UNIVERSITY OF CALIFORNIA SAN DIEGO

Front-end tooling for building and maintaining dependently-typed functional programs

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Valentin Robert

Committee in charge:

Professor Sorin Lerner, Chair
Professor William Griswold
Professor James Hollan
Professor Ranjit Jhala
Professor Todd Millstein

2018

The Dissertation of Valentin Robert is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professor Sorin Lerner for his support as my advisor and the chair of my committee.

Committing oneself to more than half a decade to a mentally-challenging, ill-paying, self-motivated job seems like an awful idea. Yet, it is the ordeal that most graduate students choose, with little to no foresight as to where the journey will lead them. I would like to dedicate the next paragraphs to all past, current, and future graduate students.

Throughout my six years as a graduate student, I have only been able to visit my family for a total of three months, both my grandparents passed away after I was unable to attend a Christmas with them, nor was I able to attend their respective funeral. I have needed to move twice, I have needed to TA seven quarters, I have developed repetitive stress injuries in both my arms, and I have loaned over $10.000 to be able to bring this work to completion.

It was not without its worth though.

I have made a solid group of friends without which the journey would have been too much to bear. Zachary Tatlock brought me up to speed and offered me several great opportunities over the years. Alexander Bakst has been a great roommate, and it was always nice going for a bike ride to our local coffee shop for getting work done. Neha Chachra was a great neighbour, and helped me work through a lot of mental issues. Marcela Mendoza also provided mental support, and helped me focus on working hard for several years, as well as recovering from hard work sessions by being a great travel companion. Sohini Manna was helpful in yelling at me to get things done. Many recent new friends have made the final years of the journey wonderful, with lots of music and enjoyable hangouts. Dimitar Bounov accompanied me all the way to the end of the journey, as well as on many memorable hikes. In particular, my partner Andrea Frank provided a wonderful emotional support over the past year, and with her roommate, John Renner, have provided the hosting without which I would not have been financially able to complete this journey.

I would like to thank the rest of the UCSD Programming Systems group, who have provided a great environment for research. In particular, Eric Seidel has always been tremendous at listening to my issues and offering the perfect, appropriate technical pointers to resolve them. My work would often have been weeks behind without his help.

I would also like to thank the Gallium team of Inria, and in particular my former mentor, Xavier Leroy, for his great mentorship and the confidence he inspired in me. Gabriel Scherer, Jonathan Protzenko, and Nicolas Pouillard, among many others, have also been excellent lunch and coffee break companions, who opened my eyes to many of the great concepts of the programming language community. Working with all of them was a daily delight.

Andrew Kennedy, who accepted me as an intern at Microsoft Research, gave me the opportunity to work on a project where I got to improve my understanding of *Coq* at a tremendous speed, and introduced me to a project that I had not believed possible.

Thanks to all the employees of Galois, Inc., where I gave a presentation of my work earlier this year that was very well received, and followed by great feedback. In particular, David Christiansen pointed me to relevant literature that made part of the work much easier. It is an immense pleasure to have been offered a position to work alongside the people I met that day.

Finally, I must thank my mother for her love and support. This journey must have been quite the ordeal for her too, and it was hard to convey the reasons why I still had so much left to do before I was done. Yet, she was understanding, and always came through at the most difficult moments, no questions asked. Such care is commendable.

Chapters 3 and 4, in part, are currently being prepared for submission for publication of the material. The dissertation author was the primary investigator and author of this material. The only co-author is their advisor, Sorin Lerner.

VITA

2012 - 2018	Doctorate of Philosophy, Computer Science, University of California San Diego
2008 - 2012	Diploma of Engineering (Master's degree), ENSEIRB-MATMECA
2006 - 2008	Classe préparatoire, Lycée Louis Barthou
2001 - 2006	Baccalauréat, Science specialization, Lycée Saint-Cricq

PUBLICATIONS

Robert, V. and Leroy, X., 2012, December. A formally-verified alias analysis. In International Conference on Certified Programs and Proofs (pp. 11-26). Springer, Berlin, Heidelberg.

Ricketts, D., Robert, V., Jang, D., Tatlock, Z. and Lerner, S., 2014, June. Automating formal proofs for reactive systems. In ACM SIGPLAN Notices (Vol. 49, No. 6, pp. 452-462). ACM.

ABSTRACT OF THE DISSERTATION

Front-end tooling for building and maintaining dependently-typed functional programs

by

Valentin Robert

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Sorin Lerner, Chair

Dependently-typed functional languages are increasingly popular, but due to the complexity of their type systems, there is still a lot of friction in the user experience, both for beginners who try to learn the concepts, and expert users who must write and maintain complex code bases. We explore ways to alleviate those burdens by providing novel front-end tooling for this class of languages.

In order to help beginners, we explore new visualizations and automation techniques, focusing on three pain points we identified in the learning process. We evaluate, via a longitudinal user study and an A-B study, their effectiveness in terms of learning to use those languages, enjoyment of the learning process, and productivity on solving beginner-level exercises.

In order to help experts, we prototype a tool that helps in the refactoring of programs, partially eliminating the tedium of propagating changes throughout large code bases. Our tool is built around a small dependently-typed functional core language, but it supports extensions to richer languages, with similar or weaker type systems. We demonstrate this by extending it to support *OCaml*, a widely used, modern functional language without dependent types.

Introduction

This thesis aims to design and evaluate novel tools for assisting users of functional programming languages and proof assistants.

Over the past couple decades, the concept of *functional programming* has flourished from an academic object of interest to a trending topic in both academia and industry. Concepts that used to be mostly relevant in the context of functional programming languages, for instance, *anonymous functions*, *currying*, *monads*, or *dependent types*, are percolating into many functional languages, as well as mainstream imperative languages.

Unfortunately, these concepts tend to have a higher level of abstraction than their imperative counterparts, and are often dismissed by programmers as esoteric, until their benefits are made clear. This has happened repeatedly over the history of programming languages: structured programming seemed restrictive until its maintenance benefits became clear, garbage collection seemed non-viable until their implementation became fast enough, monads were just a mathematical concept until their relationship with effectful computations was made crisp.

Proof assistants, that is, software tools that allow programmers to reason formally about their code, are currently in a shifting period. While they have existed for several decades, their recognition as a valuable tool is still in progress. There are multiple explanations for this. First, the underlying meta-theories are still under active research, and as a result, the software implementations are often not as polished as their mainstream programming counterparts. Second, learning to use these proof assistants is a hard task, often requiring extensive research and academic readings. Compared to mainstream programming languages, tutorials for proof assistants are both scarce and terse. Finally, there are but a few examples of successful, broadly-used software built within proof assistants.

On this subject, a few pioneer projects have helped make the case for using proof assistants. The CompCert C compiler Leroy [22] demonstrated that a verified compiler for a "real-world" programming language was not only feasible, but was independently proven to be more robust than its competitors by Yang et al. [37] and Le et al. [21]. The seL4 micro-kernel Klein et al. [19] also demonstrated that verified software can build such low-level artifacts as the kernel of an operating system. On a more theoretical side, the *Coq* proof assistant was also used to build a mechanized proof of the four-color theorem Gonthier [13], as well as the odd-order theorem Gonthier et al. [15]. These proof efforts not only helped make software verification more broadly known and respected, they also provided several libraries and design principles for software verification. The Mathematical Components library Gonthier [14], for instance, was built alongside the two theorems previously mentioned, and offers a vast array of reusable mathematical objects, as well as a methodology for developing large-scale proofs using small-scale reflection, packaged in a library called *SSReflect*.

For proof assistants to become popular, we believe it will take efforts on multiple fronts. In order to be approachable, we need better course material, accessible to programmers outside of academic settings. We also need tools that are more suitable to beginners, with softer edges, so to speak. There is good progress on this front. The Software Foundations Pierce et al. [29] collection of books has been incrementally designed over the past decade, and is regarded as one of the best introductions to the *Coq* proof assistant and mechanized theorem proving in general. Other languages are also providing their own learning material, for instance, Programming Language Foundations [1] for *Agda*, or the *Idris* tutorial [2].

The design of proof assistants, their libraries, and the programming languages they are based upon, will also need refining. Proof engineering is still a fairly new endeavor, and will require years of effort before proper abstraction mechanisms, design principles, and tools are designed, evaluated, and adopted. For instance, there are two separate mechanisms for ad hoc overloading in *Coq*, with each their strength and weaknesses, namely *type classes* and *canonical structures*. The former is not very robust, often resulting in undecipherable error messages, while the latter is more powerful, but requires abusing other features of the proof assistant to program it.

In this dissertation, we want to explore two aspects of the interaction of users of proof assistants. The first aspect of interest is the barrier to entry previously mentioned. The author noticed several rough edges in the process of learning to use a proof assistant, both through their own learning experience, and through teaching other beginners to use the *Coq* proof assistant. These problems range from conceptual ones, like the ability to make proper mental models for what happens “behind the scenes” when one interacts with a proof assistant, to more practical issues, like being able to find the relevant theorems, lemmas, tactics, etc.

The second aspect we are interested in caters to a broader audience. Beginners often lack the foresight to get their data type definitions to be well tailored for a problem on the first try. Experts often need to refactor, add or remove features, to an existing program. In both cases, the user will want to go back to existing definitions, update

[1] https://plfa.github.io/

[2] http://docs.idris-lang.org/en/latest/tutorial/

them, and re-execute the existing proofs to get back to their problem at hand. Unfortunately, in most proof assistants, and programming languages in general, an update in some definition will have repercussions throughout a code base, that will require the user's attention to fix. While some of these changes indeed require scrutinizing, either because new values or new proofs need to be created from scratch, often times, a decent fraction of the changes necessary are systematic, deterministic consequences of the changes made upstream.

This slows down beginners in their learning process, especially since they will lack the knowledge necessary to make their code and proof robust to such changes. It also occasionally slows down experts, who need to update existing, working proofs, if only to rename variables, reorder arguments to a function or constructor, etc. We are interested in the possibility of eliminating the tedium out of this process, accelerating users by automatically figuring out the systematic changes, and leaving them only the task of filling the blanks that require creativity or expertise.

Chapter 1

Background

This section provides a brief high-level introduction to basic concepts used in the rest of the dissertation.

1.1 Static typing

Programmers typically do not write programs by simply manipulating bits in the computer's memory: most languages offer abstraction mechanisms that let their users reason at a higher level. One such high-level abstraction is the notion of a *data type*. A programmer will want to manipulate structured data such as numbers, Boolean values, or compound values made out of different values organized together in some fashion. They will then use, and possibly define, a set of operations to work on values from different data types. Those operations might sometimes be entirely oblivious to what data type they are manipulating, but frequently, they will expect some properties out of their input values, and provide some properties for their output values.

A *type system* is a mechanism that enforces some discipline about the proper use of values in a program according to a set of typing rules. *Typing rules* usually ascribe a type to every value of a program, and prescribe the typed contexts within which a value of a given type may be used. A *type* can be anything from a simple classification of values according to their nature (for instance, distinguishing numbers from functions), to more semantic rules that may sometimes be defined by the user of the language. In this dissertation, we will use the notation $t : \tau$ to indicate that a term t has type τ.

A *static* typing discipline allows the programming environment (be it a compiler, an interpreter, or any other language tool) to reject programs **before they run**. On the opposite, a *dynamic* typing discipline enforces its typing discipline on-the-fly, as programs execute. This allows more programs to execute, as long as their execution path does not encounter a typing violation. On the other hand, this provides less safety, as latent errors in the programs stay unnoticed until an execution path triggers them. In this dissertation, we will focus solely on static typing disciplines.

There can be many reasons for wanting to reject a program, whether statically or dynamically. The simplest case is a breach of expectation between a value and the context into which it is passed: this is usually called a *type error*. For instance, a function declared to expect a number input might not be allowed to receive a string instead.

Another, less obvious, unfortunate reason for rejecting a syntactically-valid, semantically valid program, is that the static enforcement of typing rules cannot, in general, be complete. For instance, dependent type systems (covered in Section 1.2) cannot safely ensure their typing discipline in the presence of arbitrary recursion. In order to reject all *unsafe* programs before they are run, they must restrict the programs they allow to some conservative subsets of all *safe* programs.

While the idea of preventing programmers from running syntactically-valid programs might seem like a nuisance, it provides at least two benefits. From the programmer's point of view, a disciplined use of the type system can help them catch, before their program is run, conditions that would make the program crash were it to be executed. These conditions can often be indicated at locations close to the source of error. The typing discipline can also provide information that can be leveraged as documentation, or in order to perform code analyses and transformations that would be intractable or unsafe without types.

Strong typing disciplines can even benefit the programmer tenfold, by providing means of tracking security properties Volpano and Smith [33], resource usage and sharingNaden et al. [27], dimensions of units-of-measure Kennedy [18], etc.

1.2 Dependent types

This dissertation will mostly focus on *dependent* type systems. A *dependent* type is a type whose definition depends on a program value. Non-dependent type systems can only express lightweight relationships between the input and output of functions, as well as lightweight constraints on what these inputs and outputs may be. In a dependent type system, one can express such types as *the type of functions that take a number and return a larger number*, or *the type of positive numbers that are less than 256*. In order to express these, one must be able to mention in the type, either concrete values like `256`, or program values like the input value of the function.

In order to define functions whose output type depend on the value of their input, dependent type systems must have a *type former* (i.e. a syntactic construct) for *dependent functions* [1]. In the literature, the type of functions which accept an input value `a` of type `A`, and return an output value of type `B(a)` (where `B` is a family of types indexed by values of type `A`), is often written as either:

[1] *Dependent functions* are sometimes referred to as *dependent products* in the literature. Unfortunately,

- `(a : A) → B(a)`
- `∀(a : A), B(a)` [2]
- `Π(a : A) → B(a)` [3]

We will tend to use the latter, and refer to those types as Π-types.

When a dependent function takes several arguments, we will group them all into a single Π, and coalesce successive values of the same type in groups under the same set of parentheses. In practice, this means that the types:

```
Π(a : X) (b c : Y) (d : Z) → R(a, b, c, d)
```

and

```
Π(a : X) → Π(b : Y) → Π(c : Y) → Π(d : Z) → R(a, b, c, d)
```

are syntactically equivalent, and we will favor the former (shorter) syntax.

Nested Π-types, that is, ones that appear to the right of the arrow of an enclosing Π-type, can depend on the values of the previously quantified variables. We call a Π-*telescope* any sequence of Π-types *directly* nested within one another. We can summarize them in a list of bindings and their types, where each type is allowed to, but does not have to, depend on previous bindings. For instance, the type:

```
Π(a : X) (b : Y(a)) (c : Z(a)) → R(a, b, c)
```

contains a telescope of three bindings (namely, `a`, `b`, and `c`), and the type of the last

the similar but different concept of a *dependent pair* is sometimes referred to as either a *dependent sum* or, confusingly, a *dependent product*. Since both views are reasonable, and in order to avoid confusion, we will strictly adhere to the unambiguous names of *dependent function type* and *dependent pair type* in this dissertation.

[2] The ∀ symbol is pronounced "for all".

[3] The Π symbol may also be pronounced "for all" in such context, or "pi".

two bindings depend on the value of the first binding.

Dependent types may also manifest themselves in other forms depending on the constructs of the language that integrates them. For instance, languages with records may allow dependent records, where the type of some fields may depend on the value of some other fields. A typical example is the packing of a plain data structure with extra properties that we want it to have, for instance, ensuring the binary-search-tree (or *BST*) property of a binary tree:

```
Inductive BinarySearchTree a := MakeBinarySearchTree
  { tree  : BinaryTree a
  , isBST : IsBinarySearchTree tree
  }.
```

For the scope of this dissertation, we do not handle such features explicitly, but believe that the constructions we present are amenable to the introduction of those features: the simplest flavor of dependent records can be simulated using a combination of constructs we cover.

1.3 Proof assistants

Our first contribution focuses mostly on programs built using a *proof assistant*. A proof assistant is a software tool allowing a user to define mathematical structures, with their axioms and rules, and carry mechanized proofs of properties of those structures. By *mechanized*, we mean that the software has the ability to check that a proof is correct, with respect to a set of rules. Many proof assistants rely on the Curry-Howard isomorphism [16], bridging the gap between formal logic and typed programming. In those systems, logical propositions can be readily expressed as types, in the same formal system within which programs can be defined. Proofs built this way are essentially programs, whose computational content is often less interesting than the fact that they are well-typed, and as such, are *witnesses* for the type/proposition they inhabit.

Proof assistants come in a large variety, both from a theoretical point of view, and in terms of user experience. On the theory side, there are many logical systems of interest, and different proof assistants tend to focus on some classes of those. For instance, the *Coq* proof assistant focuses, by default, on an intuitionistic fragment of the calculus of (co)-inductive constructions, or *CoC*, but allows itself to be extended with axioms to support classical reasoning, or extensional notions of equality, if needed.

A proof assistant is typically composed of several interacting pieces:

- a *programming language*, allowing the user to define programs that they wish to execute and/or reason about formally,
- a *specification language*, allowing the user to define properties of programs, or general theorems, that they wish to prove,
- a *proof language*, allowing the user to build those proofs.

Note that these languages need not be different from one another, and several languages can help fulfill one of those purposes. For instance, in the *Coq* proof assistant, the programming language and the specification language are the same language, called *Gallina*, and the proof language can be either *Gallina* itself, or a proof-building scripting language called *Ltac*.

Learning to use such a proof assistant therefore requires familiarizing oneself with not only a programming language with a complex type system, but also the logical foundation of formal proofs, and the idiosyncrasies of the proof environment of choice.

