can be computed for each substitution, so that the size of output is equal to the number of different substitutions just as the definition for the meaning of \( u \ where \ q \) in [2].

However, this approach is computationally expensive and can lead to waste of resources as only one of the hiproofs will be used at the end. For this reason, a design decision has been made that in case the query is ambiguous, the implementation will pick the first substitution and apply one transformation according to that transformation. Moreover, the user will be warned that more than one outcome is possible so that they can disambiguate the query on their own.

Another approach that would be more user friendly, but would require more interface engineering would be to ask the user to specify which substitution they had in mind.
Chapter 6
Implementation

This chapter discussed the implementation phase of the project and includes indepth the architecture of the system, reasons behind implementation decisions, and some more important data structures and algorithms.

6.1 Architecture review

First of all, it would be beneficial to introduce the reader to the architecture of the implementation. In Part 1, the system has been divided into two separate parts: the front end (the user-facing part) and the back end (the query satisfaction engine). The front end was then divided further into modules: lexer, parser, and type checker. The back end constituted a monolith from the design point of view, however its code has been divided into OCaml modules for code clarity.

In the second part of the project, there has been no change introduced into the architecture of the front end. However, another module has been added to the back end which is now divided into two modules: query satisfaction and proof transformation.

The overview of the architecture has been presented in Figure 6.1.

![Diagram describing architecture of the system.](image)

In Part 1, some work has also been done to enhance the hiproof recording module HipCam authored by Stephen Obua [15]. This piece of software will be more
central in this part of the project as the hiproof operations as discussed in Section 3.4 will be delivered as an OCaml’s module in HipCam.

6.2 Revisiting the front end

To implement the transformation queries as discussed in Section 4.2 some changes need to be introduced into the front end of the system. The biggest changes are going to happen to the lexer and parser which will be replaced with completely new modules. Minor changes will be introduced to the typechecker as well.

6.2.1 Changes to the abstract syntax tree

The abstract syntax tree that is the main product of the front end has been improved significantly. This has been enabled by the decision to abandon the hand crafted parser which was the major consumer of that data structure.

At the high level, the AST has type hiprql which corresponds to the seven PrQL queries:

\[
\text{type hiprql} = \\
| \text{Select of vars \ast string \ast prql}
| \text{Box of var\_name \ast var\_name list \ast var\_name \ast string \ast prql}
| \text{Unbox of var\_name \ast string \ast prql}
| \text{Rename\_as of var\_name \ast string \ast string \ast prql}
| \text{Refine\_with of var\_name \ast string \ast string \ast prql}
| \text{Delete\_tree of var\_name \ast string \ast prql}
| \text{Replace\_by of var\_name \ast var\_name \ast string \ast prql}
\]

The prql type above corresponds to the structural clause of the PrQL query:

\[
\text{type prql} = \\
| \text{And of prqlist}
| \text{At of name\_match}
| \text{Atomic of name\_match}
| \text{Axiom of name\_match}
| \text{Beside of prqlist}
| \text{Callrec of ident\_name}
| \text{Everywhere of prql}
| \text{Ingoals of goal\_match}
| \text{Isbeside}
...
The abstract syntax tree of PrQL queries has undergone an overhaul compared to the original implementation. Naturally, the hiprql type has been extended with the six transformation queries. Moreover, the name and goal matches have been extracted to a separate data types:

```ocaml
type goal_match = ... | Gvar of var_name | ...
```

```ocaml
type name_match = ... | Nvar of var_name | ...
```

instead of just `type prql = ... | Var of var_name | ...`. In previous version, the `prql` data structure was flatter in the sense that it included the name and goal matching constructs as well (e.g. Atomic of prql instead of Atomic of name_match).

This made the abstract syntax tree of the query more structured and imposes more constraints on its shape. It also makes structures such as atomic somewhere atomic illegal on the level of types. In the previous implementation, the barrier against such queries was imposed only during parsing, while a more OCaml savvy user could feed such illegal queries freely into the query engine. The new solution is more robust, as it guards against such constructs both at the level of parser and at the level of OCaml’s types.

### 6.2.2 New lexer and parser

As discussed in Section 5.2.1, the hand crafted lexer and parser, implemented in Part 1, have been replaced with a automatically generated ones using ocamllex and ocamlyacc.

All that is necessary for `ocamllex` to generate a fully functional lexer is a list of regular expressions and tokens associated with them. The code that is generated by `ocamllex` is standalone and can be used without any additional modules.

The structure of the parser is more complex than the one of the lexer. `ocamlyacc` (which is short for OCaml yet another compiler compiler) compiles a file that expresses the grammar of the language into OCaml source code. The grammar is expressed as a list of rules and a description of how do they translate into a data structure (the abstract syntax tree) of the hiprql type as discussed in Section 6.2.1. The simplest cases are the single-token terminal structural queries, for example `iscut` or `isthen`:

```ocaml
query :
...
| ISTHEN { Isthen }
| ISCUT { Iscut }
...
```

The code above states that in the rule called `query` the `ISTHEN` token corresponds to the `Isthen` construct of the abstract syntax tree. A little more complex con-
structs are those for (simple) name matching:

\[
\text{simple\_name\_match}:
\begin{align*}
| & \text{VAR} \{ \text{Nvar}($1) \} \\
| & \text{IDENT} \{ \text{Nident}($1) \} \\
| & \text{WILDCARD} \{ \text{Nwild} \}
\end{align*}
\]

The simple name match corresponds to either a variable, a constant, or a wildcard. The above rule states that if the parser encounters a VAR token (e.g. \$A), it wraps it around the Nvar construct of type goal match.

As in case of the lexer, the parser that is generated by ocamlyacc is a standalone solution that can be shipped without any dependencies what makes it an ideal tool for this purpose as PrQL’s grammar is quite modest.

It is important to underline that the old and new parser are 100% compatible and all queries accepted by the old parser will be accepted by the new one. One subtle change is that the rich name matches (matching the names of parameters passed to a HOL Light tactics) no longer need to be inside single quotes.

### 6.2.3 Changes to the typechecker

The typechecker required wiring in the new six transformation queries and adapting to the changes introduced to the abstract syntax tree data structure. No other changes to the typechecker were performed except those required by introducing new queries and restoring its usability.

### 6.3 Improvements in the query engine

The changes in the lower parts of the project obviously triggered the necessity to refactor the upper-level modules to use the new data structure. This required to rewrite whole matching and satisfaction module and obviously the tests for these parts of code. Nonetheless, apart from these more technical issues, some meaningful improvements have been introduced to the query engine as well.

#### 6.3.1 Making queries native

In the original implementation, the derived forms as presented in Figure D.1, where desugared back into the basic queries before being transformed into the negation normal form and finally fed into the query satisfaction engine.

However, desugaring queries into the basic forms meant that more operations had to be performed during the query satisfaction phase. For example, the axiom \( nm \) derived form transforms to atomic \( nm \land \text{outgoals} \[ \] \), which in the abstract
syntax tree is expressed as \textbf{And} [Atomic \textit{nm}; \textbf{Outgoals} [ []]]. This means that a range of additional functions were necessary to be invoked to perform such a query compared to a situation when this derived form would have been implemented as a \textit{native} query. The same holds for some other derived forms, namely \textit{somewhere}, \textit{everywhere}, \textit{provesgoal}, \textit{somewherebeside}.

These derived forms have now been implemented as native. This has improved the overall performance of the system as discussed in Chapter 9.

6.4 Implementing the transformations

The same approach has been taken in this project for implementation of the five primitive hiproof operations: \textit{graft}, \textit{chop}, \textit{cover}, \textit{box}, and \textit{unbox}. They have been implemented within \textit{HipCam} on the general type \texttt{'a hiproof} and are parametrized later on in the \textit{hiproof transformation engine} which employs these five operations to implement the hiproof transformations.

6.4.1 Auxiliary graph data structure

\textit{PrQL} does not give its user the freedom to pick an arbitrary node to perform operations on or to extract information from. Rather, it makes the user \textit{describe} the node by the name of the tactic used and the ingoals to start with, but even this does not guarantee uniqueness. The \texttt{'a hiproof} data structure produced by \textit{HipCam} does not allow us to reference nodes of the hiproof as well.

Moreover, as discussed in \textit{Part 1} of the project, calculation of outgoals of a node using the \texttt{'a hiproof} data structure is quite an expensive operation.

These two issues have been solved in \textit{Part 1} of the project by an auxiliary data structure \texttt{'a prqlgraph}. This data structure keeps an unique identifier (an integer), the ingoals of every node in a hiproof, and the edges between nodes. Since it would be beneficial for the transformations to be able to reference nodes uniquely, this data structure is moved to the \textit{HipCam}’s newly created \texttt{Hitrans} module that will deal with the hiproof transformations and renamed to \texttt{'a himap}.

Moreover, the structure is expanded with two more constructs so that it can match the \texttt{'a hiproof} structure one-to-one. The resulting data type is presented in Figure 6.2.

The \texttt{hiedge} type augments the \texttt{'a hiproof} structure’s indirect expression of the $\rightarrow_s$ relation as discussed in Section 5.4.

The \texttt{Hitrans} module provides a function \texttt{himap_of : 'a hiproof $\rightarrow$ 'a himap} that computes a \texttt{himap} from a hiproof and a function \texttt{edges_of : 'a himap $\rightarrow$ 'a hi} that takes a \texttt{himap} and returns a full graph-like representation of the hiproof. These functions have been split from each other, as the process of computing the
Chapter 6. Implementation

```haskell
type node = int

type 'a himap =
    Map.atomic of node * int * 'a
  | Map.id of node * 'a
  | Map.cut of node * 'a
  | Map.var of node * 'a
  | Map.label of node * 'a himap * 'a
  | Map.tensor of node * 'a himap list
  | Map.sequence of node * 'a himap list


type hiedge = Map.edge of node * node

type 'a hi = Himap of 'a himap * hiedge list
```

Figure 6.2: Auxiliary data structure to store unique node identifiers and sequence and outgoals computation.

‘edges’ of a hiproof is quite computationally expensive and is not required for majority of the operations.

6.4.2 Operations on hiproofs

This section discusses the implementation details of the operations on hiproofs. All six operations are provided within the Hitans module that has been included into the HipCam hiproof recording software. Since in many cases these are quite subtle and minute pieces of code that actually have crucial meaning to the correctness of the operations, excerpts of code will be provided to argue that the operations are correct.

It is also crucial that the operations have been implemented on the general type 'a hiproof, rather than the (term list * term) hiproof which represents the hiproofs for HOL Light. This allows for testing the implementation on data of arbitrary types. To simplify testing, the functions are tested on int hiproof structures what corresponds to an integer being stored as an ingoal on each node.

The general idea behind each of the operation is to traverse the hiproof looking for specific structures. The hiproof is traversed alongside the corresponding himap either supplied or generated on the fly. The himaps are generated deterministically what assures that two himaps for a single hiproof are always identical.

6.4.2.1 Rename

The rename operation is the simplest operation, as it does not mutate the structure of the hiproof itself, rather changes the data on the nodes. The concrete rename function takes as an argument a hiproof, an identifier of the node to relabel and a new label.
let rec traverse hi_map =
    match (typecheck hi_map) with
      ( . . )
      | Hi_label(label, ingoal, child, info), Map_label(id', hi_, _) ->
        if id = id' then
          Hi_label(new_label, ingoal, traverse child hi_, info)
        else
          Hi_label(label, ingoal, traverse child hi_, info)
      | Hi_atomic(n, _, ingoal, info), Map_atomic(id', _, _) ->
        if id = id' then
          Hi_atomic(n, new_label, ingoal, info)
        else
          hi
      ( . . )

Figure 6.3: Excerpt from the rename function.

To argue the correctness of the function I would like to draw the attention to the code excerpt presented in Figure 6.3. On encountering a Hi_label or a Hi_atomic, the identifiers are compared and if they agree, the label is being replaced. In case of the Hi_label, there is also the case of traversing down the subhiproof within that node.

The idea behind the rename function is to traverse the whole hiproof and on encountering a node with the supplied identifier, the label is being replaced. However, by default, the function does not terminate after one occurrence to allow for multiple relabellings at a time.

The rename function takes the himap of the hiproof as an optional argument. If the himap is not supplied, one is automatically generated using the himap_of function. The reason behind this is that it may be more efficient in some cases to relabel a number of nodes in one go when the new label is the same for all of them. For example, a himap may be created such that all nodes to be relabelled are identified by 1, and 0 otherwise.

6.4.2.2 Unbox

The unbox operation is a little more complex than the replace, however the idea here is similar. The hiproof is traversed and on encountering a nested node with the supplied identifier, the node is ‘unnested’.

The characteristic of the ‘a hiproof that allows us to do simply that is the fact that all nodes that are within the box are represented as a single subhiproof. As only the layer of hierarchy is removed, the child ordering within the subhiproof is preserved hence the overall structure of the hiproof remains valid.

The unbox operation has been implemented as a single function unbox. The most crucial part of the unbox function is presented in Figure 6.4.
let rec traverse hi map =
  match (typecheck hi map) with
  (.)
  | Hi_label(label, ingoal, child, info), Map_label(id', map, .) ->
    let child, map = traverse child map in
    if id' = id then
      (child, map)
    else
      (Hi_label(label, ingoal, child, info),
       Map_label(id', map, ingoal))
  (.)

Figure 6.4: Excerpt from the unbox function.

On encountering a Hi_label which is equivalent to a nested node, the subhiproof within the hiproof is traversed recursively to unbox any nodes within as necessary. Then, the identifier on the nested node is compared to the supplied one and if they agree, the function returns only the result of recursively unboxing the inner subhiproof, hence unboxing it. Otherwise, the traversed subhiproof is wrapped back into its box.

As unboxing conserves the left-to-right child ordering, the operation is guaranteed not to alter the overall structure of the hiproof.

6.4.2.3 Graft

The graft operation is an inverse to chop and cover. Given any hiproof $H$ that contains a dangler, the graft operation literally ‘grafts’ another hiproof $H'$ on a relevant dangling node in $H$. It is important that the ingoal of $H'$ (e.g. the goal it proves) is the same that the goal on the dangling node.

The graft operation is implemented by the higraft function of the Hitrans module. The higraft function traverses the whole hiproof and on encountering a dangler with a matching node identifier and ingoals, replaces the dangler with the supplied subhiproof.

6.4.2.4 Chop and Cover

The cover($H, v$) operation is supposed to return all nodes within and below a certain node $v$ within the hiproof $H$. To achieve this within the current 'a hiproof structure, the whole hiproof needs to be inspected: as discussed in Section 6.4.1, this data structure indirectly uses the $\preceq$ child ordering relation to convey the notion of sequencing.

To compute a cover of a node, the function needs to traverse the data structure until it finds the node in question. Each node has one single incoming goal and
may have any number of *outgoing goals*. The cover function keeps track of the connections between the nodes using a list of node identifiers. When a node \( v \) with \( n \) *outgoals* is inspected, it is assumed that it consumes a single (first) element of the list. Subsequently, the \( v \) node identifier is inserted \( n \) times into the aforementioned list of nodes. The identifiers can be inserted at the beginning of the list, at the end of the list, or discarded completely, depending on the node that is visited. The cover function needs also to keep track of such a list of node identifiers which constitute the cover: if the node \( v' \) consumes *ingoal* of a node that is in the cover, that node \( v' \) is a part of the cover as well. The cover function should collect all such node and output them, diposing of the rest of the hiproof. It is important to traverse the hiproof in a certain direction only due to the left-to-right child ordering imposed on the nodes.