For the *Coq* proof assistant, novice users are generally taught to use *Ltac* to build proofs, since manually building *Gallina* terms requires more expertise and is usually tedious. An *Ltac* script is a sequence of commands (usually called *tactics*) directing the proof assistant as to what logical rules to use in order to progress in building the proof term. These tactics include:

- *proof-solving tactics*, which attempt to complete a proof obligation by constructing the complete proof,

- *case-splitting tactics*, which break a proof obligation into several sub-obligations, by using some rule of the formal logic with multiple antecedents,

- *bookkeeping tactics*, which modify the context or the goal of the current proof obligation, either by adding/removing/reordering/rewriting in hypotheses, or by performing modifications in the goal.

A typical proof can therefore be thought of as a tree, branching on case-splitting tactics, extending on bookkeeping tactics, and with proof-solving tactics as leaves. The proving process is rarely linear: many proofs will require using concepts such as *case analysis* and *induction* in order to break down a complex goal into specialized sub-goals that can be solved separately. In those sub-cases, one will often need to perform applications and rewriting using existing theorems.

The *application* of a theorem (or a hypothesis) `(t : τ)` can be performed in either a hypothesis or a goal. To apply it in a hypothesis `(a : α)`:

- the type of the applied term, `τ`, must reduce to a Π-telescope:

  ```
  Π(t1 : τ1) ... (tn : τn) → tr : τr
  ```

- the type of the target hypothesis, `α`, must be compatible with of one of the `τi` (if there are multiple candidates, the first such `τi` will be considered)

It then yields the conclusion of the applied term, `τr`, as a hypothesis, but also adds new obligations for all the other antecedents in the telescope. Intuitively, the applied term `t` was fully applied, with formal parameter `τi` receiving argument `a`, and all other formal parameters receiving arguments to be determined by solving the new obligations. That is, given the context:

```
A, B, C : Prop
H1 : A → B → C
H2 : B
==================
C
```

applying hypothesis `H1` in hypothesis `H2` yields the following two obligations:

```
(* 1. Hypothesis H2 became the conclusion of H1. *)
A, B, C : Prop
H1 : A → B → C
H2 : C
==================
C
```

and:

```
(* 2. In the original context, one must prove A. *)
A, B, C : Prop
H1 : A → B → C
H2 : B
==================
A
```

Conversely, a theorem or hypothesis can be applied to the goal of the current obligation if its conclusion matches the goal. It yields an obligation for each antecedent. For instance, applying hypothesis `H1` from the original context to the goal would yield the following two obligations:

```
(* 1. One must prove the first antecedent A. *)
A, B, C : Prop
H1 : A → B → C
H2 : B
```

```
==================
A
```

and:

```
(* 2. One must prove the second antecedent B. *)
A, B, C : Prop
H1 : A → B → C
H2 : B
==================
B
```

Rewriting with a theorem or a hypothesis is a similar concept, but for dealing with equalities (or, more generally, structures that admit equivalence relations, called *setoids*). We will focus on the simple case of equalities. Again, one can either use rewrite in a hypothesis, or over the goal. One can rewrite with a theorem or hypothesis as long as its conclusion is an equality. The rewriting consists of replacing one side of the equality with the other side, for a given occurrence. It is a directed operation, either replacing the left operand of the binary relation with the right one, or vice-versa.

For instance, given this original context:

```
P, Q : ℕ → Prop
x, y, z : ℕ
H1 : Q z → x = y
H2 : P x
==================
P y
```

rewriting from left to right with hypothesis `H1` in hypothesis `H2` yields the two obligations:

```
(* 1. x has been replaced with y in H2. *)
P, Q : ℕ → Prop
x, y, z : ℕ
H1 : Q z → x = y
H2 : P y
==================
P y
```

and:

```
(* 2. One must prove the antecedent. *)
P, Q : ℕ → Prop
x, y, z : ℕ
H1 : Q z → x = y
H2 : P x
==================
Q z
```

while rewriting from right to left with hypothesis `H1` in the goal yields the two obligations:

```
(* 1. y has been replaced with x in the goal. *)
P, Q : ℕ → Prop
x, y, z : ℕ
H1 : Q z → x = y
H2 : P x
==================
P x
```

and:

```
(* 2. Again, one must prove the antecedent. *)
P, Q : ℕ → Prop
x, y, z : ℕ
H1 : Q z → x = y
H2 : P x
==================
Q z
```

A novice user will need to learn to use these tactics effectively, but will also need to learn about the families of theorems that are applicable in their proof development. For instance, the *Coq* prelude contains 106 equality theorems, and importing the module about lists increases this number to 1006. While there are facilities to look up theorems of interest based on the shape of their type, this still lacks in ease of discovery, and it is not rare for one to miss or re-derive an existing theorem.

1.4 Program refactoring

Programs are rarely written once and for all: bugs may be found that must be fixed, requirements may evolve in ways that require rewriting parts of a program, and programs may be re-written in semantically-equivalent ways in order to account for performance, resource-efficiency, or even simply stylistic concerns.

The concept of *refactoring*, which was introduced as early as in Wirfs-Brock and Johnson [36], characterizes program transformations that preserve the semantics of the program being manipulated. Semantics-preserving transformations of a program can include tasks such as:

- reordering parameters to functions, classes, modules, and other parameterized language constructs,
- changing the location of some units of code, possibly crossing module or file boundaries,
- consistent renaming of variables and constants,
- abstracting over some concrete code, or concretizing some abstract code,
- etc.

Examples of non semantics-preserving transformations include:

- adding, removing, or changing parameters to a function,
- adding, removing, or changing parameters to a data constructor.

Chapter 2

PeaCoq: novel display and user interaction in proof assistants

This chapter will focus on the development and evaluation of innovative front-end features for proof assistants, with a focus on helping novice users.

2.1 Background

Proof assistants have a steep learning curve. Not only do they require an understanding of high-level mathematical concepts, including but extending beyond the ones presented in Chapter 1, but they also require the user to learn the idiosyncrasies of their proof assistant of choice. Even experts of a given proof assistant would require a decent amount of time to become proficient in a different proof assistant.

Let's focus on the *Coq* proof assistant. Learning to use it requires getting familiar with the Calculus of (Co-)Inductive Constructions, a dependently-typed lambda calculus. Proficiency in this language requires becoming familiar with dependent pattern matching, universe hierarchies, induction and co-induction, well-founded relations, etc. This covers the knowledge required to write code and specifications in this proof assistant.

In order to write proofs, the user must learn an additional, untyped language of tactics called *Ltac*. This language contains more than a hundred tactic formers, with several variations for each tactic. While many proofs can be carried out with just a small subset of those, complicated proofs often require a large arsenal of tactics and a thorough understanding of the tactic language.

Additionally, writing proofs over standard data types requires being able to locate relevant theorems within the standard library. This standard library, by default, includes more than a thousand lemmas. While *Coq* offers facilities to search a relevant lemma by giving the shape of its expected type, this is still a somewhat tedious endeavor.

We have identified three challenges in the learning process for novice users that we would like to tackle with *PeaCoq*:

1. conceptualizing and keeping track of the *proof tree structure* while building proofs,
2. identifying the effects of a tactic on the proof context,
3. identifying relevant tactics that can be applied in a given proof obligation.

2.2 Design

2.2.1 Design of *PeaCoq*

We will discuss the global architecture of *PeaCoq* that allows us to build all the features we mentioned. Each feature will then be described individually.

PeaCoq is designed following a client-server architecture. The server handles the *Coq* process, and exposes end-points to communicate with it via HTTP. The client is a web application, containing the front-end view that users interact with, and facilities for driving the interactions with the server. Let us describe each side further.

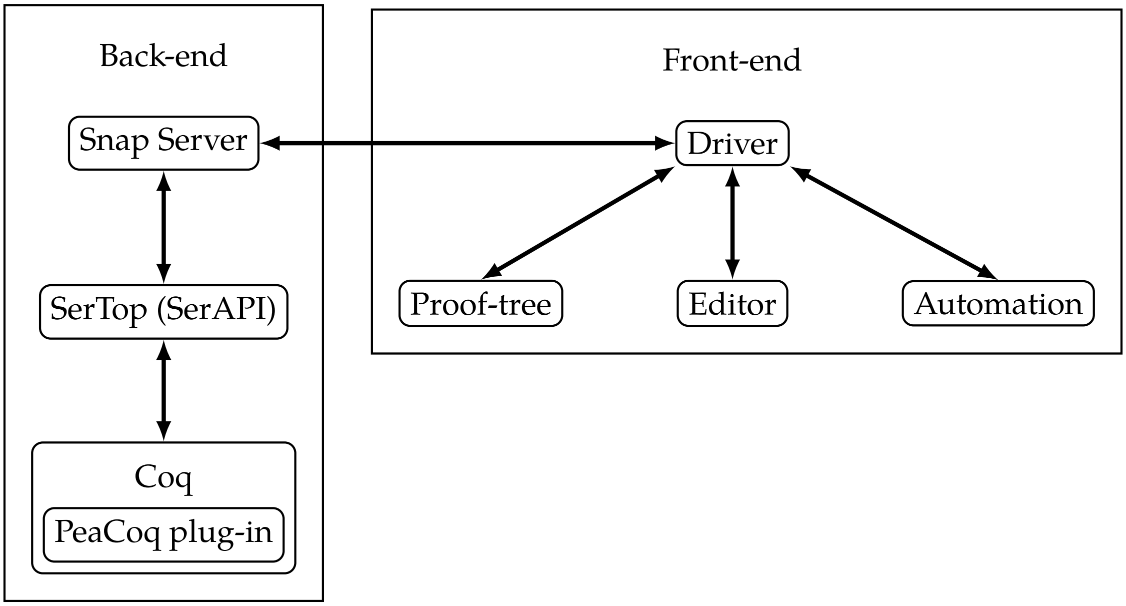


Figure 2.1. *PeaCoq*'s architecture

Back-end / Server

In order to let front-ends interact with it, the *Coq* process exposes an API that serializes metadata and accepts certain commands. Unfortunately, prior to *PeaCoq*'s development, very few tools had used this API, and as a result, it is lacking both in polish and in features.

The protocol it uses is based upon XML syntax, and exposes commands to manipulate an abstract document (a collection of *Coq* sentences) and command and observe its execution.

While developing *PeaCoq*, we interacted with Emilio Jesús Gallego Arias, who was developing a layer over this XML protocol, named *SerAPI* (for serialization API). This layer offers a less rough API based on s-expressions, abstracting over some minute details of *Coq*'s implementation.

However, at the time *PeaCoq* was built, *Coq* was not exposing enough of its internals for our needs. In order to remedy this, we also built a *Coq* plug-in to expose this data. Plug-ins circumvent the IDE API by directly being compiled and loaded alongside the *Coq* code: thus, they have access to all the public interfaces of *Coq*'s internal modules. Once our plug-in is loaded, it registers as a *Vernacular* command that can be invoked either by users, or through the IDE API.

In order to communicate with front-ends, the back-end is driven by a HTTP server, based on *Haskell*'s *Snap* framework. The details of its implementation are fairly mundane: it simply listens to requests from the front-end, passes them down to *Coq* through *SerAPI*, and forwards the responses back without much processing. However, this architecture provided several advantages:

- Since the back-end communicates with the front-end over HTTP, the two need not be on the same physical machine. For instance, we have had the back-end running on a powerful Internet-facing machine, and connected it from a less powerful phone, on public transit. It worked flawlessly, even in the presence of computation-intensive loads like our automation, because the computation was happening server-side. Meanwhile, the lightweight client was only processing display and communication with the server.

- The back-end was built in such a way that it could accept multiple connections at once. Each connection would spawn its own, separate instance of *Coq*. This means that multiple clients could independently connect and work on the same server. We used this in our study at the University of Washington, where one server was shared among all students, who did not need to install *Coq* on their machine.

Front-end / Client

The front-end of *PeaCoq* is a web application, currently written in *TypeScript*, a statically-typed dialect of *JavaScript*. We call *driver* the part of the code responsible for communicating with the back-end. We use a reactive programming library called *RxJS*, which provides a stream abstraction for sequences of values. Messages from the back-end are published as streams, and each of the components involved in the front-end (namely, the editor, the proof-tree, and the automation layer) can subscribe to only those messages they need to observe. Conversely, those components publish streams of commands they'd like the back-end to process, which the driver subscribes to, and orchestrates the dispatch of.

This orchestration can require some finesse. The API provided by *Coq* is fairly stateless, that is, *not* in the sense that there are no states, but rather, in the sense that there is no notion of a current, mutable state. Instead, each state is given a unique identifier, and subsequent commands may explicitly state over which previous state they wish to be applied. This abstraction holds well at the document level, but unfortunately, some commands change global, mutable flags, in ways that can be observed.

In order to prevent issues with such observable, mutable state, some sequences of commands must be processed atomically, that is, no other action may be interleaved with the sequence. In order to achieve this, the layers emit their commands not as a simple sequence of commands, but as a sequence of sequences of commands, to be dispatched atomically.

This raises another concern: some atomic sequences of commands have data dependencies. In particular, later commands often need to know the *output* of previous commands. A pervasive example of such a pattern is found in our automation layer, where commands must be silently tried in the background, their output must be gathered, and then their effect must be cancelled. In order to cancel an action, we must tell *Coq* the identifier of the state(s) to be cancelled, but that identifier is only know asynchronously, in a response from the action that created said state.

Once again, this problem is easily solved by issuing atomic sequences to the driver not as an array of sequences, but as a stream of commands. When the driver chooses to emit a given atomic sequence, it subscribes to it. It will subsequently, and asynchronously, receive one or many commands, until the stream indicates it has completed. Elements from this stream can be asynchronously built from other streams, and so we can build those dependent atomic sequences as:

```
const commandToCancel = new Add('Command To Cancel')

// By convention, we put a $ sign behind stream variables.
const answer$ =
  added$
  .filter(sameTagAs(commandToCancel))
  .takeUntil(completed$.filter(sameTagAs(commandToCancel)))

const output$ = answer$.map(...) // do what you need

const cancelCommand$ =
  answer$.map(a => new Cancel([a.stateId]))

const commands$$ =
  Rx.Observable.concat([
    Rx.Observable.of(commandToCancel),
    cancelCommand$
  ])
```

Understanding this code is not necessary, but here is a high-level explanation for the interested reader:

- We create (but, do *not* issue yet!) the command whose output we want to observe. When a command is created, it acquires a unique tag, that is sent to *Coq*, and appears in responses. This helps us filter those answers that correspond to this query.
- We preemptively subscribe to the stream of answers, looking for ones with the tag of our command. In case there is no answer, we also cut this subscription short when the stream of completed answers emits. Every command will produce such an item, so we are guaranteed to terminate.
- We can listen to this answer, and compute our output however we see fit.
- We can also listen to this answer in order to emit a cancel command for it.
- The final atomic sequence of commands we send to the driver is the concatenation of two observables: the command, and its corresponding cancel command.

This is the most important abstraction on the front-end side. Most of the code is event-driven by subscribing to those streams and producing streams of requests.

2.2.2 Conceptualizing the proof tree structure: the tree view

When a user of the *Coq* proof assistant writes a proof script using the *Ltac* tactic language, they are effectively guiding the tool in building a derivation, in the underlying formal system, witnessing the truth of the theorem at hand.

According to the Curry-Howard correspondence, this derivation can be equally thought of as a well-formed tree, combining axioms and rules of the logical system, or, as a well-typed λ-term. For instance, the theorem:

```
Theorem swap : Π (A B : Prop) → A ∧ B → B ∧ A.
```

can be witnessed by the following logical derivation:

$$
\cfrac{
 \cfrac{
  \cfrac{
   \cfrac{
    \cfrac{}{\ldots, H : A \land B \vdash A \land B}
    \qquad
    \cfrac{
     \cfrac{}{\ldots, H_A : A, H_B : B \vdash B}
     \quad
     \cfrac{}{\ldots, H_A : A, H_B : B \vdash A}
    }{\ldots, H : A \land B, H_A : A, H_B : B \vdash B \land A}\;\land\text{-}\textsc{intro}
   }{\ldots, H : A \land B \vdash B \land A}\;\land\text{-}\textsc{elim}
  }{A : \mathtt{Prop}, B : \mathtt{Prop} \vdash A \land B \to B \land A}\;\Pi\text{-}\textsc{intro}
 }{A : \mathtt{Prop} \vdash \Pi(B : \mathtt{Prop}) \to A \land B \to B \land A}\;\Pi\text{-}\textsc{intro}
}{\vdash \Pi(A\,B : \mathtt{Prop}) \to A \land B \to B \land A}\;\Pi\text{-}\textsc{intro}
$$

A corresponding *Coq* proof following the same strategy matches the structure of the derivation quite closely:

```
Proof.
  intros A B H.
  destruct H as [HA HB].
  split.
  + exact HA.
  + exact HB.
Qed.
```

and the proof term that it generates also follows the same structure, though it is less obvious to the beginner:

```
λ A B H → match H with
          | conj HA HB => conj HB HA
          end
```

In fact, the earlier derivation corresponds exactly to the one that the type-checker follows when checking the type of this last term:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\ldots, H : A \wedge B \vdash \texttt{H} : A \wedge B}
\qquad
\dfrac{
\dfrac{}{\ldots, H_A : A, H_B : B \vdash \texttt{HA} : A}
\qquad
\dfrac{}{\ldots, H_A : A, H_B : B \vdash \texttt{HB} : B}
}{\ldots, H : A \wedge B, H_A : A, H_B : B \vdash \texttt{conj HB HA} : A \wedge B}\ \wedge\text{-}\textsc{intro}
}{\ldots, H : A \wedge B \vdash \texttt{match H with conj HA HB => conj HB HA end} : B \wedge A}\ \wedge\text{-}\textsc{elim}
}{A : \texttt{Prop}, B : \texttt{Prop} \vdash \texttt{λ H → match ... end} : A \wedge B \to B \wedge A}\ \Pi\text{-}\textsc{intro}
}{A : \texttt{Prop} \vdash \texttt{λ B H → match ... end} : \Pi(B : \texttt{Prop}) \to A \wedge B \to B \wedge A}\ \Pi\text{-}\textsc{intro}
}{\vdash \texttt{λ A B H → match ... end} : \Pi(A\ B : \texttt{Prop}) \to A \wedge B \to B \wedge A}\ \Pi\text{-}\textsc{intro}
$$

Therefore, there are two equivalent ways to think about tactics:

- they add steps in the derivation tree, possibly finishing, prolonging, or splitting branches,
- equivalently, they add subterms in the partial proof term, possibly filling, continuing, or adding holes.