To get the result of the *chop* operation is to do the opposite: to remove these nodes that belong to the cover and replace the root node with a dangler \((\text{Hi\_cut})\).

It is important to notice that the majority of the transformation queries require both parts of the hiproof to process a query. Since the procedures to compute cover and chop are very similar in their nature, these two operations have been implemented as a single function \( \text{hisplit} \) that returns a tuple containing a cover of a node and the original hiproof without the cover: \( \text{hisplit}(H, v) = (\text{cover}(H, v), \text{chop}(H, v)) \).

### 6.4.2.5 Box

The \( \text{addboxat}(H, l, v, v_1, \ldots, v_n) \) operation is the most complex one as it involves chopping off and grafting back multiple subhiproofs. The aim of the operation is to draw a labelled box around a certain portion of a hiproof to introduce hierarchy. First of all, the box is bounded from above by a single root node \( v \). To achieve this, the hiproof is split using the \( \text{hisplit} \) function into the cover of node \( v \) and the remaining part of the hiproof. The box is also bounded from below and the list \( v_1, \ldots, v_n \) represents the border nodes whose covers are not supposed to be included in the box. To apply the boundary from below, covers of these nodes need to be chopped off as well.

There is a slight technicality in composing such a decomposed hiproof back to its complete form. If the box is drawn around the subhiproof without the covers of the border nodes, the covers would be pulled into the box on grafting them back on the subhiproof. This would obviously defy the purpose of cutting them off from the very beginning. To avoid that, all danglers need to be removed off the subhiproof before it can be put into a box. After the box is drawn around the resultant subhiproof, the danglers can be added to it and subsequently, the covers of the border nodes can be grafted back on the subhiproof. Finally, the subhiproof can be grafted back on the rest of the hiproof.

The \( \text{hibox} \) function supplied as a part of the \( \text{Hitrans} \) module behaves as \( \text{addboxat} \).
6.4.2.6 Further considerations of correctness

It is assumed that the hiproofs recorded using HipCam are valid. Moreover, HipCam exposes an API that allows for constructive operations on hiforests such as sequencing, tensoring, labelling, and creation of atomic boxes. Section 5.5 discusses the behaviour of these functions at length.

This implementation relies heavily on the API to reconstruct a hiproof in operations such as unboxing or hisplit. Moreover, in cases when the hiproof is being modified structurally, its himap is being modified in the same way as well, what required implementation of himap duals for sequence, tensor and so on. This allows to chain result of operations by running the structural query once, and then updating the hiproof iteratively by applying the operations step by step.

As the above excerpts from the code show, each case-matching is performed on an invocation of a typecheck function, match (typecheck hi map) with instead match hi, map with. The typecheck function compares the immediate structure of the hiproof and its himap and allows the operation to continue only in cases when they agree. For example, if the himap describes a sequence that is longer than the hiproof, this means that there has been some error in the operation performed on that hiproof since the structures do not match and the operation is aborted. This is a way for the implementation to self-check its correctness at a very basic level. This obviously arises from the confidence in the HipCam implementation, but also allows the two implementations to contribute to the general confidence in the correctness of the implementation.

6.5 Hiproof transformation engine

The operations on hiproofs implemented in the Hitrans module form the basis for the implementation of hiproof transformations according to their meaning as outlined in Section 4.3.4. Section 5.6 discussed the design decisions behind the interpretations of the transformations with respect to multiple possible substitution produced by the query satisfaction engine.

For any PrQL query u where q, the query satisfaction engine produces a set of possible substitutions for the structural clause q and feeds the result to the transformation engine that performs the transformation described by the transformation clause u according to the discussion in Section 5.6. If the result of the structural clause is empty, then the transformation engine returns no result as well, if there is a single possible substitution, then the transformation is applied and the result returned. When there is more than one possible substitution for the given q, then the user is warned about the ambiguity of the query and first of the substitutions is applied and the result returned.
Chapter 7

Evaluation

This chapter evaluates the implementation using different approaches, looking at both correctness and performance of the engine. Evidence towards correctness of the implementation will be established basing on a few examples presented below and by a test suite accompanying the implementation that will be discussed at length.

7.1 Using PrQL

This PrQL implementation comes as a module for HOL Light automated theorem prover and works within its environment. To start using PrQL, the OCaml’s REPL needs to be used. The ledit program may be used as a wrapper on the standard ocaml to take advantage of its more advance input manipulation capabilities. The #use "hol.ml";; statement will start HOL Light with PrQL and HipCam included. The expected output of the OCaml REPL is:

```
OCaml version 4.02.1

# #use "hol.ml";;
val hol.version : string = "2.20++"
(...)
PrQL 0.2 loaded
- : unit = ()
- : unit = ()
   Camlp5 parsing version 6.12
#
```

After the prompt #, the queries to PrQL may be issued using the Prqlengine.query and Prqlengine.query_and_export, with the former outputting the result back to the REPL, while the latter will export the result to the HTML format and display it in the browser.
**OCaml** is a functional language and as such treats all variables as immutable unless otherwise stated. The theorems in **HOL Light** are delivered as immutable variables as well and any **PrQL** query issued on a hiproof using the **query** or **query_and_export** functions will not alter the original hiproof, but will simply return the new hiproof as a result. To update a theorem with a new hiproof, the **query_and_save** function may be used in the following manner:

```let FACT_MONO = query_and_save "... in FACT_MONO where ..."
```

### 7.2 Examples of queries

Let us consider the theorem **FACT_MONO** that proves that the factorial function **fact** monotonically increases ∀n, m ∈ N. n ≤ m ⇒ fact(n) ≤ fact(m) that is a part of HOL Light’s extensive library of theorems. This particular theorem has been chosen due to its fairly small size, and interesting features such as the presence of nested boxing as well as lack of boxes on the right hand side. Figure 7.1 presents the original hiproof of **FACT_MONO** as recorded by **HipCam**. The **unbox** transformation may be used to remove hierarchy by unboxing a subhiproof. For example, the **Induction** box can be lifted using the following query:

```unbox $X in FACT_MONO where somewhere (islabel 'Induction' and at $X)
```

It is important to note that the actual nodes in the hiproof to perform the operation on are expressed using the **at** basic query. Figure 7.2 presents the result of

![Figure 7.1: Hiproof of FACT_MONO.](image-url)
7.2. Examples of queries

Figure 7.2: Unboxing Induction in FACT_MONO.

Having a fair confidence in the hierarchy removal transformation, the opposite may now be investigated. As said before, the right-hand-side of the hiproof is rather flat and does not exhibit any interesting hiproof features. This can be improved using PrQL. For example, let us consider the whole subhiproof under the SYM box to be a single logical entity, but excluding the cover of TRANS. The following query will draw a box around the cover of SYM excluding TRANS and its cover:

box $X$ to $W$ as 'Sym' in FACT_MONO where
  somewhere (* then (atomic 'SYM' and at $X$) then
  * then (somewhere atomic 'TRANS' and at $W$))

Figure 7.3 shows the outcome of the query from the point of view of the hiproof as a whole, while Figure 7.4 presents the box in more detail. From the point of view of the hiproof as a whole, the outcome is as expected. However, when looking more in detail inside the Sym box, one can notice that an additional identity box ID has appeared. This box, which is a graphical representation of the Hi_id construct, is used to preserve the child ordering among the child nodes of the EQ_MP box which has two child nodes, but only one of them is kept inside the box, while the second is exported outside of the box. The identity box allows to solve the issue of passing goals as discussed in Section 5.4.

Another query that has been implemented in this project is deletetree which simply removed a cover of a node. For example, the following query removes the
Figure 7.3: Partial boxing of the cover of \texttt{SYM} in \texttt{FACT\_MONO}.