Unfortunately, due to the sequential nature of the proving process in a proof assistant, this tree structure is somewhat hidden from the user, who receives proof obligations one by one in a traversal of the derivation. For instance, in the previous proof, after calling `split.`, the user is left with two obligations, originating from the two arguments that the conjunction introduction rule must receive. However, in *CoqIDE*, the main interface to the proof assistant, the resulting state is displayed thus:

```
2 subgoals
A, B : Prop
HA : A
HB : B
______________________________________(1/2)
B
______________________________________(2/2)
A
```

All remaining sub-obligations are counted, and displayed sequentially, no matter where they come from. In this simple example, it is quite easy to follow what has happened, and to remember that a second sub-obligation must eventually be solved. In more complex examples, the delayed sub-obligations can accumulate as the proof derivation splits into multiple cases, and it is often not immediately clear where we are in a large proof after we finish a sub-obligation.

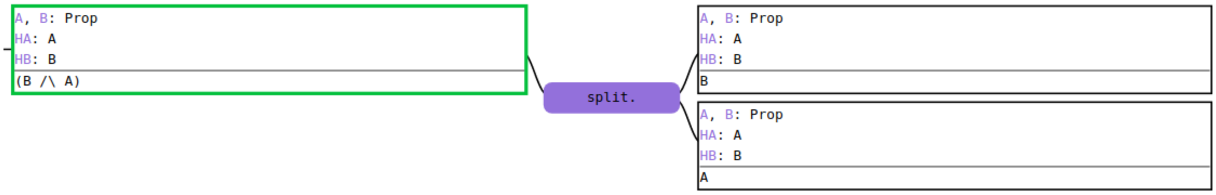
The newest versions of *Coq* include a mechanism to help with this bookkeeping, named *bullets*. By using bullets like `+`, `-`, `*`, after a splitting point in a proof, the user can indicate their intent to focus on the sub-obligations generated during that last step. Preexisting sub-obligations are temporarily hidden, until all newly generated sub-obligations are solved, at which point the preexisting ones are restored back into view. While this feature gives the user some agency over the list of proof obligations being displayed at any given time, it still requires the user to have a mental map of their location in the underlying proof tree.

In order to make this mental map more tangible in the user experience, we designed a feature that will display this proof tree, as it is being built and navigated, to the user.

Building the proof-tree view

Our proof-tree view is a tree, as shown in Figure 2.2, whose nodes fall in two categories:

- nodes in odd layers are *obligation nodes*, that is, they are related to a given proof obligation,
- nodes in even layers are *tactic nodes*, that is, they are related to the invocation of a given tactic.



```
A, B: Prop
HA: A
HB: B
(B /\ A)
```

```
split.
```

```
A, B: Prop
HA: A
HB: B
B
```

```
A, B: Prop
HA: A
HB: B
A
```

Figure 2.2. Proof-tree view: three obligations nodes and one tactic node

When the user enters a proof, a single, root obligation node is created. This node corresponds to the current, single proof obligation. In order to progress, the user will invoke a tactic. When they do, a tactic node will be inserted as a child of the current obligation node, thus denoting that this tactic was ran from that context. Depending on the outcome of the tactic execution, one of the following will happen:

- if the tactic yields sub-obligations, these are added as children to the tactic node, and the focus shifts to the first such sub-obligation,

- if the tactic yields no obligation (i.e. concludes the current obligation), then the solved sub-trees are visually folded, and the focus moves to the next pending obligation, if any. When there are none, it means the proof is completed.

If the current obligation resulted from the execution of a tactic (i.e. for all obligations but the root one), the user may backtrack their decision and return to the state prior to the execution of the parent tactic.

2.2.3 Identifying the effects of a tactic: visual diffs

Apart from terminators, which finish an obligation, most tactics will operate a transformation on the goal of the current context. The transformations can end up changing entire types, or replacing sub-terms of some types with other terms. Some tactics will also add hypotheses, remove some, or reorder hypotheses, whether voluntarily or as part of how they need to operate.

In order for the user of a proof assistant to assess the usefulness of a tactic's execution, they must be able to identify what changed by running the tactic. This is often done in a ad-hoc way, by going back and forth between the state before and after the tactic, while visually inspecting differences. This is not satisfactory for several reasons.

First, finding out differences using this method is tedious. Proof contexts often contain dozens of variables and hypotheses, which makes the task of finding the changes between two contexts both long, because the user might need to repeatedly read several lines, and is error-prone, since the user might not notice a change.

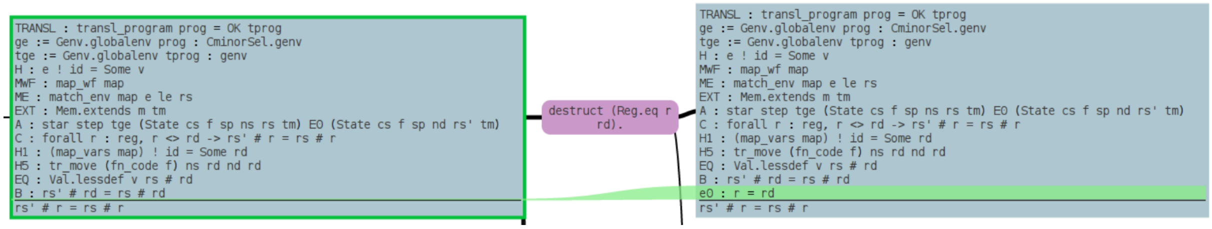
Second, *Coq* does not offer great facilities for caching results, or viewing results at a different proof context than the current one. While rolling back to the state prior to a tactic's execution is a fairly inexpensive action, due to the *Coq*'s state model, going back to the state after the tactic's execution requires running the tactic again from scratch. For slow tactics, the process can be extremely slow, and while the tactic is running, the user might forget about what they were looking for in the first place.

In order to have a better user experience, one would therefore need to be able to inspect the state, before and after the execution of a tactic, without having to run the tactic more than once. Additionally, visual help indicating which parts of the context have been modified could help accelerate finding the effects of a tactic, while reducing the chances of missing a change or mistakenly spotting a spurious change.

Fortunately, our proof-tree mechanism already provides us with a way of displaying a before/after view of a proof context with respect to a tactic's execution. When a tactic is tried, but not committed to, we can display it as a child of the current obligation node, and display its result as the child of this tactic node. By carefully aligning the two nodes, the user can have an instant view of the two sides for comparison, without needing to run the tactic ever again. We cache those results so that the user can move in the tree however they want. Our trees are created in such a way that obligations nodes have a unique execution history, such that when the user visits the same obligation node, we can display the same tactic nodes, without needing to run those tactics again.

We now introduce our notion of *visual diffs* as a means to highlight changes between two proof contexts that are visually juxtaposed. For a given hypothesis in the original proof context, one of the following three outcomes might happen to it as the result of a tactic execution: it may remain the same, it may disappear, or it may by modified (either by being moved around, of by having its name, term, or type changed). Similarly, for a given hypothesis in the final proof context, it may have originated from an unchanged hypothesis in the original proof context, from a changed hypothesis in the original proof context, or it may be a newly introduced hypothesis.

Examples of visual diffs are given in Figures 2.3 and 2.4. The visualization uses colored ribbons to indicate the three type of changes that may happen to a hypothesis.



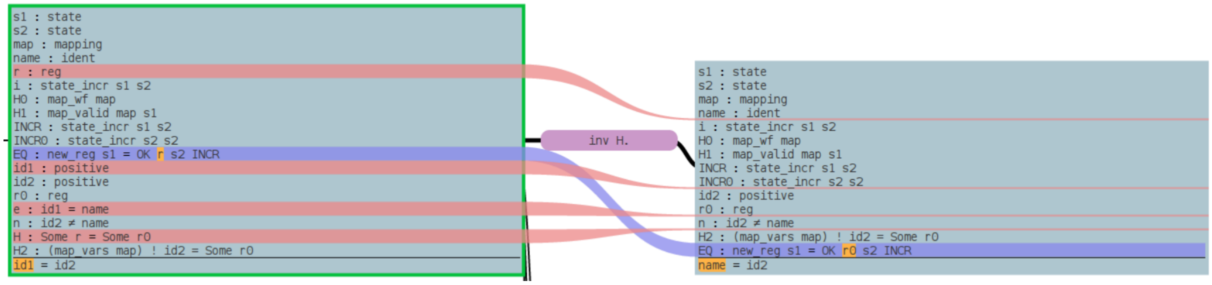
```
TRANSL : transl_program prog = OK tprog
ge := Genv.globalenv prog : CminorSel.genv
tge := Genv.globalenv tprog : genv
H : e ! id = Some v
MWF : map_wf map
ME : match_env map e le rs
EXT : Mem.extends m tm
A : star step tge (State cs f sp ns rs tm) E0 (State cs f sp nd rs' tm)
C : forall r : reg, r <> rd -> rs' # r = rs # r
H1 : (map_vars map) ! id = Some rd
H5 : tr_move (fn_code f) ns rd nd rd
EQ : Val.lessdef v rs # rd
B : rs' # rd = rs # rd
rs' # r = rs # r
```

```
destruct (Reg.eq r
rd).
```

```
TRANSL : transl_program prog = OK tprog
ge := Genv.globalenv prog : CminorSel.genv
tge := Genv.globalenv tprog : genv
H : e ! id = Some v
MWF : map_wf map
ME : match_env map e le rs
EXT : Mem.extends m tm
A : star step tge (State cs f sp ns rs tm) E0 (State cs f sp nd rs' tm)
C : forall r : reg, r <> rd -> rs' # r = rs # r
H1 : (map_vars map) ! id = Some rd
H5 : tr_move (fn_code f) ns rd nd rd
EQ : Val.lessdef v rs # rd
B : rs' # rd = rs # rd
e0 : r = rd
rs' # r = rs # r
```

Figure 2.3. Proof-tree visual diff between two obligation nodes (green ribbon)

A *green, expanding ribbon*, as seen in Figure 2.3 indicates that a hypothesis has been introduced in the new context. Its shrunk left end points between two hypotheses if the new hypothesis appears in the resulting context between these two hypotheses. Most of the times, introduced hypotheses appear last in the resulting context, and so their shrunk left end points at the end of the original context.



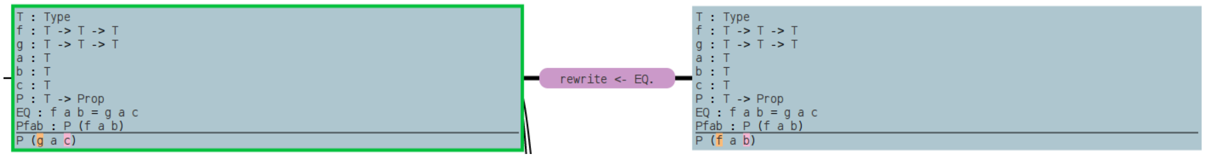
```
s1 : state
s2 : state
map : mapping
name : ident
r : reg
i : state_incr s1 s2
H0 : map_wf map
H1 : map_valid map s1
INCR : state_incr s1 s2
INCR0 : state_incr s2 s2
EQ : new_reg s1 = OK r s2 INCR
id1 : positive
id2 : positive
r0 : reg
e : id1 = name
n : id2 ≠ name
H : Some r = Some r0
H2 : (map_vars map) ! id2 = Some r0
id1 = id2
```

```
s1 : state
s2 : state
map : mapping
name : ident
i : state_incr s1 s2
H0 : map_wf map
H1 : map_valid map s1
INCR : state_incr s1 s2
INCR0 : state_incr s2 s2
id2 : positive
r0 : reg
n : id2 ≠ name
H2 : (map_vars map) ! id2 = Some r0
EQ : new_reg s1 = OK r0 s2 INCR
name = id2
```

Figure 2.4. Proof-tree visual diff between two obligation nodes (red and blue ribbons)

Conversely, a *red, shrinking ribbon*, as seen in Figure 2.4, indicates that a hypothesis has disappeared in the new context. Its shrunk right end points between the hypotheses that were surrounding it before its disappearance.

A *blue, constant-size ribbon*, as also seen in Figure 2.4, indicates that a hypothesis has either undergone any or all of those modifications: moved around, changed name, change type. For instance, the blue ribbon from Figure 2.4, highlights both how the `EQ` has moved from the middle of the context towards the end, and also the fact that it has changed.



```
T : Type
f : T -> T -> T
g : T -> T -> T
a : T
b : T
c : T
P : T -> Prop
EQ : f a b = g a c
Pfab : P (f a b)
P (g a c)
```

```
T : Type
f : T -> T -> T
g : T -> T -> T
a : T
b : T
c : T
P : T -> Prop
EQ : f a b = g a c
Pfab : P (f a b)
P (f a b)
```

Figure 2.5. Proof-tree visual diff between two obligation nodes (sub-term highlights)

Finally, *colored inline highlights* indicate, in both the old and new context, subterms that have changed. Those changes can appear within hypotheses that are part of a blue ribbon, as depicted for hypothesis `EQ` in Figure 2.4. They can also appear within the goal, as is the case in both Figure 2.4 and Figure 2.5. When there are several pairs of sub-terms that have changed within the same hypothesis (or the same goal), each pair is given a unique color, so as to facilitate distinguishing related pairs quickly for the user.

2.2.4 Automating tactic exploration in the background

In order to tackle our third identified challenge, that is, helping the user discover what actions they may take in a given context, we experimented with an automation technique. The technique itself is simple: while the user is not actively entering tactics, we can, in the background, generate and evaluate the results of a large body of tactics. We can then estimate which of those tactics seems relevant to the users and choose how to display them to the user.

The difficulty lies in the details, in particular, we must address the following problems:

- what tactics to try,
- which results are relevant,
- and how to display those results we deem relevant.

We will cover those in order, describing the set of options available. In our evaluation, we will highlight which choices we made.

What tactics to try?

Because *Coq* admits *Gallina* terms as arguments to some *Ltac* tactics, there is effectively an infinite number of tactics that can be performed at any step. Even ignoring those, many tactics take terms in the environment as arguments, and the standard library already introduces more than a thousand terms in the ambient environment. This makes it clear that an exhaustive search is not likely. A couple criteria will help us discern what tactics are worth trying.

First, we can differentiate tactics called *terminators* from the ones that are not. A terminator is a tactic whose success terminates the current obligation. Terminators come in different ways, from very simple ones that find a proof or a contradiction in the immediate context, to solvers for a given theory. For instance, the `assumption` tactic is a terminator that simply looks for an assumption in the current context whose type equates to the current goal (up to some notion of equality), while the `omega` tactic is a *complete* solver for *Presburger arithmetic*. There are only two outcomes out of the execution of a terminator: either the proof is found and the tactic succeeds and finishes the current obligation, or its execution fails.

On the other hand, non-terminator tactics can succeed either by finishing the current obligation, or by modifying it in any way, or even not doing anything. They can also lead to the creation of multiple obligations. For instance, the `split` tactic succeeds on goals that contain a top-level *conjunction*, possibly under a Π-telescope: it introduces all the binders in the telescope, and yields two obligations, one for each conjunct.

It is best to test inexpensive terminators first, since a success would mean we need not look further. On the other hand, some terminators can effectively diverge, either by taking an unreasonable amount of time, or by using an increasingly larger amount of resources, eventually throttling. In general, we will need to account for such slowdowns for all tactics with a timeout mechanism, as we don't want the automation machinery to have a negative performance impact for our users.

A second important aspect to consider is what arguments to pass to tactics that require them. The families of `apply` and `rewrite` tactics, for instance, all take as input one term to work with, and, for some, a second term indicating where to perform the work. Those terms can either be variables that are local to the current proof context, or any variable that is in scope from this file or imported files. The latter set tends to be between one and two orders of magnitude larger than the former. Therefore, while it is reasonable, but expensive, to try all the proof-local variables, it would be very expensive to try all identifiers in scope!

Finally, the set of tactics available in the *Coq* proof assistant is not static: the tactic language is extensible, and users frequently define both general-purpose and domain-specific tactics. These can be registered as hints to the built-in automation mechanism, in which case tactics like `auto` will use them appropriately. However, as far as we know, at the current time, there is no built-in mechanism for discovering user-defined tactics in scope. An automation mechanism would most likely benefit from such domain-specific knowledge of tactics to be used in a given proof.

Which results are relevant?