Figure 7.4: Look inside of the newly created box for \texttt{SYM} in \texttt{FACT\_MONO}.
7.2. Examples of queries

cover of node SPEC of the FACT_MONO hiproof:

\[
\text{deletetree } \$X \text{ in } \text{FACT_MONO where somewhere atomic 'SPEC' and at } \$X
\]

The result of this query is shown in Figure 7.5. The proof recording module HipCam has been changed so that it presents the danglers as bullets \(•\) instead of full boxes to be more consistent with the papers. The cover of node SPEC has been removed from the hiproof and a dangler is placed in place of SPEC.

The only operation that does not influence the structure or hierarchy of a hiproof is rename. For example, for some reason we may found that the \(\text{MK}_0\text{COMB}\) which is an immediate child of \(\text{EQ}_0\text{MP}\) to be a ‘special’ case and want it renamed to \(\text{MK}_0\text{COMB Special}\). The following query achieves this result:

\[
\text{rename } \$Y \text{ as 'MK}_0\text{COMB Special' in } \text{FACT_MONO where somewhere } * \text{ then atomic 'EQ}_0\text{MP'} \text{ then} \\
(* \text{ beside ((atomic 'MK}_0\text{COMB' and at } \$Y) \text{ then } *))
\]

Result of this query is presented in Figure 7.6. The \(\text{MK}_0\text{COMB}\) node that is an immediate child of the \(\text{EQ}_0\text{MP}\) has been renamed, but the three other remained.

Another two queries are similar in their inner workings - these are replace and refine. Both of them replace a subhiproof with another, but in case of replace,
this is done within a single hiproof, while \texttt{refine} grafts an \textit{external} hiproof on a dangler.

The \texttt{replace} query allows to replace one subhiproof with another given that their ingoals are equal. In most cases, such a query can be used to optimize a hiproof where certain part of the proof is ‘useless’ - it does not bring anything meaningful to the proof as a whole. It would be quite laborous to look for theorems within \textit{HOL Light}’s library that exhibit such properties. Hence, the \texttt{FACT\_MONO} hiproof will be edited to include an identity tactic \texttt{ID} that would ‘do nothing’ except from passing the ingoal as the outgoal. Figure 7.7 shows the \texttt{ID} box inserted between \texttt{SPEC} and the \textit{Induction} nested box\footnote{As HipCam disregards any boxes with equal ingoals and outgoals by default, the feature that does that had to be turned off to create Figure 7.7. Please disregard the two \texttt{INSTANTIATE\_ALL} boxes at the very top, as they appear in every \textit{HOL Light} proof. They act as identity boxes from the point of view of the goal, but are important for the inner workings of \textit{HOL Light}. Since they act as identity boxes, they are hidden by default.}. The following query can remove the useless \texttt{ID} box:

\begin{verbatim}
replace $X$ by $Y$ in FACT\_MONO where
  somewhere * then (ingoals $G$ and atomic 'ID' and at $X$)
  then (ingoals $G$ and at $Y$)
\end{verbatim}

Figure 7.8 presents the result of the query. The \texttt{ID} box is removed and the sub-hiproof that followed was grafted back onto the original hiproof. It is important to note, that the redundant part removed by the \texttt{replace} query can be arbitrarily large and come from any other part of the hiproof.
7.2. Examples of queries

Figure 7.7: FACT_MONO with an additional ID box.

Figure 7.8: FACT_MONO with the ID box removed.
Chapter 7. Evaluation

The last hiproof transformation implemented in this project is the `refine $X$ with $s$` which takes an external hiproof $s$ and grafts it on a dangler $X$ within the original hiproof. First of all, let us transform the `FACT_MONO` hiproof to actually have a dangler by removing the cover of `SPEC`:

```
deletetree $X$ in `FACT_MONO` where
    somewhere atomic 'SPEC' and at $X$
```

This query has been discussed above and Figure 7.5 shows the result of the query. On the side, a part of the `FACT_MONO` theorem that proves the ingoal of `SPEC` is extracted as the `FACT_SUM` theorem. The following query grafts the hiproof of `FACT_SUM` back on a dangler in `FACT_MONO`:

```
refine $X$ with `FACT_SUM` in `FACT_MONO` where
    somewhere iscut and at $X$
```

The above query employs the non-standard query `iscut` to locate the dangler. Since there is only one dangler in the whole hiproof, there is no need to disambiguate where exactly `FACT_SUM` needs to be grafted. Figure 7.9 presents the result of the query. The newly grafted subhiproof has a box `lemma` around it, which is added for any single `HOL Light` theorem created using the `prove` method. It can be removed using a relevant query.
7.3 Tests for the hiproof operations

A hand crafted test suite has been created for each of the implemented hiproof operations: rename, unbox, higraft, hisplit, and hibox. They may be found in the hiproofs/test directory.

For the sake of these tests, a number of small examples of hiproofs have been prepared in the example_hiproofs.ml file in the test directory. These hiproofs are of type int hiproof, which means that the ingoals and outgoals in these hiproofs are numbers. The decision to use integers instead of the term list * term type that is used for HOL Light’s hiproofs has been made to simplify the creation of test cases.

The simplicity arises from the fact that integers can be compared using the OCaml built-in equality (Pervasives.(=)), while the objects of HOL Light type term can only be compared using α-equivalence function aconv. Moreover, the use of integers allowed to add an additional meaning to the values on the nodes: each ingoal of a node acts also as the unique identifier of that node.

This test suite can be ran by the following command:
#  #use "hiproofs/test/test.ml";;

7.4 Automated test cases generation

Given that each transformation query has a well-defined effect on a hiproof, it is possible to inspect the result of such a query and unequivocally decide if the transformation has been applied correctly using another query. For example, if unboxing a nested node, a PrQL query can be issued to check if the box has been indeed unboxed. Creating test cases manually may be a repetetive and mundane activity. One can go one step further, and create such test cases automatically.

For this project, automated tests have been created for the four transformations out of six: box, unbox, deletetree, and rename. It would be almost impossible to create automated tests for refine, as it requires an external hiproof with a matching ingoal, and also not every proof will have any redundancies removable by replace.

All tests were initially ran against 148 hiproofs generated from the theorems in HOL Light’s extensive library. The hiproof for LT_EXP has been removed from the test suite due to an issue discussed in Section 7.4.1.1 that could not be resolved because of the shape of the hiproof.

The test suite for the PrQL implementation as a whole can be ran by invoking the following command:
#  #use "prql/test/test.ml";;
7.4.1 Unbox

The high-level idea behind automated testing of the unbox transformation is to generate an unboxing query for every box in a hiproof and then inspect the hiproof to verify if that box has been removed. Since labels that are put on the boxes are not unique, it is important to have a way of disambiguating which particular box should be unboxed. To address this issue, an auxiliary module (Prqlunique) has been written that traverses a hiproof and generates queries uniquely identifying each box.

The automated test for the unbox transformation extracts all existing nested nodes within a hiproof using a PrQL query. Subsequently, the uniquely identifying structural clause $q$ is generated for each box. The clause is used first to unbox particular node using a query unbox $X$ where $q$, and then another query select $X$ where $q$ is issued to check if the box has been unboxed.

Out of 148 hiproofs that are included in the test suite, the test passes on 147 of them. The automated test for unbox fails for the LT_EXP theorem that has been removed from the test suite as discussed in Section 7.4.1.1.

The automated tests for unbox may be found in file prql/test/unbox.ml.

7.4.1.1 Automated test failing for LT_EXP

The automated unbox test fails for the LT_EXP theorem due to the structure of the hiproof because of two Repeat boxes being sequenced one after the other. The exact part that breaks the automated test is shown in Figure 7.10. The box labelled as Repeat consist only of a single identity box, and is sequenced with another Repeat box.