While we could display all the successful tactics and their outcome to the user, the amount of information would, in most cases, be overwhelming. In order for the suggestions to ever be useful, some triage is necessary.

After putting failing tactics out of the picture, our first observation is that many tactics produce the exact same obligations. Here, one might want to distinguish between two close varieties:

- Two tactics may produce the exact same partial proof term, yielding equal proof obligations. This is obviously the case when the two tactics are aliases of each other, but it can also happen when the two tactics take the same simple logical step. Let us refer to those as *proof-term equivalent executions*.

- Two tactics may produce the exact same proof obligations (same goals with same contexts), but build different proof terms. This could be a concern, when the terms being built have significant computational differences, especially in settings where proofs are relevant. Let us refer to those as *proof-context equivalent executions*.

Ideally, we would like to factor out proof-term equivalent executions. The user might still care about what tactic is used. For instance, wherever the `assumption` tactic works, tactics like `auto` or `intuition` should also work, but they might do additional work that is unnecessary. Similarly, if the assumption being used is called `H`, the tactic `exact H` should also work, but it might be less robust to changes, especially if the name `H` was automatically introduced by *Coq*.

How to display the relevant results?

Displaying the outcome of the automation also proves an interesting challenge. For terminators, or non-terminators that end up solving the current proof obligation, we can simply indicate to the user that they do. Tactics that make progress without solving the proof, however, must be somehow shown to the user alongside their result. In the simplest form, one could just present a list of those tactics, and let the user try them and witness their result on their own. This imposes quite a bit of cognitive load on the users, as they must visually inspect one or several proof obligations, trying to understand what changed between before and after the tactic execution.

To reduce that effort, we benefit from the visualization presented in Section 2.2.3. For each tactic we did not filter out, the user can align its resulting sub-obligation(s) to the current obligation, and get visual information about what has changed.

While visual diffs allow the user to quickly inspect the outcome of a given tactic, if there are too many tactics that make progress, going through them all one by one can still require a lot of time and concentration. Some members of one of our studies indicated the need for grouping the results. We followed a static grouping strategy, where tactics were grouped together based on similar intent: apply tactics, case analysis tactics, rewrite tactics, terminators, etc. This let users skip over entire classes of tactics that they know are not what they are currently trying to achieve in the proof.

2.3 Evaluation

We built a tool, called *PeaCoq*, to try and evaluate the usefulness of those techniques in a beginner setting. We conducted two studies: a longitudinal study in the classroom during one quarter, and a short A-B study with beginners.

2.3.1 Longitudinal study

The first study was conducted at the University of Washington, with the help of instructor Zachary Tatlock, during the Winter quarter of 2015. The study was approved by the institutional review board of the University of California San Diego (project #141713), and the institutional review board of the University of Washington (project #48738).

Study setup

Every student in the class was given the option to use *PeaCoq* instead of other IDEs for working on their homework. At any moment, they could opt in and out of using *PeaCoq* with no overhead. This study helped us iron out details on the automation and the display, based on students' feedback.

Study material

All the material of the class is available on the website of the instructor [1].

Study results

This study was done in order to evaluate what worked and did not work with the current implementation of *PeaCoq* at the time. Most of the takeaways are qualitative feedback from participants. Thirteen students chose to use *PeaCoq* over the other alternatives, and provided feedback on what they liked and disliked.

The results of the study were qualitative, indicating to us places where *PeaCoq* could be improved for users, through their feedback. The first issue that was raised concerned the amount of tactics presented by the automation. Users would routinely be shown dozens of tactic variations for a given context, with many tactics deemed unnecessary or redundant. They requested some pre-processing, sorting, and grouping of tactics, so that they could easily ignore tactics that they knew would not help.

Thanks to this feedback, we developed a grouping of tactics based on their action. For instance, the `destruct` tactic applies to any hypothesis in context that is built using a data type. Since contexts often grow to have many such hypotheses, the context

[1] https://courses.cs.washington.edu/courses/cse505/15wi

is often cluttered with a dozen of variations of the `destruct` tactic, before the next tactics get listed. When the user knows they are not interested in any of these, scrolling through them is tedious. A suggested alternative consists in grouping tactics by theme, offering browsing facilities per theme and across themes. This way, users not interested in `destruct` tactics can browse over the whole group in one action.

Another problem some beginners had was that some very general tactics were often confusing. For instance, *Coq* provides the `left` and `right` tactics, often presented for their use in proving logical conjunctions. However, those tactics are overloaded to work on any data type with two constructors. For instance, when the user needs to provide a `list`, they can run the `left` tactic to provide an empty list, or the `right` tactic to indicate their intent to provide a non-empty list. This behavior, while useful for certain expert proving tasts, is quite confusing for beginners. We solved this problem by overwriting those tactics to instead perform a guard check for the shape of the goal. Our modified `left` and `right` tactic first check that the goal is a syntactic conjunction, before applying the tactic. While this significantly reduces the power of the tactic, it is sufficient for beginner exercises, and provides less surprising results.

2.3.2 A-B study

The second study was conducted at the University of California San Diego, with the help of professor Sorin Lerner, during the Spring quarter of 2015. The study was approved by the institutional review board of the University of California San Diego (project #141713).

Participants in the A-B study were volunteers who received no financial compensation, but were offered free slices of pizza, a highly sought after treat in academic settings.

Study setup

The study was done in two instances, totaling 20 participants. For each instance of the study, 10 participants were chosen randomly but based on their availability, then participants were randomly paired in 5 groups. Both groups were informed that they would be testing a novel programming environment.

The 5 groups in the *control group* were provided with an instance of *PeaCoq* designed to imitate the usual IDEs for *Coq*: all the special features of *PeaCoq* were disabled, except for its syntax highlighting, and keyboard shortcuts.

The 5 groups in the *study group* were provided with an instance of *PeaCoq* with the proof tree view always enabled, visual diffs overlaid on top of nodes in the proof tree view, and automation running in the background to populate the proof tree view with candidate tactics, all enabled.

Neither group was informed about the fact that this was an A-B study, or about whether they were testing the real prototype or the control version. Both instances were structured identically: the first hour was a general presentation of the tool they would use, and of basics of the *Coq* proof assistant. Then, each pair of participant was tasked with solving 16 problems of increasing difficulty, testing their understanding of what they have seen so far, as well as their ability to learn about new proof solving tactics and use them effectively. The second part was scheduled to take up to an hour and a half.

After the study was over, and before they left, the participants were handed an anonymous questionnaire, asking them about some qualitative feedback and information about their education level.

Study material

During the first part of the study, participants were introduced, by the study coordinator, to the following concepts:

- an *inductive datatype definition* `day`, with constructors `monday`, `tuesday`, etc.,
- a *function definition* `tomorrow`, defined by pattern-matching,
- a theorem asserting that `tomorrow saturday = sunday`, proven introducing the tactics `simpl` and `reflexivity`,
- a recursive data type, `natlist`, representing a monomorphic list of natural numbers,
- a *recursive* function, `concat`, performing list concatenation,
- another theorem about the concatenation of two concrete lists, displaying an instance of associativity, and proven using the same tactics seen so far,
- a theorem asserting that `nil` is a left-unit for `concat`, proven with the previous tactics, with the addition of the `intros` tactic to introduce the universally-quantified list `l`,
- a theorem asserting that `nil` is a right-unit for `concat`, proven by introducing the `induction` tactic, and also introducing the first `rewrite`,
- a theorem asserting the associativity of the `concat` operation, proven by using `induction`, and the other tactics mentioned so far.

During the second part of the study, they had to try and solve all 16 exercises, as listed exhaustively in Appendix A.1. The exercises can be roughly described as follows:

1. `rev_snoc` : After introducing a recursive function `snoc` to append one element to the end of a list, and a recursive function `rev` to reverse a list, participants were asked to demonstrate that a sequence of `rev` after `snoc` is equivalent to a sequence of `cons` after `rev` .

2. `rev_involutive` : Participants were asked to demonstrate that `rev` is involutive (i.e. applying it twice consecutively yields the original input).

3. `concat_cons_snoc` : Participants were asked to prove an equality about the interplay of `concat` and `snoc` .

4. `go_somewhere` : A new concept was introduced: disjunction. Two tactics to manipulate this concept were introduced: `left` and `right` . An example was given of proving the disjunction of a falsehood on the left, and a tautology on the right. Participants were then asked to find the only disjunct that was a tautology within a nested disjunction of falsehoods. Finding the only tautology required uses of both `left` and `right` .

5. `B_is_enough` : Participants were introduced to the tactic `apply` , and asked to prove a disjunction where one disjunct was given in the premises, and one was not.

6. `more_facts` : Participants were introduced to the concept of conjunction. A new tactic, `split` , was then introduced to prove conjunctions. They were then asked to prove the conjunction of two tautologies.

7. `A_and_B` : To re-assert the importance of `apply` , participants were asked to prove a conjunction where each conjunct was found in the premises.

8. `snoc_concat_end` : Two harder exercises about list were then presented. The first one, asking about a more complex interplay between `concat` and `snoc` ,

9. `rev_distributes_over_concat` : the second one, asking to prove that `rev` distributes over `concat` .

10. `map_commutes` : Participants were introduced to the concept of the function `map` over lists. They then showed that, if two functions commute, then mapping these two functions also commutes.

11. `map_fusion` : Participants were asked to prove the map fusion property.

12. `fold_snoc` : Participants were introduced to the concept of a `fold` over a list. They then demonstrated an interplay between `fold` and `snoc` .

13. `map'_unroll` : Participants were asked to demonstrate that performing the operation `map` over a list obtained via `cons` can be unrolled one step.

14. `map_map'` : Participants were then shown how `map` can be implemented as a `fold` . The resulting implementation, named `map'` , was then demonstrated to be functionally equivalent to `map` . To help them, we axiomatized a small theorem that they could use without needing to prove.

15. `In_cons` : We introduced a recursive predicate, `In` , asserting the presence of an element in a list. They were first asked to prove that if an element is in a list, it is still present in a list with an additional element.

16. `In_concat_left` : Two final concepts were introduced: the `cases` tactic is a

custom tactic that let participants break a conjunction in their context into its components, and the `contradiction` tactic allowing them to point to the presence of falsehoods in the context. Participants were finally asked to prove that if an element belongs in a list, it also belongs in the result of concatenating said list with an arbitrary other list.

Table 2.1 lists, for each of the exercises, which tactics were meant to be exercised. The exercises were roughly sorted in order of difficulty, say for two points: the part introducing logical operators (exercises 4 through 7) was much simpler, and `rev_distributes_over_concat` was a challenging mid-point exercise.

Study results

Table 2.2 lists the average timings of both groups on each exercise, as well as a rough estimate of the difficulty of each exercise. Figure 2.6 reports the cumulative time spent on each exercise. Note that users were allowed to take breaks between exercises, and we do not report the time spent on the last exercises for those people who did not finish all exercises: this explains why all bars do not end at the same location. As a reminder, group A was our control group, using a version of *PeaCoq* with our features disabled, while we will refer to group B as PeaCoq *users*, since they used the version of *PeaCoq* with our features enabled.

Looking at Figure 2.6, we can notice several trends. First, *PeaCoq* users were much slower on the first exercise (`rev_snoc`). Even though it was an easy exercise, the automation offered several options, which overwhelmed the participants for their first proof attempt: we noticed several participants spent a long time scrutinizing different options (including many useless options) before committing to one. In the control group, participants were left to their own volition, and followed the previous proofs as

Table 2.1. *PeaCoq* A-B study exercises design

Exercise	What tactics were expected?										
	`simpl`	`reflexivity`	`intros`	`induction`	`rewrite`	`left`	`right`	`apply`	`split`	`cases`	`contradiction`
01. `rev_snoc`	✓	✓	✓	✓	✓						
02. `rev_involutive`	✓	✓	✓	✓	✓						
03. `concat_cons_snoc`	✓	✓	✓	✓	✓						
04. `go_somewhere`		✓					✓				
05. `B_is_enough`		✓				✓	✓				
06. `more_facts`		✓					✓		✓		
07. `A_and_B`			✓					✓	✓		
08. `snoc_concat_end`		✓	✓		✓						
09. `rev_distributes` …	✓	✓	✓	✓	✓						
10. `map_commutes`	✓	✓	✓	✓	✓						
11. `map_fusion`	✓	✓	✓	✓	✓						
12. `fold_snoc`	✓	✓	✓	✓	✓						
13. `map'_unroll`		✓	✓								
14. `map_map'`	✓	✓	✓	✓	✓						
15. `In_cons`	✓		✓				✓	✓			
16. `In_concat_left`	✓		✓	✓		✓	✓	✓		✓	✓

Table 2.2. *PeaCoq* A-B study exercises timings per group

All times are reported in seconds.

👨‍🎓 indicates how many pairs of participants finished the proof.

Exercise	Difficulty	Group A		Group B	
		Mean (Std. Dev.)	#	Mean (Std. Dev.)	#
01. `rev_snoc`	★☆☆	213 (74.6)	👨‍🎓 ×5	609 (495.8)	👨‍🎓 ×5
02. `rev_involutive`	★★☆	571 (234.6)	👨‍🎓 ×5	180 (179.1)	👨‍🎓 ×5
03. `concat_cons_sn` ...	★★☆	158 (63.6)	👨‍🎓 ×5	292 (146.0)	👨‍🎓 ×5
04. `go_somewhere`	★☆☆	24 (18.7)	👨‍🎓 ×5	14 (3.8)	👨‍🎓 ×5
05. `B_is_enough`	★☆☆	134 (122.9)	👨‍🎓 ×5	66 (31.8)	👨‍🎓 ×5
06. `more_facts`	★☆☆	34 (25.7)	👨‍🎓 ×5	27 (15.4)	👨‍🎓 ×5
07. `A_and_B`	★☆☆	49 (20.1)	👨‍🎓 ×5	21 (11.6)	👨‍🎓 ×5
08. `snoc_concat_en` ...	★★☆	104 (36.1)	👨‍🎓 ×5	156 (133.3)	👨‍🎓 ×5
09. `rev_distribute` ...	★★★	446 (437.4)	👨‍🎓 ×5	748 (540.6)	👨‍🎓 ×5
10. `map_commutes`	★★☆	348 (134.7)	👨‍🎓 ×5	189 (118.2)	👨‍🎓 ×5
11. `map_fusion`	★★☆	109 (47.8)	👨‍🎓 ×5	76 (47.9)	👨‍🎓 ×5
12. `fold_snoc`	★★☆	86 (39.2)	👨‍🎓 ×4	207 (182.8)	👨‍🎓 ×5
13. `map'_unroll`	★☆☆	471 (186.4)	👨‍🎓 ×4	13 (9.9)	👨‍🎓 ×5
14. `map_map'`	★★☆	317 (239.0)	👨‍🎓 ×3	80 (83.0)	👨‍🎓 ×5
15. `In_cons`	★☆☆	52 (undefined)	👨‍🎓 ×1	160 (93.0)	👨‍🎓 ×5
16. `In_concat_left`	★★★	492 (undefined)	👨‍🎓 ×1	791 (256.2)	👨‍🎓 ×3

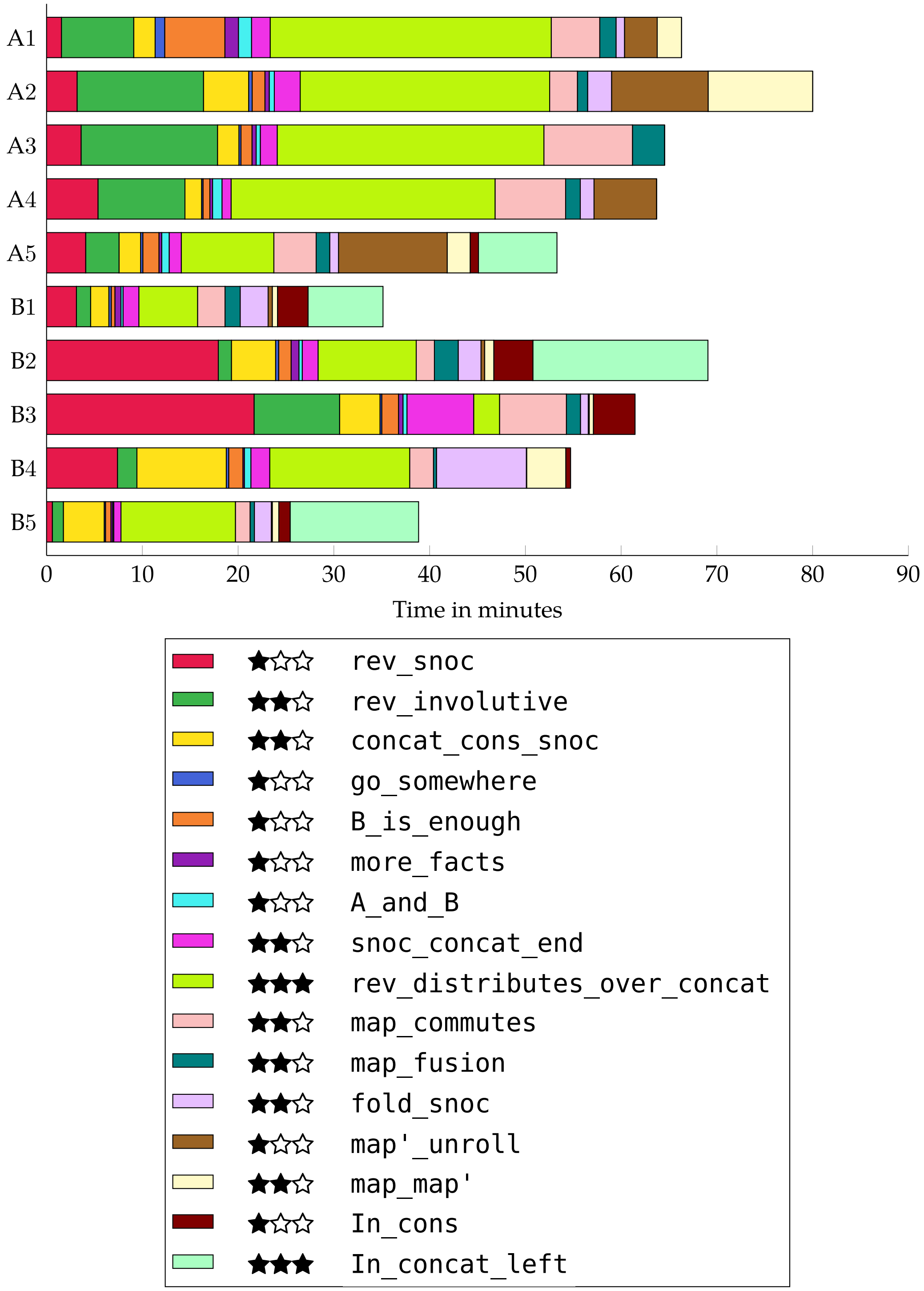


Figure 2.6. *PeaCoq* A-B study timings per participant pair

examples. Since the exercise exhibited the same pattern as the exercises demonstrated during the lesson phase, it was a good strategy.