The issue arises when the first Repeat box is unboxed. The sequence function used to reconstruct the hiproof removes any identity boxes as discussed in Section 5.5. Hence the identity box that is the only node within the first Repeat box is removed from the sequence. Consecutively, the second Repeat box takes place of the first Repeat box. When the select $X$ where $q$ is issued, a box labelled Repeat is found at the place of the previously removed one, hence the test fails.

The same situation could arise in a situation when a box had another box with the same label as its immediate child.

7.4.2 Box

The approach to automating tests for the box transformation is slightly different. The test issues a PrQL query that is looking for a sequence of three atomic nodes and boxes first two of them, verifying the result by issuing another PrQL query to check that the box has been indeed added. This test covers only a subset of possible scenarios, however since it passes for all 147 hiproofs in the test suite (and
7.4. Automated test cases generation

Figure 7.10: A case that breaks the automated testing of unbox.

also for the two removed), this gives a certain level of assurance. The automated tests for box may be found in file prql/test/box.ml.

7.4.3 Delete tree

The approach to the deletetree transformation automated testing is similar to the one for unbox. The general idea here is to list all atomic nodes of a hiproof, iteratively remove their covers, and check if that particular node has been removed using a PrQL query.

Since the names of the atomic nodes are not unique, it is important to disambiguate the node that should have its cover removed. The Prqlunique module is used again to produce structural clauses $q$ that point to the given node. The node is removed using query deletetree $X$ where $q$ and the operation result is verified using query select * where $q$.

The test passes for 147 hiproof out of 147 in the test suite, however it initially timeouted for ODD_ADD in the standard OUnit setup. This had been caused by the fact that this hiproof consisted of 830 atomic nodes, and the second largest proof in the test suite, EVEN_ODD_DECOMPOSITION, that did not timeout, consisted of 482 such nodes only. Hence, the timeout of the test suite had to be adjusted to Long (600 seconds) instead of the default Short (60 seconds). The automated tests for deletetree may be found in file prql/test/deletetree.ml.
Chapter 7. Evaluation

7.4.4 Rename

The methodology behind the automated test cases generation for **rename** is similar to the approach taken for **deletetree**. For each atomic box in a hiproof, a query is issued to rename that box to `PRQL.RENAME.TEST` and then another query checks for presence of that label in the hiproof.

For each atomic box $A$ at position $X$, a uniquely identifying structural clause $q$ is issued, and the generated transformation query takes form `rename $X$ as 'PRQL.RENAME.TEST' where q`. The result is validated using a general query `select * where somewhere atomic 'PRQL.RENAME.TEST'`.

If the verifying query succeeds it should return `[[ ]]`, which is interpreted as `true`, and `Empty_result` otherwise. For the test suite of 147 hiproofs, all test cases generated for 147 hiproofs pass. The automated tests for **rename** may be found in file `prql/test/rename.ml`.

7.5 Performance improvements

As significant changes have been introduced to the PrQL codebase, it would be beneficial to compare performance of the versions 0.1 (old) and 0.2 (new) with respect to the subset of functionality delivered by both projects. A benchmark consisting of eight different queries has been prepared (file `prql/test/benchmarks/`
queries.ml) and each query has been evaluated on 148 different proofs taken from HOL Light’s library. The experiment was ran with the hiproof for LT_EXP included in the test suite.

Figure 7.11 presented the time taken by each implementation to perform given query (Roman numerals) on all 148 proofs. The most visible result is that the refactorings performed in the second part of the project have improved the overall performance of the implementation. This can be caused mainly by the fact that many derived queries (queries that can be defined by other queries) have been implemented natively instead of desugaring them back to the basic queries. As discussed earlier, this approach, although theoretically correct, caused a significant overhead in terms of function calls. The second query, which sees the largest improvement, is looking for a list of all ingoals of the hiproof (somewhere ingoals $G$). The main increase in performance should come from the somewhere derived query being implemented natively, which not only saves on function calls but also on comparison between different values for $G$.

The above comparison of the two implementations is only quantitative, as it compares only the performance, not the equivalence of the results. Performing a qualitative comparison to check if the results are identical would be a very complicated operation, as the format of the output has changed (e.g., the introduction of Empty_result value instead of just []) and the ordering of the rows within a single answer may vary as well. Moreover, two instances of HOL Light running within a single OCaml REPL instance would duplicate the datatypes and make them uncomparable.

I argue that the hand-crafted tests that have been written as a part of the verification process can give a good base to assume that the two implementations yield identical answers to the queries.
Chapter 8

Conclusion

8.1 Summary

The main deliverable of this project is a fully functional implementation of the hiproof transformation subset of PrQL. The implementation is complete and has been found correct to a significant degree as shown by the extensive test suite and to the best of my knowledge. Nonetheless, this does not rule out any possible bugs that could be found during a more extensive period of use.

The unique contribution of this project is not only the implementation, but also the theoretical work that supports it. The type system, introduced in the first part of the project, has been extended to cover the transformation clauses. Moreover, a thorough discussion of possible approaches to the interpretation of the transformation clauses with respect to the outcome of the structural clause, not only to a single set of substitution, has been presented in Section 5.6.

Additionally, the query satisfaction engine implemented in part 1 of the project has been revisited, evaluated, and partially reimplemented what caused a significant speedup of this part of the implementation. The custom lexer and parser, delivered in part 1 of the project, have been replaced by a solution based on ocamlllex and ocamllyacc. The internal representation of the query, the abstract syntax tree, has also been revisited and significantly improved what caused the need to propagate these changes across all modules of the implementation, including the tests. Moreover, the derived queries of the structural clause have been implemented natively what also contributed to the overall performance improvement. It is important to note that the grammar and semantics of the queries have not been impacted in any way.

A significant amount of work has been devoted to verify the implementation through tests, including hand-crafted unit test and automated tests for deletetree, box, and unbox.

Moreover, the work to make the theorem prover-agnostics PrQL more native to HOL Light has been continued from the previous part of the project, by
introducing the iscut query. The HipCam hiproof recording module has also been improved to present danglers in a more visually appaling manner, by \bullet, instead of a full box.

The work on the project has also uncovered a typographical error in the original paper that introduced the transformation subset of PrQL [2]. The error has been brought to the attention of and discussed with one of the authors and the corrected version has been presented and used in this report.

8.2 Extension of the project

Although the test suite delivered as a part of this project is quite thorough what allowed to discover and fix some bugs as the project progressed, the project could benefit from even more thorough testing, especially for some corner cases that have not been identified during the project.

Formal verification can be employed as a way of establishing correctness of the implementation. As discussed in Chapter 2.2.1, HOL Light is a tool that may facilitate formal verification of software and there is no better opportunity to perform formal verification than when working within a theorem prover.

Although the current implementation of the query satisfaction engine is way faster than the original one, as shown in Section 7.5, it is still not as performant as it could be. When experimenting on larger proofs, such as BALLOT from the HOL Light’s library, the query satisfaction engine experiences a blow in memory usage (up to 10 GB) without producing any results. Therefore, it would be beneficial for the project to use an external system, for example a graph database, to answer the queries and perform the operations.
Appendix A

**HOL Light**’s interactive mode

Section 2.2.1 presented a proof of a simple theorem, that for the defined data type $expr$ representing an *abstract syntax tree* and the $eval$ function that evaluates the AST to a numerical value, it is true that

$$\forall n, m. \text{eval} (\text{Add} (\text{Num} n) (\text{Num} m)) = n + m.$$  

The proof has been presented in the declarative way by a sequence of tactics and tacticals. However, **HOL Light** allows to construct proofs *interactively* by trial and error. This appendix aims at showing how to construct a proof using the interactive mode by proving the same theorem once again.

The g function allows to state the *goal* of the proof, that will be deconstructed to prove the statement in the backward style. The e function on the other hand, allows to mutate the goal by applying a tactic or a tactical.

```
# g "! n m. eval (Add (Num n) (Num m)) = n + m" ;
val it : goalstack = 1 subgoal (1 total)
```

```
`!n m. eval (Add (Num n) (Num m)) = n + m`
```

The **INDUCT_TAC** tactic splits the single goal into two separate goals: the *Base Case* and the *Inductive Case*, to facilitate the *proof by induction*.

```
# e INDUCT_TAC ;;
val it : goalstack = 2 subgoals (2 total)
```

```
0 [ `!m. eval (Add (Num n) (Num m)) = n + m` ]
```

```
`!m. eval (Add (Num (SUC n)) (Num m)) = SUC n + m`
```

```
`!m. eval (Add (Num 0) (Num m)) = 0 + m`
```

```
# e INDUCT_TAC ;;
val it : goalstack = 2 subgoals (3 total)
```

67
0 [`eval (Add (Num 0) (Num m)) = 0 + m`]
`
`eval (Add (Num 0) (Num (SUC m))) = 0 + SUC m`
`
`eval (Add (Num 0) (Num 0)) = 0 + 0`

The \texttt{REWRITE\_TAC} is a parametrized tactic that rewrites the goal using a set of theorems. In this case, we may want the \texttt{REWRITE\_TAC} to simply \textit{unfold} the definition of the \texttt{eval} function:

\begin{verbatim}
# e (REWRITE\_TAC[EVAL]);;
val it : goalstack = 1 subgoal (2 total)
  0 [`eval (Add (Num 0) (Num m)) = 0 + m`]
`
`eval (Add (Num 0) (Num (SUC m))) = 0 + SUC m`

# e (REWRITE\_TAC[EVAL]);;
val it : goalstack = 1 subgoal (1 total)
  0 [`!m. eval (Add (Num n) (Num m)) = n + m`]
`
!m. eval (Add (Num (SUC n)) (Num m)) = SUC n + m`

# e (REWRITE\_TAC[EVAL]);;
val it : goalstack = No subgoals

# let EVAL\_SUM = top_thm ();;

val EVAL\_SUM :
  thm = |− !n m. eval (Add (Num n) (Num m)) = n + m
\end{verbatim}

It is important to note that the proof constructed using the \textit{interactive mode} will differ from the one constructed using the \texttt{prove} function, as in the interactive mode the goals are being dealt with on a case-by-case basis, while the \texttt{prove} function will attempt to apply a tactic to as many goals as possible:

\begin{verbatim}
# let EVAL\_SUM =
  prove (`! n m. eval (Add (Num n) (Num m)) = n + m`,
  INDUCT\_TAC THEN INDUCT\_TAC THEN REWRITE\_TAC[EVAL]);;

val EVAL\_SUM :
  thm = |− !n m. eval (Add (Num n) (Num m)) = n + m
\end{verbatim}
Appendix B

Formal definition of a hiproof

Aspinall et al. in [2] propose a more detailed definition of the hiproof than the one derived in Section 3.1. They introduce the notions of an ordered hiproof, which is a hiproof with the child ordering relation, and an ordered hiforest that is a more relaxed form of the ordered hiproof.

The reasoning behind the introduction of the child ordering relation is that the premises of inference rules in the underlying derivational systems have an ordering.

An ordered hiforest is a structure $H = \langle V, L, \leq_i, \rightarrow_s, \lesssim \rangle$ that consists of:

- $V$, a finite set of vertices,
- $L$, the labelling function $V \rightarrow (\mathcal{L} \cup \{\bullet\}) \times \mathcal{G}$ mapping the vertices to their labels and the incoming goals,
- $\leq_i$, the inclusion relation on $V \times V$ ($v \leq_i w$ reads as $v$ inside $w$), or the proper containment relation $>_i$,
- $\rightarrow_s$, the relation on $V \times V$ that describes the functional composition of nodes - the sequence in which they appear in the graph,
- $\lesssim$, the child ordering relation on $V \times V$ that partially orders nodes left-to-right (including the roots).

Section 3.1 introduced four constraints that have been imposed on the hiproof structure originally in [5]. In [2], the same constraints have been formulated as eight separate conditions, seven of which refer to the hiforest and the final, eighth one, concerning the uniqueness of the root, is reserved for the hiproof. The conditions below elaborate on the subtleties not immediately obvious from the original four constraints and introduce the child ordering relation. The constraints that need to be satisfied by the ordered hiforest are as follows:

0. $\langle V, \leq_i \rangle$ and $\langle V, \rightarrow_s \rangle$ each form a forest; $\leq_i$ and $\lesssim$ are partial orders on $V \times V$. 

69
1. arrows target outer nodes: \( v \rightarrow_s w \land w \leq_i v' \Rightarrow v \leq_i v' \) (if \( v \) follows \( w \), and \( w \) is inside of \( v' \), then \( v \) must be inside of \( v' \)), which means that if an arrow is targeted at a node \( w \) that is inside another node \( v' \), the source node \( v \) of that arrow must be in the same box \( v' \) as the target \( w \) (otherwise it would have targeted the outer node \( v' \)).

2. arrows emanate from inner nodes: \( v \rightarrow_s w \land v' \leq_i v \Rightarrow v = v' \), if a node \( v \) targets node \( w \), but there is another node \( v' \) within the source node \( v \), then \( v \) must be the same node as \( v' \).

3. inclusion and sequence are mutually exclusive: \( v \leq_i w \land v \rightarrow_s w \Rightarrow v = w \), if a node \( v \) is both followed by and contains a node \( w \), then this must be the same node.

4. boxes have unique roots:

\[
siblings_{\leq_i}(v, v') \land isroot_{\rightarrow_s}(v) \land isroot_{\rightarrow_s}(v') \Rightarrow v = v'
\]

(if two nodes are siblings and they are both roots with respect to the sequencing relation \( \rightarrow_s \) then they must be the same node).

5. children or top-level roots are totally ordered (but \( \leq \) is a partial order anyway since we cannot compare nodes at different "levels"):

\[
siblings_{\rightarrow_s}(v, v') \lor (isroot_{\rightarrow_s}(v) \land isroot_{\rightarrow_s}(v')) \Rightarrow v \leq v' \lor v' \leq v
\]

6. only leaves with respect to sequencing and inclusion may have the • label that symbolizes an open (unproven) goal:

\[
L(v) = (\bullet, \gamma) \Rightarrow isleaf_{\rightarrow_{\cup_i}}(v)
\]

An ordered hiproof is an ordered hiforest that satisfies an additional constraint:

7. Top level roots are unique: \( isroot_{\rightarrow_{\cup_i}}(v) \land isroot_{\rightarrow_{\cup_i}}(v') \Rightarrow v = v' \)

**B.0.1 Validity**

A hiproof is said to be valid when it corresponds to a real proof tree in the chosen derivational system. A valid hiproof can also be seen as the nested labelling applied on a proof tree [1].

The notion of validity can also be applied to hiforests: a hiforest is valid if it corresponds to a sequence of valid hiproofs [2].
Appendix C

Hiproof operations

This chapter discusses the technicalities behind hiproof operations as defined in [2]. The definitions are accompanied with a thorough explanation which is a unique contribution of this report.