Looking at the second exercise (`rev_involutive`), we notice the opposite trend: *PeaCoq* users seem overall faster than most participants from the control group, apart from group A5. This group remains an outlier for the whole study, and our post-study survey seems to indicate that one of its participants had a higher mastery of functional programming and type systems than the norm. For the other pairs, we have a hypothesis for the time gap. This exercise had almost the same proof structure as the previous one, except that, where the previous exercises required only *one* rewrite, with the inductive hypothesis, this exercises required *two* rewrites, first with the lemma proven in the previous exercise, and then with the inductive hypothesis. Noticing this required some perspicacity from the participants in the control group. On the other hand, *PeaCoq*'s automation would, as part of its functioning, try to apply previously-proven theorems to the current context. Participants were therefore shown that rewriting with `rev_snoc` was an option, without having to even ponder whether this was a possibility.

While both groups solved the logic exercises (exercises `go_somewhere` through `A_and_B`) fairly fast, *PeaCoq* users finished them even faster than the control group. For these exercises, *PeaCoq* only provided a handful of options that were easily browsed through, so participants of group B did not waste any time looking at confusing options.

Exercise `rev_distributes_over_concat` was one of the hardest lemmas overall. While there are several ways of proving it, the proof we expected includes a proof by induction, rewrites with lemmas from earlier exercises, and rewrites with lemmas from the tutorial. It is quite easy for a beginner to lose themselves and hesitate in such a proof. Another common mistake is to try and prove those lemmas in this larger context, which often results in failure. The data suggests that *PeaCoq* users did much better than all control groups participant, except for the outlier. We believe, again, that this is the result of *PeaCoq* suggesting those rewrites with earlier lemmas, as opposed to control group participants who needed to recall their existence from memory, or by browsing back through the proof file.

Exercise `map'_unroll` exhibits one of the largest disparities between the control group and the *PeaCoq* user group. This is the case of an exercise with a trivial solution, but realizing that it is the case is the hard part. The exercise asks the participants to prove the following equality:

```
map' f (cons x xs) = cons (f x) (map' f xs)
```

where `map'` is a re-definition of the original function `map` over lists, but as a `fold` operation, whereas the original uses explicit recursion. The two sides of this equation are judgmentally equal, which means that a simple call to the `reflexivity` tactic solves this goal. Unfortunately, because of the rules that govern how `Fixpoint` s are unfolded in *Coq*, the goal can not easily be simplified to look like a simple equality. Users in the control group were therefore stuck, until we gave them a hint to use the `unfold` tactic to turn the goal into:

```
fold (fun (x0 : nat) (fxs : natlist) => cons (f x0) fxs)
  (cons x xs) nil =
  cons (f x)
    (fold (fun (x0 : nat) (fxs : natlist) =>
             cons (f x0) fxs) xs nil)
```

which is still scary, but from which one can call `simpl` to finally obtain:

```
cons (f x) (fold (fun (x0 : nat) (fxs : natlist) =>
                   cons (f x0) fxs) xs nil) =
cons (f x) (fold (fun (x0 : nat) (fxs : natlist) =>
                   cons (f x0) fxs) xs nil)
```

The identical structure of both sides is finally more evident. Of course, *PeaCoq* users were almost immediately presented with the `reflexivity` tactic, as a result of our automation, and thus solved the goal immediately. While this had a significant *positive* impact on their timings for this exercise, we also believe that it had a significant *negative* impact on their understanding of this proof. While the proof assistant could convince itself that the two terms were equal, we doubt that participants understood the computation rules that made this correct.

Looking towards the right side of Figure 2.6, we can see that users of the control group did not reach the last two exercises, except for the outlier group A5. On the other hand, all *PeaCoq* users reached the final exercise (as can be witnessed by them finishing the penultimate exercise), and 3 out of 5 groups also solved the final exercise in the allotted time.

As for the post-study survey, all the answers provided by participants are anonymously catalogued in Appendix A.2.

One of the question asked was *Did you understand all the proofs you completed?*. About 6 out of 10 participants of the control group self-reported understanding all the proofs they completed, and about 5-to-6 out of 10 participants of the *PeaCoq* group. However, users of the *PeaCoq* group expressed more restraint about their answer, and explicitly mentioned that some proofs felt “magical”.

We also included questions about the effect of different tactics, in an attempt at evaluating their understanding in a way that is not self-reported. We summarize our assessment of their answers in Table 2.3. Overall, the understanding of tactics across groups is very good, with a small advantage for the control group. In particular, participants using *PeaCoq* did not quite realize what the `split` tactic did. We surmise that, because the tool suggested the tactic promptly, this group did not spend as much time reflecting on what the tactic was actually doing.

Table 2.3. *PeaCoq* A-B study tactic understanding

This table shows whether the semantics of a given tactic was explained convincingly.
Correct answers are indicated by a check mark ✓.
Incorrect answers are indicated by a cross mark ✗.
Approximate answers are indicated by a tilde $\sim$.
Empty cells indicate empty answers.

	`split`	`intro`	`induction`	`reflexivity`	`rewrite`	`right`
A1_1	✓	✓	✓	✓	$\sim$	✓
A1_2	✓	✓	✓	✓	✓	✓
A2_1	✓	✓	✓	✓	✓	✓
A2_2	✓	$\sim$	✓	✓	✓	✓
A3_1	✓	$\sim$	✓	✓	$\sim$	✓
A3_2	$\sim$	✓	✓	✓	✓	$\sim$
A4_1	✓	✓	✓	✓	✓	✓
A4_2	✓	$\sim$	✓	✓	✓	✓
A5_1	✓	✓	✓	✓	✓	✓
A5_2	✓	✓	✓	✓	✓	✓
B1_1	✓	✓	✓	✓	✓	✓
B1_2	$\sim$	✓	✓	✓	✓	✓
B2_1	✗	✓	✓	✓	$\sim$	✓
B2_2	✓	✓	✓	✓	✓	✓
B3_1	✗	$\sim$	✓	✓	✓	✓
B3_2	✓	✓	✓	✓	$\sim$	✓
B4_1	✗	✓	✓	✓	✓	✓
B4_2			✓	✓	✓	✓
B5_1	✓	✓	✓	✓	✓	✓
B5_2	✓	✓	✓	✓	✓	✓

Takeaways

This study highlighted the following:

- It appears that *PeaCoq* helps beginners go through beginner-level proofs faster than using the vanilla proof assistant.

- However, this speed sometimes comes at the price of understanding what is happening. Because *PeaCoq* alleviates too much of the mental effort, users are less challenged, resulting in them being able to finish exercises without insight.

Threats to validity

We provide a list of threats to validity of our results, in no particular order:

- The sample size was very small (20 participants total, 10 per instance). Unfortunately, the groups also had a very high variance, as can be seen by looking at the standard deviations. Often times, skilled groups finished exercises extremely fast, while struggling groups spent very long amounts of time stuck.

- The sample was biased towards graduate students, with only 3 undergraduate students (1 in the A group, 2 in the B group).

- The study was fast-paced, with only one hour to learn the rudiments of theorem proving, and an hour and a half to solve exercises. A typical *graduate* programming language course would most likely cover the topics we saw in three to five lectures.

- A bug in *PeaCoq* slowed down some pairs in group B: they had to wait for us to come fix it, and reload the page to go back to where they initially were.

- Participant $B4_2$ vocally explained their dislike for mathematical logic. This can be seen reflected in Appendix A, where their ratings were significantly lower than everyone else's. We might expect better ratings from users who are willingly trying to learn, but we should also expect similar ratings from users who are forced to learn, for instance in a course setting.

Chapter 3

Chick: a core dependently-typed language and its repair algorithm

Another pain point that both beginners and experts encounter frequently is that of refactoring existing programs and proofs. Often times, one might realize, while working on a proof, that an earlier data type must be updated to add additional information. One might also need to update existing data types and functions when they want to extend an existing formalization. In both situations, after updating their definition, the programmer must perform a series of corrections to the rest of their code base to account for the newly-modified definitions. Many of those changes are mechanical, structural changes that require no knowledge of the program domain, and no creative input from the programmer. We would like to lower the burden on the programmer for such maintenance tasks, only leaving the complex parts of the refactoring to them.

This chapter will focus on the development and evaluation of a tool (named *Chick*[1]) whose purpose is to help functional programmers propagate changes, made to some of their definitions, to the rest of their code.

[1] All code for *Chick* is publicly available at: https://github.com/Ptival/chick

Section 3.1 gives background specific to this chapter.

Section 3.2 describes the global design of the tool and its components.

Section 3.3 covers the syntax of the language over which *Chick* operates.

Section 3.4 describes a family of data types that allow us to perform repairs.

Section 3.5 will present both usual lookup rules, and *Chick*-specific lookup rules, that are necessary to describe the repair algorithm.

Section 3.6 describes all the components of the repair algorithm.

Section 3.7 goes into more details into how some diff operations needed by the repair algorithm can be automatically derived from small descriptions of their reactions to certain changes in the data structure they operate on.

Section 3.8, finally, describes how we automatically infer diffs from pairs of programs without requiring additional user input.

3.1 Background

We discussed the concepts of dependent types and *refactoring* in Chapter 1. In the concept of a dependently-typed language, the problem of refactoring is the subject of an interesting tension.

On the one hand, a dependent type system allows the user to describe the types of the objects of discourse at a very high level of expressiveness. This could lead to two advantages. First, if the types are indeed more precise, maybe the refactoring algorithm could benefit from the additional, tighter information provided by them. Second, because the types are more precise, the type system may enforce more strict requirements, so that failures of the refactoring algorithm could be less dramatic, if they end up caught by the type system.

On the other hand, because the types are stronger, the terms often end up being more complex. For instance, simple functions from the standard library of languages like *Haskell* and *OCaml* might be encoded using more complex type-level machinery in a dependently-typed language. Because of this, the refactoring algorithm is confronted with more complex problems, which are often much less tractable than their simply-typed and polymorphically-typed counterparts.

In our preliminary research, we found out only a few refactoring techniques for functional languages (mostly Li et al. [23]), and barely any work on refactoring with dependent types (apart from the very recent Ringer et al. [31]). We would like to build on top of the existing body of work for refactoring functional programs, but in the context of a dependently-typed language. In particular, we would like to support the following refactoring scenarios:

- any modification to the declaration of an inductive data type should be propagated throughout the program,
- any modification to the type of a function should be propagated throughout the program,

In the case of a proof assistant with a tactic language, it would be best to also be able to propagate changes through proof scripts written using this tactic language. While we do not contribute such changes in this dissertation, we demonstrate a path towards this goal. Unfortunately, achieving such a goal would likely require developing a formal semantics for the tactic language. The only known attempt at formalizing the semantics of *Ltac* that the author is aware of is in Jedynak [17], and it only formalizes a very small, simplified fragment of it, which is already quite an achievement. We don't believe that anyone is working on such a formalization at the moment, and in fact, the tactic language itself is currently being revised by the *Coq* maintainers.

Any attempt at refactoring tactics would require work similar to the one we are presenting as a prerequisite, so we consider our work as a stepping stone towards this ambitious goal.

3.2 Design of *Chick*

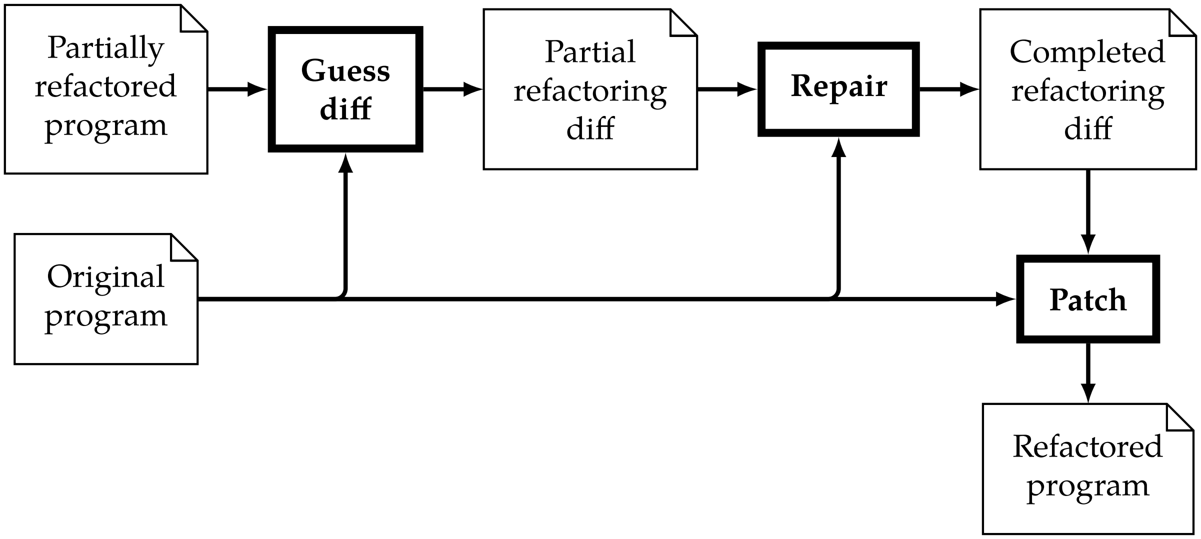


Figure 3.1. *Chick*'s workflow

In order to help programmers in dependently-typed languages refactor their programs, we built a prototype tool called *Chick*. The high-level workflow of *Chick* is given in Figure 3.1. The main idea is to take the original, existing program prior to the refactoring attempt, and the modified program where a partial refactoring has been performed, and automatically finish propagating changes through the rest of the program. However, we do not perform this operation all at once. Instead, our workflow consists in three phases.

First, we analyze the original program against the modified program to build a list of all the differences between the two. We will describe the data types used to represent those differences in Section 3.4, but will defer explanations about how this guess is performed until Section 3.8. We try to capture the intent of the programmer in this list of differences; as we will discuss, there can be several ways of interpreting the same change, and the repair performed will depend on which interpretation is chosen.

Second, this list of differences is used to compute a repair of the program. Here again, we do not directly compute a repaired program (a value of type `program`), but instead compute a value of type Δ_{Program} (as described in Section 3.4) describing how the program must be modified in order to be repaired. We do so because the diff contains more information about the changes than the repaired program, as we will demonstrate in Section 3.4.6.

Third and finally, the repaired program is obtained by simply applying the repaired diff to the original program.

Figure 3.2 gives a concrete example of the kind of repair we want to propagate. We will use it as our running example in the following sections. On the left-hand side is an original program, prior to any attempt at refactoring it. In this example, the user defined a data type `list`, and a couple of definitions and functions over this data type. Let us now assume they would like to replace this type list with a richer type of length-indexed lists. We chose this example because it brings up many of the feature we would like to have. On the right-hand side of the same figure, the orange part is the partial refactoring the user has manually done. The green parts on the right-hand side constitute the rest of the refactoring, as automatically produced by our tool.

```
Inductive nat : Set :=
| O : nat
| S : ∀ (n : nat), nat.

Inductive list (A : Type)
: Type :=
| nil : list A
| cons : A →
    list A → list A.

Definition a_list
: list nat :=
  cons nat (S (S (S O)))


    (nil nat).

Definition length :
  ∀ (T : Type),
    list T → nat :=
  λ T l, list_rect T (λ _, nat) O
    (λ _ _ lt, S lt) l.

Fixpoint map :
  ∀ (A B : Type), (A → B) →
    list A →
    list B :=
  λ _ B f l,
    match l with
    | nil _      => nil B
    | cons _ h t =>
      cons B (f h)
           (map A B f t)
    end.
```

```
Inductive nat : Set :=
| O : nat
| S : ∀ (n : nat), nat.

Inductive vec (A : Type)
: nat → Type :=
| vnil : vec A O
| vcons : A → ∀ (n : nat),
    vec A n → vec A (S n).

Definition a_list
: vec nat (_ : nat) :=
  vcons nat (S (S (S O)))
    (_ : nat)
    (vnil nat).

Definition length :
  ∀ (T : Type),
    vec T (_ : nat) → nat :=
  λ T l, vec_rect T (λ _ _, nat) O
    (λ _ _ _ lt, S lt) l.

Fixpoint map :
  ∀ (A B : Type), (A → B) →
    vec A (_ : nat) →
    vec B (_ : nat) :=
  λ _ B f l,
    match l with
    | vnil _          => vnil B
    | vcons _ h _ t =>
      vcons B (f h) (_ : nat)
           (map A B f t)
    end.
```

Figure 3.2. Running example of a *Chick* repair. The user-provided partial refactoring is highlighted in orange. The computed repairs are highlighted in teal.