C.1 Operations

C.1.1 Cover

A cover of an element with respect to a relation $R$ is defined as $cover_R(x) = \{ y \mid xR^*y \}$. Given a hiforest $H$ and a vertex $v \in V$, we define the cover of $v$ as all nodes below or inside $v$ by $V' = cover \rightarrow_{\cup \supset}(v)$. The resultant hiforest has the labelling and relations restricted to $V' \times V'$:

$$cover(H,v) = \langle V', L|_{V' \times \supset}, \leq_i |_{V' \times V'}, \rightarrow_s |_{V' \times V'}, \triangleq |_{V' \times V'} \rangle$$

C.1.2 Chop

Given a hiforest $H$ and vertex $v$, then the operation of chopping off the vertex $v$ creates a new hiforest $H'$ that is $H$ without the cover of $v$ (without nodes below or inside of $v$) by limiting the set of nodes $V'$ to $V' = (V - cover \rightarrow_{\cup \supset}(v)) \cup \{v\}$ and adjusting all relations to be closed in $V' \times V'$:

$$chop(H,v) = \langle V', L|_{V' - cover \rightarrow_{\cup \supset}(v)} \cup \{v \mapsto (\ast, \gamma) \mid L(v) = (l, \gamma)\}, \leq_i |_{V' \times V'}, \rightarrow_s |_{V' \times V'}, \triangleq |_{V' \times V'} \rangle$$

The resulting hiforest has the cover of $v$ removed, but the node $v$ itself remains and is relabelled with $\ast$ signalling that it is an open (unproven) goal.
C.1.3 Graft

Let $H$ be a hiforest $H \models g_1 \rightarrow g$ mapping incoming goals $g_1$ to a set of unproven outgoing goals $g$. Let $\{v_1, ..., v_n\} \subset V$ be a set of dangling vertices of $H$ such that $\forall v_i. L(v_i) = (\bullet, \gamma_i)$, and $n \leq \text{length}(g)$. Let $H' = \langle V', L', \leq', \rightarrow', \preceq' \rangle$ be another hiforest with $H' \models g' \rightarrow g_2$ that has $n$ overall roots $\{v_{r_1}, ..., v_{r_n}\} \subset V'$ ordered by $\preceq'$, with $\forall v_{r_i}. L(v_{r_i}) = (l_i, \gamma_i)$. In other words, the set of ingoals of hiforest $H'$ is a subset of matching danglers of $H$. The graft operation can be described as:

$$\text{graft}(H, H', v_1, ..., v_n) = \langle V - \{v_1, ..., v_n\} \cup V', L|_{V - \{v_1, ..., v_n\}} \cup L', \leq'', \rightarrow'', \preceq'' \rangle$$

with the relations $\leq'', \rightarrow'', \preceq''$ being defined as:

C.1.3.1 Inclusion relation

$$v \leq'' w \iff \begin{cases} v \leq_i w \land w \notin \{v_1, ..., v_n\} \\ v \leq_i v_{r_i} \land v \leq_i w \\ v \leq_i' w \end{cases}$$

This means that $w$ contains $v$ in the resultant hiproof if:

- $v$ is inside $w$ in hiforest $H$ and $w$ is not a dangler (danglers get removed from $H$ in the process of grafting),
- $v$ is within one of the root nodes of $H'$ and the matching dangler in $H$ was inside $w$,
- $v$ is inside of $w$ within $H'$.

C.1.3.2 Functional composition relation

$$v \rightarrow'' w \iff \begin{cases} v \rightarrow_s w \land w \notin \{v_1, ..., v_n\} \\ v \rightarrow_s v_{r_i} \rightarrow' w \\ v \rightarrow'_s w \end{cases}$$

In the resultant hiforest $v \rightarrow'' w$ iff:

- $v$ targets $w$ in $H$ and $w$ is not a dangler since danglers are removed,
- $v$ targets a dangler $v_{r_i}$ in $H$ and the matching root $v_{r_i}$ in $H'$ targets $w$,
- $v$ targets $w$ within $H'$.

C.1.3.3 Child ordering relation

$$v \preceq'' w \iff \begin{cases} v \preceq w \land w \notin \{v_1, ..., v_n\} \\ (v \preceq v_{r_i} = w) \lor (v = v_{r_i} \land v_i \preceq w) \\ v \preceq'_i w \end{cases}$$

This means that $w$ targets $v$ in the resultant hiproof if:

- $v$ targets $w$ in $H$ and $w$ is not a dangler since danglers are removed,
- $v$ targets a dangler $v_{r_i}$ in $H$ and the matching root $v_{r_i}$ in $H'$ targets $w$,
- $v$ targets $w$ within $H'$. 
The child order is updated accordingly so that \( v \precsim'' w \) iff:

- \( v \) is left of \( w \) in \( H \) and \( w \) is not a dangler,
- \( v \) is left of a dangler \( v_i \) in \( H \) and \( w \) is a root in \( H' \) that corresponds to that dangler,
- \( v \) is a root in \( H' \) and its corresponding dangler \( v \) in \( H \) is left of \( w \) (also in \( H \)),
- \( v \) is left of \( w \) in \( H' \).

The above definition has been fixed as an error has been introduced in the original paper as discussed in Section 3.5.

C.1.4 At

The operations above are defined on a hiforest as a whole. To allow transforming specific subhiforests, one would have to chop a cover of a chosen vertex, apply the transformation, and graft it again. The \( \text{at} \) combinator is introduced as a shorthand for applying a transformation function \( f \) on a subhiforest:

\[
\text{at}(H, v, f) = \text{graft}(\text{chop}(H, v), f(\text{cover}(H, v)))
\]

C.1.5 Box

Let \( v_r \) be the root of a valid hiproof \( H \). Let \( l \) be the label to be put on the hiproof \( H \), then:

\[
\text{box}(H, l) = (V \cup \{\star\}, L \cup \{\star \mapsto (l, \gamma) \mid L(v_r) = (l', \gamma)\},
\]

\[
\leq_i \cup \{(v, \star) \mid v \in V, L(v) = (l, \gamma) \land l \neq \bullet\}, \rightarrow_s, \leq \cup \{(\star, \star)\}
\]

The \( \text{box} \) operation adds an auxiliary node \( \star \) \((V \cup \{\star\})\), that has the same set of incoming goals as the root of the hiproof \( v_r \), but a different label \( l \) \((L \cup \{\star \mapsto (l, \gamma) \mid L(v_r) = (l', \gamma)\})\). The new node \( \star \) is also the parent of all nodes in \( V \) with respect to the inclusion relation \((\leq_i \cup \{(v, \star) \mid v \in V\})\) except the dangling nodes \((L(v) = (l, \gamma) \land l \neq \bullet)\).

What is important to note is that the dangling nodes are left outside of the box. This behaviour will be crucial in defining the \( \text{addboxat} \) operation in Section (..).

C.1.6 Unbox

Given a hiproof \( H \) with an overall root \( v_r \), let \( V' \) be the set of vertices of the unboxed hiproof as follows:

\[
V' = \begin{cases} 
  V - \{v_r\} & \text{isroot} \rightarrow_{s \cup >}, L(v) = (l, \gamma) \land l \neq \bullet \\
  V & \text{otherwise}
\end{cases}
\]
Then the operation of unboxing the hiproof $H$ is defined as follows:

$$\text{unbox}(H) = (V', L|_{V'}, \leq_1 |_{V' \times V'}, \rightarrow_s |_{V' \times V'}, \preceq |_{V'})$$

The overall root $v_r$ of the original hiproof is removed from the set of vertices only if it is not a dangler. If the $v_r$ node is a dangler, it means that $V = \{v_r\}$ and removing the dangler would make the hiproof empty and not valid at all (empty hiproofs are not valid). Then, all sets and relations for the original hiproof $H$ are restricted to $V'$ and $V' \times V'$ only.
Appendix D

PrQL 12 details

D.1 Derived forms

Building on the basic queries, the authors proposed also a set of derived forms which are useful shorthands for repeatable queries. These derived forms are presented in Figure D.1. These have been discussed at length in the Part 1 report and since they are not an object of interest in query transformation, their further discussion is omitted on purpose.