In the `a_list` definition, the type must be renamed, and a type index must be added. While the repair algorithm knows that an index must be inserted, it does not know what value to use: here, because the length is static, the right value would be `S O`, but in more complex cases (as in the next function, `length`), the value can be a new variable, or the result of an arbitrary computation, so guessing what it should be is a hard problem. Instead, we simply insert a typed hole `(_ : nat)`. The user might need to eventually provide a concrete value of type `nat`, though, there are cases where the type system will infer such value without any user assistance. The term of the definition was also updated, to rename the constructor and add a type hole for its new argument. While this change seems trivial, it relies on our ability to match constructors from the old inductive definition to constructors in the new inductive definition. In order to guess that `nil` and `vnil` are related, and that `cons` and `vcons` are related, our analysis must evaluate the most likely pairs of matching constructors. Since a user might also have removed a constructor, and created an entirely different one, we also want to assess our confidence in matching two constructors.

The definition of `length` is updated in a similar way, but notice that the function call being updated is a call to the *eliminator* `vec_rect`. We will describe eliminators later, in Section 3.7, but for now, all we need to highlight is that this eliminator is never explicitly defined in the user code: it is a function that is automatically generated when the `vec` definition is accepted. Our tool supports patching such functions, as long as we describe to it how those functions change when the inductive definition they are generated from changes.

Finally, in the definition of `map`, the patterns of a `match` construct are updated. This will require some care since it may introduce or remove binders, but for this example, our repair algorithm took the conservative approach of not binding the new arguments, using a wildcard `_` pattern.

3.3 Syntax of *Chick*

This section will cover the syntax of *Chick*. Section 3.3.1 describes the syntax of the terms of the language, a language similar to *Coq*'s *Gallina*. Section 3.3.2 describes the syntax of the host language, a language similar to *Coq*'s *Vernacular*, within which functions and data types can be defined.

3.3.1 Syntax of *Chick* terms

Chick is designed as a small functional programming language upon which the repair algorithm will operate. The aim was to represent a language such as *Gallina*, and as such, *Chick* is a dependently-typed lambda calculus, with inductive constructions. The abstract syntax of *Chick* terms is given in the following grammar:

⟨*term*⟩ ::=

| ⟨*var*⟩ (variable)

| ⟨*term*⟩ ⟨*term*⟩ (function application)

| `λ` ⟨*binder*⟩ `,` ⟨*term*⟩ (term abstraction)

| `Π(` ⟨*binder*⟩ `:` ⟨*term*⟩ `)` `→` ⟨*term*⟩ (type abstraction)

| ⟨*universe*⟩ (universes)

| `match` ⟨*term*⟩ `with` $\overline{\langle \mathit{pattern} \rangle}$ `end` (pattern matching)

| ⟨*term*⟩ `:` ⟨*term*⟩ (type annotation)

| `_` (hole)

⟨*universe*⟩ ::= `Prop` | `Set` | `Type`

⟨*binder*⟩ ::=

| ⟨*var*⟩ (named)

| `_` (anonymous)

$\langle \textit{pattern} \rangle ::= (\overline{\langle \textit{binder} \rangle}, \langle \textit{term} \rangle)$

The *function application*, *value abstraction*, *type abstraction*, and *variable* constructs, are standard constructs of a dependently-typed lambda calculus.

The *type annotation* construct lets us ascribe a type to a given term. The *hole* construct can take the place of a term anywhere: it stands for a missing value that should be filled by the programmer. Together, these two constructs let us have *typed holes*, that is, placeholder values whose type is known. This proves invaluable throughout the repair algorithm, as we will see, since there are situations where the algorithm must come up with arbitrary values, knowing only the type they should bear. In the absence of synthesis techniques to generate such values, the algorithm can safely insert a typed hole, leaving to the programmer the task of figuring out the proper value.

Finally, the *pattern matching* construct is found in languages with algebraic data types, and lets us dispatch code based on the shape of some such data type. The attentive reader may notice that our pattern language is somewhat limited. We do not cover nested patterns, or other advanced pattern features.

We use the same universe names as found in *Gallina*, though we do not yet support its cumulative universe hierarchy syntactically. This does *not* mean that *Chick* cannot repair *Gallina* programs that use the universe hierarchy, but rather, that it cannot syntactically represent universe levels. Instead, *Chick* is oblivious to universe levels, so it will repair programs regardless of their universe level. The downside is that, because it cannot represent universe levels explicitly, it might break programs that require them. Bare support for universe levels can be added almost for free, since the repair algorithm could simply carry them along and attempt no repair. A universe-aware repair algorithm would require further investigation.

About underscores

While we use the underscore symbol (_) to represent both a hole term and an anonymous binder, the two concepts can never appear in the same program location, and therefore, there can be no confusion: in a term context, the symbol represents a hole, while in a binding context, it represents an anonymous binder.

3.3.2 Syntax of *Chick* programs

Our programs are defined as sequences of commands in the following vernacular (to borrow Coq's parlance):

$\langle \textit{program} \rangle$ ::= $\overline{\langle \textit{vernacular} \rangle}$

$\langle \textit{vernacular} \rangle$ ::=

| `Definition` ({ kind : ⟨*definition-kind*⟩ , name : ⟨*var*⟩, type : ⟨*term*⟩, body : ⟨*term*⟩ })

| `Inductive` (⟨*inductive*⟩)

⟨*definition-kind*⟩ ::= `Definition` | `Fixpoint`

Our *definition kinds* follow *Coq*'s *Vernacular* conventions: a `Definition` is *never* recursive, while a `Fixpoint` is *allowed* to be recursive. Declarations are ordered, and may only depend on previous declarations. This will ease the task of our repair algorithm, since dependencies are syntactically ordered. In a language where declarations are allowed to depend on other declarations backward *and* forward, we would want to repair programs according to the dependency tree rather than the syntactic order of declarations.

Inductive definitions are not mutual, and follow the following syntax:

$\langle \textit{inductive} \rangle$ ::= `Inductive` ({

name : $\langle \textit{var} \rangle$,

parameters : $\overline{(\langle \textit{var} \rangle, \langle \textit{term} \rangle)}$,

indices : $\overline{(\langle \textit{binder} \rangle, \langle \textit{term} \rangle)}$,

universe : $\langle \textit{universe} \rangle$,

constructors : $\overline{\langle \textit{constructor} \rangle}$

})

$\langle \textit{constructor} \rangle$ ::= `Constructor` ({

name : $\langle \textit{var} \rangle$,

parameters : $\overline{(\langle \textit{binder} \rangle, \langle \textit{term} \rangle)}$,

indices : $\overline{\langle \textit{term} \rangle}$

})

For a presentation of inductive data type declarations, we refer the reader to Coquand and Paulin [7] for an academic description, or to Pierce et al. [29] for some educational material. We will briefly recall the difference between *inductive parameters*, *inductive indices*, *constructor parameters*, and *constructor indices*.

Inductive parameters indicate that a type is generic in some input type. Most often, this type parameter will be a type that the constructors may contain instances of, or operate on values of. For instance, many containers will have a carrier type for the type of values they may contain. The choice of type to be contained does *not* restrict the shapes that the container may take: we say that it behaves *parametrically*. For instance, the type of list of boolean values, `list bool`, and the type of list of natural number values, `list nat`, contain the same “shapes”: `[]`, _ :: `[]`, _ :: _ :: `[]`, etc. The choice of the type parameter only tells us what values we are allowed to put in those holes, but does not dictate which of those shapes we may or may not instantiate.

On the other hand, *inductive indices* can affect what inhabitants we may find in an inductive type. Instead of being called an inductive type, one that is indexed is often instead called an *inductive family of types*, or *inductive type family*, to capture the idea that the choice of index has an effect on what inhabitants may exist. Knowing the index of an inductive value gives us information about what shape it may have. For instance, we could build a type of lists indexed by whether their length is odd or even as:[2,3]

```
Inductive parity : Type :=
| Even : parity
| Odd  : parity
.

Definition next_parity (p : parity) : parity :=
  match p with
  | Even => Odd
  | Odd  => Even
  end.

Inductive parity_list (T : Type) : parity → Type :=
| []   : parity_list T Even
| (::) : ∀ (h : T) {p : parity} (t : parity_list T p) →
        parity_list T (next_parity p)
.
```

Now, if we consider the type family `parity_list T p` as a whole, for a given choice of carrier type `T`, the possible shapes of values are still the same as for `list`: `[]`, `_ :: []`, `_ :: _ :: []`, etc. However, for a given choice of `p`, the valid shapes become restricted by construction. The type `parity_list T Even` only contains shapes `[]`, `_ :: _ :: []`, `_ :: _ :: _ :: _ :: []`, etc. Dually, the type `parity_list T Odd` only contains shapes `_ :: []`, `_ :: _ :: _ :: []`, etc.

[2] *Coq* syntax does not let us define constructors that look like operators, but we do it here to keep the code clear and concise.

[3] In constructor `(::)`, we use curly braces `p` to mark the constructor parameter as implicit, indicating it can be implicitly inserted unambiguously by Coq when the next constructor parameter, `t`, is supplied.

In general, different families may have entirely disjoint shapes, as presented here, or partially-overlapping shapes (for instance, the type family of finite sets), or even all the same shapes (in which case, the index might be used as a type-level marker for some information, a technique often called `phantom type` in statically-typed languages).

Constructor parameters are just arguments that are passed to a given constructor. They are completely independent for each constructor, and are simply used to store data relevant to the given constructor. For instance, in our `parity_list` example, `h`, `p`, and `t` are three constructor parameters for constructor `(::)`.

The return type of a constructor must indicate in what family is exists. To do so, it must provide values for the inductive indices. We call those values the *constructor indices*. For instance, in our `parity_list` example, the inductive index of type `parity` is instantiated with value `(next_parity p)` in the `(::)` constructor. Note that it is possible to have indices be computed values, as is the case here.

Finally, let us discuss scope for those constructs.

- Following *Coq*'s convention, inductive parameters *must* be named. They are naturally in scope for the whole definition of the inductive type.

- Inductive indices *may* be anonymous, or named, and they form a telescope (as described in Section 1.2). They are *not* scoped over the rest of the definition. Therefore, the only purpose for named indices is to create dependent indices.

- Constructor parameters *may* be anonymous, or named. They also form a telescope all the way to the constructor's return type, such that both subsequent parameters, as well as the constructor's indices, may refer to them.

- Constructor indices *must* be passed as explicit arguments to the return type, and as such, there is no binding involved.

3.4 Describing program modifications with diffs

We define a family of data types that allows us to describe changes made to terms from the language presented in Section 3.3.1, as well as programs from the language presented in Section 3.3.2. We will refer to these descriptions of changes as *diffs* [4]. We will usually denote a *diff type* Δ_τ if it corresponds to values of the diffed type τ. Informally, one can think of the type Δ_τ as a descriptor for how some value v_1 of type τ can be transformed into another value v_2.

It is important to clarify how we intend to use those diff types. We will want to describe changes made to the data types in the abstract syntax tree (*AST*) of the user's program. An example will help us avoid some misconception: consider the user program and its modification in Figure 3.3. While, from the program point of view, a value of type `bool` was modified from the value `True` to the value `False`, we will *not* use the type Δ `bool` to describe this change! From the point of view of the abstract syntax tree, a value of type `term` has changed from the original value `Var "True"` to the value `Var "False"`, which we will capture using the diff type Δ `Term`.

To be precise, we will have to describe how a `Definition` has changed, and describe how its name has remained `a`, its type has remained `bool`, but its definition has changed. This will be described as a value of type Δ `Vernacular`, containing a diff for each of those three components that may change, including the one of diff type Δ `Term`.

[4] Our approach is very similar to that of Miraldo et al. [26], though it was derived independently from their work, at around the same time they were publishing it.

```
Inductive bool : Set :=
| True  : bool
| False : bool.

Definition a : bool := True.
```

```
Inductive bool : Set :=
| True  : bool
| False : bool.

Definition a : bool := False.
```

Figure 3.3. A simple program and its modification

When the user makes a change to their program, we will attempt to guess the structure of their changes, as a value of one of the program diff type $\Delta_{\texttt{Program}}$. To illustrate the kind of changes we are interested in describing, let us look back at our motivating example in Figure 3.2, where we can observe the following partial attempts at refactoring:

1. they renamed `list` into `vec`,
2. added an index of type `nat`,
3. renamed constructor `nil` into `vnil`,
4. instantiated the index for the first constructor with `O`,
5. renamed constructor `cons` into `vcons`,
6. added a parameter `n` of type `nat` to the second constructor,
7. updated the recursive occurrence's name,
8. updated the recursive occurrence's index,
9. and instantiated the index for the second constructor with `(S n)`.

Intuitively, we want to capture changes like insertion, modifications, deletions, and permutations, at all syntactic levels of the meta-language (within terms, within inductive declarations, etc.). We will use the same descriptions to capture user-provided and repair-generated modifications.

Every diff type is accompanied by a patching function. Given an element of the diffed type, say $(x : \chi)$, and an element of the diff type for that particular type, say $(\delta_x : \Delta_\chi)$, the patching function produces, when successful, a patched element (with the same type as the original one), say $(x' : \chi)$. This operation is partial for many diff types, because they often capture modifications that only make sense for certain constructors of the diffed type: for instance, a diff stating that the head of a list has been modified can not be meaningfully applied to an empty list. We will overload the notation $x \overset{\delta_x}{\rightsquigarrow} \boxed{x'}$ to indicate that x' is the (optional) value obtained when (successfully) patching x according to the diff δ_x, using the relevant patching function for that diff type. In all of our notations, we use a black frame and a teal highlight to indicate that a value is an output.

3.4.1 Atomic diff

There are many data types for which we will only capture changes at an atomic granularity: either the value is the same in the new program, or it has been replaced with a different, unrelated value. For instance, a binder can either have the same name, or have been renamed. Similarly, the only possible change to the recursive flag of a definition is to be atomically changed to a different value. The parameterized diff type Δ_{Atomic} captures such cases for a given type τ:

$\langle \Delta_{\text{Atomic}}\ \tau \rangle ::=$
$\mid \quad \mathbb{1}$ (unchanged)
$\mid \quad \mathbb{K}(\langle \tau \rangle)$ (replaced)

with the following semantics:

$$\frac{}{x \overset{\mathbb{1}}{\rightsquigarrow} \boxed{x}}\ \textsc{Identity} \qquad\qquad \frac{}{x \overset{\mathbb{K}(y)}{\rightsquigarrow} \boxed{y}}\ \textsc{Replace}$$

Notation

We chose this notation to be reminiscent of the identity function (for $\mathbb{1}$), and the constant function (for $\mathbb{K}$). However, these are not functions, but simply inert constructors.

3.4.2 List diff

We will now describe our diff type for lists. *Again*, the subject of discourse is *not* lists as they appear in the user's program, but lists of abstract syntax tree constructs, as they appear in the AST of said program. We refer the reader to Section 3.3.1 and Section 3.3.2, to remind themselves of the nature of such lists and their prevalence: the patterns of a `match` construct, the parameters and indices to an inductive type, as well as programs themselves, are all instances of lists of syntactic constructs.

We provide a rich selection of diff operations for lists. The aim is not to have a canonical representation, but rather to capture closely the intent of the user modifications. We give the abstract syntax for these operators as prefix and infix operators, following a visual intuition of what happens to a list `h :: t`, though the reader is encouraged to read the semantics, immediately following, in order to understand the intent of each construct, as it will probably feel opaque on first glance. We also provide a concrete example of a diff in Section 3.4.5 that uses many of these constructs.[5].

$\langle \Delta_{\text{List}}\ \tau\ \Delta_\tau \rangle ::=$

$\mid$	$\langle \Delta_{\text{Atomic}}\ \tau \rangle$	(atomic modification of the whole list)
$\mid$	$\langle \tau \rangle \overset{\text{Ins}}{::} \langle \Delta_{\text{List}}\ \tau\ \Delta_\tau \rangle$	(insert a head)
$\mid$	$\langle \Delta_\tau \rangle \overset{\text{Mod}}{::} \langle \Delta_{\text{List}}\ \tau\ \Delta_\tau \rangle$	(modify and keep the head)
$\mid$	$\overset{\text{Drop}}{::} \langle \Delta_{\text{List}}\ \tau\ \Delta_\tau \rangle$	(drop the head)
$\mid$	$\overset{\overset{p}{\rightleftarrows}}{::} \langle \Delta_{\text{List}}\ \tau\ \Delta_\tau \rangle$	(permute according to a permutation p)

with the following semantics:

$$\frac{l \overset{\delta_l}{\rightsquigarrow} \boxed{l'}}{l \overset{h \overset{\text{Ins}}{::} \delta_l}{\rightsquigarrow} \boxed{(h :: l')}}\ \textsc{Insert} \qquad \frac{h \overset{\delta_h}{\rightsquigarrow} \boxed{h'} \qquad t \overset{\delta_t}{\rightsquigarrow} \boxed{t'}}{(h :: t) \overset{\delta_h \overset{\text{Mod}}{::} \delta_t}{\rightsquigarrow} \boxed{(h' :: t')}}\ \textsc{Modify}$$

$$\frac{t \overset{\delta_t}{\rightsquigarrow} \boxed{t'}}{(h :: t) \overset{\overset{\text{Drop}}{::} \delta_t}{\rightsquigarrow} \boxed{t'}}\ \textsc{Drop} \qquad \frac{(h_{p(1)} :: \ldots :: h_{p(|p|)} :: t) \overset{\delta}{\rightsquigarrow} \boxed{l}}{(h_1 :: \ldots :: h_{|p|} :: t) \overset{\overset{\overset{p}{\rightleftarrows}}{::} \delta}{\rightsquigarrow} \boxed{l}}\ \textsc{Permute}$$

Note that we defined the semantics of $\overset{\text{Mod}}{\Pi}$ so that it both modifies and keeps the head: the recursive occurrence in rule Modify, δ_t, therefore applies to the tail t and not the whole list after the head has been repaired. On the other hand, the recursive occurrence in rule Permute, named δ, targets the entire list after the permutation is performed, not solely the tail: this is necessary because we will want to perform modifications of elements after having shuffled them around.

[5] Note that we omit the constructor for the atomic list diff, even though our implementation requires it to lift values of the atomic diff type to the corresponding list diff type.

3.4.3 Term diff

The diffs for terms include atomic changes, as well as insertion, modification, deletion, and permutation of most constructors. We illustrate a couple of these:

$$
\begin{array}{lll}
\langle \Delta_{\text{Term}} \rangle ::= & & \\
\mid & \langle \Delta_{\text{Atomic}}\ t \rangle & \text{(atomic modification)} \\
\mid & \langle \Delta_{\text{Term}} \rangle \overset{\text{Ins}}{\$} \langle \Delta_{\text{Term}} \rangle & \text{(insert application)} \\
\mid & \overset{\text{Ins}}{\lambda}\ \langle v \rangle, \langle \Delta_{\text{Term}} \rangle & \text{(insert value abstraction)} \\
\mid & \overset{\text{Ins}}{\Pi}\ (\langle v \rangle : \langle \Delta_{\text{Term}} \rangle), \langle \Delta_{\text{Term}} \rangle & \text{(insert type abstraction)} \\
\mid & \ldots & \text{(other insertions)} \\
\mid & \ldots & \text{(removals/modifications/permutations)}
\end{array}
$$

Note that we use an infix dollar sign (\$) as a symbol for function application in our diffs, even though we use an infix space for function application in our terms, which is not ideal, but should help with readability. For our purpose, we biased the diffs on binary operations in the least surprising way: deleting a function application keeps its left child, i.e. removes the function call and keeps the function (Rule Rm-App), while deleting a Π keeps its right child, i.e. removes the value being quantified but keeps the return type (Rule Rm-Pi). For insertion, we allow maximal flexibility by passing the entire old term to both recursive occurrences: for instance, the diff $(\mathbb{1} \overset{\text{Ins}}{\$} \mathbb{1})$ turns any term t into the self-application $(t\ t)$ (Rule Ins-App). However, it is often the case that only one recursive occurrence will use the original term, while the other ones will replace it: for instance, the diff $(\mathbb{1} \overset{\text{Ins}}{\$} \mathbb{K}(x))$ turns any term t into the application $(t\ x)$.

$$\dfrac{f \overset{\delta}{\rightsquigarrow} \boxed{f'}}{f\ x \overset{\overset{\text{Drop}}{\$}\ \delta}{\rightsquigarrow} \boxed{f'}}\ \textsc{Rm-App} \qquad \dfrac{\tau_2 \overset{\delta}{\rightsquigarrow} \boxed{\tau_2'}}{\Pi(x : \tau_1) \to \tau_2 \overset{\overset{\text{Drop}}{\Pi}\ \delta}{\rightsquigarrow} \boxed{\tau_2'}}\ \textsc{Rm-Pi}$$

$$\dfrac{t \overset{\delta_1}{\rightsquigarrow} \boxed{t_1} \qquad t \overset{\delta_2}{\rightsquigarrow} \boxed{t_2}}{t \overset{\overset{\text{Ins}}{\delta_1\ \$\ \delta_2}}{\rightsquigarrow} \boxed{t_1\ t_2}}\ \textsc{Ins-App}$$

3.4.4 Other diff types

We also need diffs for many other internal data types. Diff types for tuples are derived from diff types of their constituents in a straightforward way. Inductive data type definitions, as well as constructor definitions, behave essentially like a tuple of all their arguments, so their diff type is derived accordingly.

3.4.5 Example of a program diff

With all of this machinery, we can define the original diff for our running example (again, referring to the changes seen on Figure 3.2). The diff is a value of type $\Delta_{\text{Inductive}}$, as shown in Figure 3.4. Because the diffs for the constructors take a lot of space, they are abbreviated as δ `nil` and δ `cons` , and shown separately in Figures 3.26 for `nil` and 3.6 for `cons` .

While the inductive diff should be somewhat straightforward, the reader might be surprised by δ `nil` (in Figure 3.26) *not* mentioning the renaming of `list` (on the left) into `vec` on the right. While this renaming appears syntactically in the concrete

We demonstrate a diff between the following two programs. The constructors are elided, and their diff (δ `nil` and δ `cons`) is shown in Figures 3.26 and 3.6 respectively.

```
Inductive nat : Set :=
| O : nat
| S : ∀ (n : nat), nat.

Inductive list (A : Type)
: Type := …

… (* rest of first program *)
```

```
Inductive nat : Set :=
| O : nat
| S : ∀ (n : nat), nat.

Inductive vec (A : Type)
: nat → Type := …

… (* identical *)
```

$\mathbb{1}_{\text{Inductive}} \overset{\text{Mod}}{::}$	do not modify inductive `nat`
$\delta_{\text{Inductive}}($	modify `list` into `vec`
$\quad \mathbb{K}($ `vec` $),$	modify the name
$\quad \mathbb{1}_{\text{List}},$	keep the parameter
$\quad$ `nat` $\overset{\text{Ins}}{::} \mathbb{1}_{\text{List}},$	add the `nat` index
$\quad \mathbb{1}_{\text{Universe}},$	keep the universe
$\quad \delta$ `nil` $\overset{\text{Mod}}{::} \delta$ `cons` $\overset{\text{Mod}}{::} \mathbb{1}_{\text{List}}$	modify the constructors (elided)
$) \overset{\text{Mod}}{::}$	
$\mathbb{1}_{\text{List}}$	do not modify the rest of the program

Figure 3.4. Diff for our running example (constructors elided)

syntax, it is only a surface-level syntactic requirement that constructors repeat the type they inhabit. The abstract syntax tree only contains the list of indices, and does *not* repeat the name of the inductive type, nor the parameters of the inductive type, in every constructor.

… `| nil : list A`	… `| vnil : vec A 0`

$\delta\,\texttt{nil} :=$	
$\quad\delta_{\text{Constructor}}($	modify the `nil` constructor
$\qquad\mathbb{K}(\,\texttt{vnil}\,),$	modify constructor name
$\qquad\mathbb{1},$	no change to parameters
$\qquad\texttt{0}\ \overset{\text{Ins}}{::}\ \mathbb{1}$	instantiate the `nat` index with value `0`
$\quad)$	

Figure 3.5. Diff for our running example (`nil` constructor only)

One might also find the modification of the parameter in Figure 3.6 hard to read. The diff $(\mathbb{K}(\,\texttt{vec}\,)\ \overset{\text{Mod}}{\$}\ \mathbb{1}_{\text{Term}})\ \overset{\text{Ins}}{\$}\ \texttt{n}$ might seem inverted, but it only appears so because function application is left-associative. Therefore, when comparing `list A` with `vec A n`, we are comparing `(app list A)` and `(app (app vec A) n)`. In this form, we can better see that the outermost application of `(app list A)` corresponds to the innermost application of `(app (app vec A) n)`, that is, `(app vec A)`. This should make it clear that the diff between these two terms needs to insert an application node first (the one with argument `n`), and then will modify the leftmost argument from `list` to `vec`. The reader might want to refer to rule Ins-App from Section 3.4.3 for the semantics of $\overset{\text{Ins}}{\$}$.

Again, the diff shown in Figure 3.4 would be the input to our repair algorithm,

```
…
| cons : A →
    list A → list A.
```

```
…
| vcons : A → ∀ (n : nat),
    vec A n → vec A (S n).
```

$$
\begin{array}{lr}
\delta\,\texttt{cons} := & \\
\quad \delta_{\text{Constructor}}(& \text{modify the } \texttt{cons} \text{ constructor}\\
\qquad \mathbb{K}(\texttt{vcons}), & \text{modify constructor name}\\
\qquad \mathbb{1} \overset{\text{Mod}}{::} & \text{keep the first parameter}\\
\qquad (\texttt{n} : \texttt{nat}) \overset{\text{Ins}}{::} & \text{insert a second parameter}\\
\qquad (\mathbb{1}_{\text{Binder}} : (\mathbb{K}(\texttt{vec}) \overset{\text{Mod}}{\$} \mathbb{1}_{\text{Term}}) \overset{\text{Ins}}{\$} \texttt{n}) \overset{\text{Mod}}{::} & \text{modify } \texttt{list A} \text{ into } \texttt{vec A n}\\
 & \text{while keeping binder anonymous}\\
\qquad \mathbb{1}_{\text{List}}, & \text{no other parameter}\\
\qquad (\texttt{S n}) \overset{\text{Ins}}{::} & \text{instantiate the } \texttt{nat} \text{ index}\\
\qquad \mathbb{1}_{\text{List}}, & \text{no other index}\\
\quad) &
\end{array}
$$

Figure 3.6. Diff for our running example (`cons` constructor only)

alongside the original program. The repair algorithm should propagate changes in such a way as to generate all the other fixes seen in teal in Figure 3.2.

3.4.6 Why compute diffs rather than repaired values?

We have now hinted multiple times to the importance of computing and storing diffs for our repair algorithm, rather than just computing and storing the repaired data. We now have the necessary syntax and semantics to explain how the two differ. Consider an original function `f`, with type `nat → bool → string`. Now consider the following two diffs that can change this type to `bool → nat → string`:

$$\delta_1 = \overset{[1,0]}{\overset{\rightleftarrows}{\Pi}}\ \mathbb{1}$$

$$\delta_2 = \overset{\text{Drop}}{\Pi}\ (\ \overset{\text{Mod}}{\Pi}\ (\mathbb{1} : \mathbb{1}) \rightarrow \overset{\text{Ins}}{\Pi}(\ \texttt{nat}\ :\ \_\ ) \rightarrow \mathbb{1})$$

Indeed if we run our patching function, we obtain:

$$\texttt{nat → bool → string} \overset{\delta_1}{\rightsquigarrow} \boxed{\texttt{bool → nat → string}}$$

$$\texttt{nat → bool → string} \overset{\delta_2}{\rightsquigarrow} \boxed{\texttt{bool → nat → string}}$$

Now, consider some client code that calls this function `f`. For instance, prior to repairing, let's say that we have a call:

```
let result := f (S n) b in …
```

If the only information we have available, about the type of `f`, is that its new type is `bool → nat → string`, all we can do is make a wild guess as to how we should modify this function call:

- should the `bool` argument be `b`?
- should the `nat` argument be `(S n)`?

Knowing which diff turned the old type of `f` into the new type gives us much more structural information, allowing us to make an educated guess. If the diff was δ_1, that is, $\overset{[1,0]}{\overset{\rightleftarrows}{\Pi}}\ \mathbb{1}$, then it appears the arguments were swapped, and so we should patch the client code as follows:

```
let result := f b (S n) in …
```

On the other hand, if the diff was δ_2, that is, $\overset{\text{Drop}}{\Pi}\ (\overset{\text{Mod}}{\Pi}\ (\mathbb{1} : \mathbb{1}) \to \overset{\text{Ins}}{\Pi}(\,\texttt{nat} : \_\,) \to \mathbb{1})$, then it appears that the parameter of type `bool` has moved to first position, but the parameter of type `nat` is a brand new parameter, unrelated to the one that was present in the old type. Therefore, the patch to the client code should rather produce the following:

```
let result := f b (_ : nat) in …
```

where we inserted a typed hole of type `nat` rather than conserve the value `(S n)` from the old program, as our diff indicates it is not relevant.

This should illustrate how diffs contain more information than before-after pairs. As long as we keep propagating changes through the program as diffs, rather than patched constructs, we can retain the semantic intent of the changes throughout our repair algorithm. However, this relies on obtaining an original diff, based on the refactoring attempt from the user, that captures their intent. We will describe how we can guess satisfactory diffs in Section 3.8, though we will sometimes need the programmer to disambiguate their intent explicitly.

3.5 Lookup rules

When encountering a variable in a program under repair, we will often need to answer the question “What has happened to this variable?”. In order to do so, we generalize the usual notion of variable lookup. In presentations of lambda calculi where both a global environment and a local context carry typing information, the usual notion of variable lookup first looks in the local context (since it carries the “closest” enclosing scopes), then, if no binding is found, looks in the global environment.

We use the following judgments for looking up a variable in the local context Γ (on the left), and in the global environment E (on the right): [6]

$$\Gamma \vdash v : \boxed{\tau} \qquad E \vdash v : \boxed{\tau}$$

3.5.1 Lookup rules for types

The rules for looking up types are straightforward, as shown in Figures 3.7, 3.8, and 3.9. In the latter, we handle the complexity of inductive data type definitions. For those, a system like *Coq* will introduce not only a name for the inductive type being defined, but also one name per elimination principles it generates, and finally one name

[6] While the two judgments share the same syntax, it should always be clear whether the value to the left of the turnstile is a local context or a global environment.

$$\textsc{Lookup-Context}\ \dfrac{\Gamma \vdash v : \boxed{\tau}}{E, \Gamma \vdash v : \boxed{\tau}} \qquad \textsc{Lookup-Environment}\ \dfrac{\Gamma \nvdash v : \boxed{\tau} \qquad E \vdash v : \boxed{\tau}}{E, \Gamma \vdash v : \boxed{\tau}}$$

Figure 3.7. Lookup rules (local context and global environment)

$$\textsc{Lookup-Context-Here}\ \dfrac{}{(v : \tau) :: \Gamma \vdash v : \boxed{\tau}} \qquad \textsc{Lookup-Context-There}\ \dfrac{w \neq v \qquad \Gamma \vdash v : \boxed{\tau}}{(w : \tau) :: \Gamma \vdash v : \boxed{\tau}}$$

Figure 3.8. Lookup rules (local context)

per the constructor of the data type. This is captured in the lookup rules, where we check those in order before looking in the rest of the environment. Unfortunately, rule Lookup-Environment-There has to be quite bloated, but its premises only ensure that we look in the rest of the environment only when neither of the three previous rules hold.

3.5.2 Lookup rules for diffs

Our second notion of a lookup essentially follows the same strategy, but we are ***not*** interested in finding the type of the variable, but rather in knowing **how** *both* the binding *and* the binding type have been modified. To do so, we use the following judgment:

$$\left[\begin{array}{c} E \\ \delta_E \end{array}\right], \left[\begin{array}{c} \Gamma \\ \delta_\Gamma \end{array}\right] \vdash \left[\begin{array}{c} v \\ \boxed{\delta_v} \end{array}\right] : \left[\begin{array}{c} ? \\ \boxed{\delta_\tau} \end{array}\right]$$

$$\frac{\textsc{Lookup-Environment-Definition-Here}}{\text{Definition}(\{k, v, \tau, t\}) :: E \vdash v : \boxed{\tau}}$$

$$\textsc{Lookup-Environment-Definition-There}\quad \frac{w \neq v \qquad E \vdash v : \boxed{\tau}}{\text{Definition}(\{k, w, \tau, t\}) :: E \vdash v : \boxed{\tau}}$$

$$\textsc{Lookup-Environment-Inductive-Type-Here}$$

$$\frac{\begin{gathered} I = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u, \overline{c}\}) \\ \begin{cases} \quad v = n_{\text{ind}} & \text{and } \text{InductiveType}(n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}) = \boxed{\tau} \\ \text{or } \exists\, u_{\text{elim}} \in \texttt{Universe} \text{ such that} & \\ \qquad v = \text{EliminatorName}(I, u_{\text{elim}}) & \text{and } \text{EliminatorType}(I, u_{\text{elim}}) = \boxed{\tau} \\ \text{or } \exists\, C \in \overline{c} \text{ such that} & \\ \qquad C = \text{Constructor}(\{v, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) & \text{and } \text{ConstructorType}(I, C) = \boxed{\tau} \end{cases} \end{gathered}}{I :: E \vdash v : \boxed{\tau}}$$

$$\textsc{Lookup-Environment-There}$$

$$\frac{\begin{gathered} I = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\ \begin{cases} \quad v \neq n_{\text{ind}} \\ \text{and } \forall\, u_{\text{elim}} \in \texttt{Universe}\,, v \neq \text{EliminatorName}(I, u_{\text{elim}}) \\ \text{and } \forall\, \text{Constructor}(\{n_{\text{ctor}}, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \in \overline{c_{\text{ind}}}, v \neq n_{\text{ctor}} \end{cases} \\ E \vdash v : \boxed{\tau} \end{gathered}}{I :: E \vdash v : \boxed{\tau}}$$

Figure 3.9. Lookup rules (global environment)

$$\textsc{Diff-Lookup-Context}\quad \frac{\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix}v\\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix}?\\ \boxed{\delta_\tau}\end{bmatrix}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix}v\\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix}?\\ \boxed{\delta_\tau}\end{bmatrix}} \qquad \textsc{Diff-Lookup-Environment}\quad \frac{\begin{bmatrix}E\\ \delta_E\end{bmatrix} \vdash \begin{bmatrix}v\\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix}?\\ \boxed{\delta_\tau}\end{bmatrix}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix}v\\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix}?\\ \boxed{\delta_\tau}\end{bmatrix}}$$

Figure 3.10. Diff lookup rules (local context and global environment)

Notation

This judgment can be read as follows:

- **Top**: If you are interested in the variable v, as bound in the original global environment E and original local context Γ,
- **Bottom left**: and the global environment underwent modification δ_E,
- **Bottom left**: and the local context underwent modification δ_Γ,
- **Bottom right**: then, references to the variable must undergo modification $\boxed{\delta_v}$,
- **Bottom right**: and the variable's type has undergone modification $\boxed{\delta_\tau}$.