D.2 Name and goal matching

The \( nm \) and \( gm \) in Figure 4.1 stand for name matches and goal matches, which are patterns that can be used to query for labels on the hiproof’s nodes. Figure D.2 presents the acceptable forms the name and goal matches can take.

As discussed in Section 3.3.2, a label on a node consists of a tuple \( L(v) = (l, \gamma) \), where \( l \) is the name of the tactic or subtactic and \( \gamma \) is the set of input goals of the given node.

The name matches, which are used in basic queries such as atomic and inside, allow for inspecting the names of the tactics \( (l) \). A name match can be a:

- **constant** \( (l) \) any possible string representing a name of a tactic,
- **any value** \( (*) \) any possible value that a name can take (names of all possible atomics and labelled boxes),
- **a predicate** \( (\xi) \) a logic-dependent predicate on names (\( \text{Induction}(T) \) will return all variables that are being inducted on, in this case \( x \)),
- **name variables** \( (N) \) a set of variables standing for names (e.g. \( \$A \)),
- **negation** a complement of a match (e.g. \( \neg A \))[18].
somewhere \( q := \mu Q.q \lor (\text{inside } * Q) \lor (Q \text{ then } *) \lor (* \text{ then } Q) \lor (Q \text{ beside } *) \lor (* \text{ beside } Q) \)  

everywhere \( q := \mu Q.q \land (\text{atomic } * \lor \text{nothing} \lor (\text{inside } * Q)) \land (Q \text{ then } Q) \land (Q \text{ beside } Q) \)  

\[ q_1 \text{ when } q_2 := q_1 \lor \neg q_2 \]

\[ \text{isthen } := * \text{ then } * \]

\[ \text{isbeside } := * \text{ beside } * \]

\[ \text{provesgoal } \gamma := \text{ingoals } [\gamma ] \land \text{outgoals } [] \]

\[ \text{axiom } nm := \text{atomic } nm \land \text{outgoals } [] \]

\[ \text{islabel } nm := \text{inside } nm * \]

\[ \text{whenin } nm q := \text{inside } nm q \text{ when islabel } nm \]

\[ \text{somewherebeside } q := \mu Q.q \lor (Q \text{ beside } *) \lor (* \text{ beside } Q) \]

\[ \text{nearby } q_1 := \mu Q.q \lor (Q \text{ then } *) \lor (* \text{ then } Q) \lor (Q \text{ beside } *) \lor (Q \text{ beside } Q) \]

\[ \text{separately } q_1 \text{ and } q_2 := \text{somewhere } ((\text{somewhere } q_1 \text{ then somewhere } q_2) \lor (\text{somewhere } q_1 \text{ beside somewhere } q_2)) \]

**Figure D.1:** Derived forms for the *structural clause* [1].

\[
\begin{align*}
nm & := l | * | \xi | N | \neg nm \\
gm & := [\psi_1, ..., \psi_n] | G | \neg gm
\end{align*}
\]

**Figure D.2:** Matches in PrQL [1]

The *goal matches*, used in two basic queries, namely *ingoals* and *outgoals*, are matched against the goals on the label (\( \gamma \)) and can take a form of:

**a predicate** \( \psi_i \) a logic-dependent predicate on a goal (e.g. \( \phi_{\text{hornclause}}(\gamma) \)), including the goal itself as a constant (e.g. \( [0 + 1 > 0; S(k) + 1 > S(k)] \)),

**goal variables** \( G \) a set of variables standing for names (e.g. \( G \)),

**negation** a complement of a match \( \neg G \)[18].
Appendix E

PrQL type system

Figure E.1 shows the type checking rules while Figure E.2 presents the type inference rules for the *structural clauses* of PrQL queries. These rules have been introduced in *Part 1* of the project.

$$
\tau ::= \text{atomic} \mid \text{label} \mid \text{goal} \mid \text{goals} \mid \text{rec}
$$

$$
\frac{$V: \tau \in \Gamma$}{\Gamma \vdash {\$V: \tau}} \quad \frac{\Gamma \vdash A: \text{atomic}}{\Gamma \vdash \text{atomic } A} \quad \frac{\Gamma \vdash q}{\Gamma \vdash \neg q \text{ negI}}
$$

$$
\frac{\Gamma \vdash G: \text{goals}}{\Gamma \vdash \text{ingoals } G} \quad \frac{\Gamma \vdash G_1: \text{goal} \ldots \Gamma \vdash G_n: \text{goal}}{\Gamma \vdash \text{ingoals } [G_1, \ldots, G_n]}
$$

$$
\frac{\Gamma \vdash G: \text{goals}}{\Gamma \vdash \text{outgoals } G} \quad \frac{\Gamma \vdash G_1: \text{goal} \ldots \Gamma \vdash G_n: \text{goal}}{\Gamma \vdash \text{outgoals } [G_1, \ldots, G_n]}
$$

$$
\frac{\Gamma \vdash q_1 \quad \Gamma \vdash q_2}{\Gamma \vdash q_1 \text{ then } q_2} \quad \frac{\Gamma \vdash q_1 \quad \Gamma \vdash q_2}{\Gamma \vdash q_1 \text{ beside } q_2} \quad \frac{\Gamma \vdash q_1 \quad \Gamma \vdash q_2}{\Gamma \vdash q_1 \text{ and } q_2}
$$

$$
\frac{\Gamma \vdash q_1 \quad \Gamma \vdash q_2}{\Gamma \vdash q_1 \text{ or } q_2} \quad \frac{\Gamma \vdash q \quad \Gamma \vdash L: \text{label}}{\Gamma \vdash \text{inside } L q} \quad \frac{\Gamma \vdash \text{rec } \in \Gamma \quad \Gamma \vdash q \text{ recI}}{\Gamma \vdash \text{rec } Q.q} \quad \frac{\Gamma \vdash Q: \text{rec}}{\Gamma \vdash \text{recApp } Q}
$$

Figure E.1: Proposed PrQL type checking rules.
Appendix E. PrQL type system

\[ \tau ::= \text{atomic} \mid \text{label} \mid \text{goal} \mid \text{goals} \mid \text{rec} \]

\[
\frac{}{q \Rightarrow \Gamma}
\]

\[
\frac{}{-q \Rightarrow \Gamma \neg I}
\]

atomic \( A \Rightarrow A : \text{atomic} \)

inside \( L q \Rightarrow \Gamma, L : \text{label} \)

\[
\frac{q_1 \Rightarrow \Gamma_1 \quad \Gamma_1, q_2 \Rightarrow \Gamma}{q_1 \text{ then } q_2 \Rightarrow \Gamma}
\]

\[
\frac{q_1 \Rightarrow \Gamma_1 \quad \Gamma_1, q_2 \Rightarrow \Gamma}{q_1 \text{ beside } q_2 \Rightarrow \Gamma}
\]

\[
\frac{q_1 \Rightarrow \Gamma_1 \quad \Gamma_1, q_2 \Rightarrow \Gamma}{q_1 \text{ and } q_2 \Rightarrow \Gamma}
\]

\[
\frac{q_1 \Rightarrow \Gamma_1 \quad \Gamma_1, q_2 \Rightarrow \Gamma}{q_1 \text{ or } q_2 \Rightarrow \Gamma}
\]

ingoals \( G \Rightarrow G : \text{goals} \)

outgoals \( G \Rightarrow G : \text{goals} \)

ingoals \([G_1, ..., G_n] \Rightarrow G_1 : \text{goal}, ..., G_n : \text{goal} \)

outgoals \([G_1, ..., G_n] \Rightarrow G_1 : \text{goal}, ..., G_n : \text{goal} \)

\[
\frac{}{q \Rightarrow \Gamma \quad \frac{}{\Gamma, Q : \tau \not\in \Gamma \quad Q : \text{rec} \quad \text{recI}}}
\]

Figure E.2: Proposed type inference rules for PrQL.
Bibliography