The rules in Figure 3.10 summarize the global strategy of first looking up in the local context, and then falling back to the global environment, analogously to the process in Figure 3.7.

Lookup rules for diffs in the local context

Figure 3.11 shows how we proceed to lookup for the diff of a variable from the original program in the local context. Note that, since we assume the original program

Diff-Lookup-Context-Same

$$\frac{\Gamma \vdash v : \boxed{\tau_v}}{\begin{bmatrix}\Gamma\\ \mathbb{1}\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\mathbb{1}}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\mathbb{1}}\end{bmatrix}}$$

Diff-Lookup-Context-Ins

$$\frac{\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_v}}\end{bmatrix}}{\begin{bmatrix}\Gamma\\ (\delta_w : \delta_{\tau_w}) \overset{\text{Ins}}{::} \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_v}}\end{bmatrix}}$$

Diff-Lookup-Context-Mod-Here

$$\frac{}{\begin{bmatrix}(v : \tau) :: \Gamma\\ (\delta_v : \delta_\tau) \overset{\text{Mod}}{::} \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}$$

Diff-Lookup-Context-Mod-There

$$\frac{w \neq v \qquad \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}{\begin{bmatrix}(w : \tau) :: \Gamma\\ (\delta_v : \delta_\tau) \overset{\text{Mod}}{::} \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}$$

Diff-Lookup-Context-Drop-Here

$$\frac{\text{Deprecate}(v) = \boxed{v'}}{\begin{bmatrix}(v : \tau) :: \Gamma\\ \overset{\text{Drop}}{::} \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\mathbb{K}(v')}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\mathbb{K}(\_)}\end{bmatrix}}$$

Diff-Lookup-Context-Drop-There

$$\frac{w \neq v \qquad \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}{\begin{bmatrix}(v : \tau) :: \Gamma\\ \overset{\text{Drop}}{::} \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}$$

Diff-Lookup-Context-Permute

$$\frac{\Gamma \overset{\overset{p}{\rightleftarrows}}{\underset{\rightsquigarrow}{::\mathbb{1}}} \boxed{\Gamma'} \qquad \begin{bmatrix}\Gamma'\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}{\begin{bmatrix}\Gamma\\ \overset{\overset{p}{\rightleftarrows}}{::} \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau}\end{bmatrix}}$$

Figure 3.11. Diff lookup rules (local context)

type-checked, when we look up variables, we can assume that they are properly bound: were it not the case, we would need additional rules to account for the absence of a variable, and to change the output types to allow for failure.

When the context hasn't changed, we can obtain its type from the old environment (Rule Diff-Lookup-Context-Same). When some variable has been inserted, we can ignore it with no effort, since we only care about shadowing in the original program (Rule Diff-Lookup-Context-Ins).

When we find the target variable in the local context, there are only two remaining possibilities for the diff: either it is a $\overset{\text{Mod}}{\Pi}$, or a $\overset{\text{Drop}}{\Pi}$. When it has been modified, we have found exactly the information we were looking for, so we can simply output it (Rule Diff-Lookup-Context-Mod-Here). If it has been dropped, then there is no satisfactory result, the variable is no longer in the program. In order to make sure the user notices, we rename the variable using the helper Deprecate, and make its diff as uninformative as possible using a hole (Rule Diff-Lookup-Context-Drop-Here).

In the cases were a variable other than our target is being modified (covered in Rule Diff-Lookup-Context-Mod-There), or in the cases were a variable other than our target is being dropped (covered in Rule Diff-Lookup-Context-Drop-There), we simply keep looking recursively into the remaining context. Finally, if the context has undergone a permutation (Rule Diff-Lookup-Context-Permute), we can permute the context appropriately and keep looking recursively.

Lookup rules for diffs in the global environment

Unfortunately, the number of combinations we need to consider for diffs grow alongside the number of constructs that introduce variables, as well as along the num-

Diff-Lookup-Env-Same

$$\frac{E \vdash v : \boxed{\tau_v}}{\begin{bmatrix} E \\ \mathbb{1} \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\mathbb{1}} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\mathbb{1}} \end{bmatrix}}$$

Diff-Lookup-Env-Ins

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_v}} \end{bmatrix}}{\begin{bmatrix} E \\ e \overset{\text{Ins}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_v}} \end{bmatrix}}$$

Diff-Lookup-Env-Permute

$$\frac{E \overset{\overset{p}{\rightleftarrows}}{::} \mathbb{1} \rightsquigarrow \boxed{E'} \qquad \begin{bmatrix} E' \\ \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}{\begin{bmatrix} E \\ \overset{\overset{p}{\rightleftarrows}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}$$

Figure 3.12. Lookup in the global environment (identity, insertion, permutation)

ber of modifications that happen to those constructs. Since inductive data type definitions introduce three classes of variables (namely, the type, the eliminators, and the constructors), and we have four classes of changes for lists ($\mathbb{1}$, $\overset{\text{Mod}}{::}$, $\overset{\text{Drop}}{::}$, $\rightleftarrows$), this raises the number of cases in a quadratic fashion. We do our best to summarize those cases in few rules, by using disjunctive premises to account for many cases in a single rule.

Figure 3.12 covers the simplest cases. When the environment has not changed, as in Rule Diff-Lookup-Env-Same, the variable must not have changed either, as long as it actually existed. When any Definition or Inductive has been inserted, we can skip it and keep looking recursively (Rule Diff-Lookup-Env-Ins). Finally, when the environment has undergone a permutation, we can simply proceed recursively on the appropriately permuted environment (Rule Diff-Lookup-Env-Permute).

When an element has been removed from the environment (Figure 3.13), we need to check whether it was our target variable that has been removed. For the case of a `Definition`, it suffices to check whether the name matches, but for an `Inductive`,

Diff-Lookup-Env-Drop-Here

$$\frac{\begin{array}{c}\left\{\begin{array}{ll} & e = \text{Definition}(\{k, v, \tau, t\}) \\ \text{or} & \left\{\begin{array}{ll} & e = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\ \text{and} & \left\{\begin{array}{l} \quad v = n_{\text{ind}} \\ \text{or } \exists\, u_{\text{elim}} \in \texttt{Universe} \text{ such that } v = \text{EliminatorName}(e, u_{\text{elim}}) \\ \text{or } \exists\, C \in \overline{c} \text{ such that } C = \text{Constructor}(\{v, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \end{array}\right. \end{array}\right. \end{array}\right. \\ \text{Deprecate}(v) = \boxed{v'} \end{array}}{\begin{bmatrix} e :: E \\ \overset{\text{Drop}}{::}\ \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\mathbb{K}(v')} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\mathbb{K}(\_)} \end{bmatrix}}$$

Diff-Lookup-Env-Drop-There

$$\frac{\begin{array}{c}\left\{\begin{array}{ll} & e = \text{Definition}(\{k, w, \tau, t\}) \text{ and } v \neq w \\ \text{or} & \left\{\begin{array}{ll} & e = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\ \text{and} & \left\{\begin{array}{l} \quad v \neq n_{\text{ind}} \\ \text{and } \forall\, u_{\text{elim}} \in \texttt{Universe}\,, v \neq \text{EliminatorName}(e, u_{\text{elim}}) \\ \text{and } \forall\, \text{Constructor}(\{n_{\text{ctor}}, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \in \overline{c_{\text{ind}}}, v \neq n_{\text{ctor}} \end{array}\right. \end{array}\right. \end{array}\right. \\ \begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix} \end{array}}{\begin{bmatrix} e :: E \\ \overset{\text{Drop}}{::}\ \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}$$

Figure 3.13. Lookup in the global environment (drop)

we need to check whether the name matches the inductive type, any of the eliminators, or any of the constructors. If it does, then we indicate that the variable has been removed by using Deprecate again (Rule Diff-Lookup-Env-Drop-Here). Otherwise, we keep looking recursively (Rule Diff-Lookup-Env-Drop-There).

When an element has been modified in the environment (Figure 3.14), we again need to check whether our target variable is the one that has been removed. For the same reasons as in the drop case, we need to check all possibilities for inductive data type definitions, yielding Rule Diff-Lookup-Env-Mod-Here for cases where the target variable is being modified, and Rule Diff-Lookup-Env-Mod-There for cases where other variables than our target are being modified.

Finally, the rules in Figures 3.15 and 3.16 describe diffs lookups in cases where the variable we are looking up could be one of the constructors of a modified inductive data type declaration. The rules search through the list of constructors, ignoring inserted constructors, until they find the target or the lookup or exhaust the list of constructors.

3.6 Repairing programs by propagating changes

In this section, we assume that we are given an original program p, that is, a sequence of vernacular commands as described in Section 3.3.2, and a diff of that program δ_p as described in Section 3.4, capturing a partial refactoring made by the user. We assume that the original proof script type-checked, and attempt to build a repaired diff δ'_p such that:

Diff-Lookup-Env-Mod-Here

$$\dfrac{\left\{\begin{array}{l} e = \text{Definition}(\{k, v, \tau, t\}) \text{ and } \delta_e = \delta_{\text{Definition}}(\{\delta_k, \boxed{\delta_v}, \boxed{\delta_\tau}, \delta_t\}) \\ \text{or} \\ \left\{\begin{array}{l} \quad e = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\ \text{and } \delta_e = \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta\overline{c_{\text{ind}}}\}) \\ \text{and} \left\{\begin{array}{l} \left\{\begin{array}{l} \qquad v = n_{\text{ind}} \\ \text{and } \delta_n = \boxed{\delta_v} \\ \text{and } \delta_{\text{InductiveType}}\left(\begin{bmatrix} n_{\text{ind}} \\ \delta n_{\text{ind}} \end{bmatrix}, \begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} \overline{i_{\text{ind}}} \\ \delta\overline{i_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} u_{\text{ind}} \\ \delta u_{\text{ind}} \end{bmatrix}\right) = \boxed{\delta_\tau} \end{array}\right. \\ \text{or} \left\{\begin{array}{l} \qquad \exists\, u_{\text{elim}} \in \texttt{Universe} \\ \qquad \text{such that } v = \text{EliminatorName}(e, u_{\text{elim}}) \\ \text{and } \delta_{\text{EliminatorName}}\left(\begin{bmatrix} e \\ \delta_e \end{bmatrix}, u_{\text{elim}}\right) = \boxed{\delta_v} \\ \text{and } \delta_{\text{EliminatorType}}\left(\begin{bmatrix} e \\ \delta_e \end{bmatrix}, u_{\text{elim}}\right) = \boxed{\delta_\tau} \end{array}\right. \end{array}\right. \end{array}\right. \end{array}\right.}{\begin{bmatrix} e :: E \\ \delta_e \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}$$

Diff-Lookup-Env-Mod-There

$$\dfrac{\left\{\begin{array}{l} \qquad e = \text{Definition}(\{k, w, \tau, t\}) \text{ and } v \neq w \\ \text{or} \left\{\begin{array}{l} e = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\ \text{and} \left\{\begin{array}{l} \qquad v \neq n_{\text{ind}} \\ \text{and } \forall\, u_{\text{elim}} \in \texttt{Universe}, v \neq \text{EliminatorName}(e, u_{\text{elim}}) \\ \text{and } \forall\, \text{Constructor}(\{n_{\text{ctor}}, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \in \overline{c_{\text{ind}}}, v \neq n_{\text{ctor}} \end{array}\right. \end{array}\right. \end{array}\right. \\[1em] \begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}{\begin{bmatrix} e :: E \\ \delta_e \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}$$

Figure 3.14. Lookup in the global environment (modification)

Diff-Lookup-Env-Mod-Constructor-Same-Here

$$\frac{C = \text{Constructor}(\{v, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\})}{\left[\begin{array}{c} \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, C :: \overline{c_{\text{ind}}}\}) :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \mathbb{1}\}) \end{array}\right] \vdash \left[\begin{array}{c} v \\ \boxed{\mathbb{1}} \end{array}\right] : \left[\begin{array}{c} ? \\ \boxed{\mathbb{1}} \end{array}\right]}$$

Diff-Lookup-Env-Mod-Constructor-Mod-Here

$$\frac{\begin{array}{c} I = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, C :: \overline{c_{\text{ind}}}\}) \\[1em] \delta_I = \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta_C \overset{\text{Mod}}{::} \delta\overline{c_{\text{ind}}}\}) \\[1em] C = \text{Constructor}(\{v, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \qquad \delta_C = \delta_{\text{Constructor}}(\{\delta_v, \delta_{\overline{p_{\text{ctor}}}}, \delta_{\overline{i_{\text{ctor}}}}\}) \\[1em] \delta_{\text{ConstructorType}}\left(\left[\begin{array}{c} I \\ \delta_I \end{array}\right], \left[\begin{array}{c} C \\ \delta_C \end{array}\right]\right) = \boxed{\delta_\tau} \end{array}}{\left[\begin{array}{c} I :: E \\ \delta_I \overset{\text{Mod}}{::} \delta_E \end{array}\right] \vdash \left[\begin{array}{c} v \\ \boxed{\delta_v} \end{array}\right] : \left[\begin{array}{c} ? \\ \boxed{\delta_\tau} \end{array}\right]}$$

Diff-Lookup-Env-Mod-Constructor-Drop-Here

$$\frac{C = \text{Constructor}(\{v, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \qquad \text{Deprecate}(v) = v'}{\left[\begin{array}{c} \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, C :: \overline{c_{\text{ind}}}\}) :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \overset{\text{Drop}}{::} \delta\overline{c_{\text{ind}}}\}) \overset{\text{Mod}}{::} \delta_E \end{array}\right] \vdash \left[\begin{array}{c} v \\ \boxed{v'} \end{array}\right] : \left[\begin{array}{c} ? \\ \boxed{\mathbb{K}(\_)} \end{array}\right]}$$

Diff-Lookup-Env-Mod-Constructor-There

$$\frac{\begin{array}{c} C = \text{Constructor}(\{w, \overline{p_{\text{ctor}}}, \overline{i_{\text{ctor}}}\}) \qquad v \neq w \\[1em] \begin{cases} \quad \delta = \mathbb{1} & \delta' = \mathbb{1} \\ \text{or } \delta = \delta_C \overset{\text{Mod}}{::} \delta\overline{c_{\text{ind}}} & \delta' = \delta\overline{c_{\text{ind}}} \\ \text{or } \delta = \overset{\text{Drop}}{::} \delta\overline{c_{\text{ind}}} & \delta' = \delta\overline{c_{\text{ind}}} \end{cases} \\[1em] \left[\begin{array}{c} \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta'\}) \overset{\text{Mod}}{::} \delta_E \end{array}\right] \vdash \left[\begin{array}{c} v \\ \boxed{\delta_v} \end{array}\right] : \left[\begin{array}{c} ? \\ \boxed{\delta_\tau} \end{array}\right] \end{array}}{\left[\begin{array}{c} \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, C :: \overline{c_{\text{ind}}}\}) :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta\}) \overset{\text{Mod}}{::} \delta_E \end{array}\right] \vdash \left[\begin{array}{c} v \\ \boxed{\delta_v} \end{array}\right] : \left[\begin{array}{c} ? \\ \boxed{\delta_\tau} \end{array}\right]}$$

Figure 3.15. Lookup in the global environment (constructor rules, part 1/2)

Diff-Lookup-Env-Mod-Constructor-Ins

$$\frac{\begin{array}{c} I = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\[1em] \delta_I = \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, C \overset{\text{Ins}}{::} \delta\overline{c_{\text{ind}}}\}) \\[1em] \begin{bmatrix} I :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta\overline{c_{\text{ind}}}\}) \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix} \end{array}}{\begin{bmatrix} I :: E \\ \delta_I \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}$$

Diff-Lookup-Env-Mod-Constructor-Permute

$$\frac{\begin{array}{c} \overline{c_{\text{ind}}} \underset{\rightsquigarrow}{\overset{\overset{p}{\rightleftarrows}}{::\mathbb{1}}} \boxed{\overline{c_{\text{ind}}}'} \\[1em] \begin{bmatrix} \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}'\}) :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta\overline{c_{\text{ind}}}\}) \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix} \end{array}}{\begin{bmatrix} \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) :: E \\ \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \overset{\overset{p}{\rightleftarrows}}{::} \delta\overline{c_{\text{ind}}}\}) \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_\tau} \end{bmatrix}}$$

Figure 3.16. Lookup in the global environment (constructor rules, part 2/2)

- δ'_p contains the changes from δ_p,

- δ'_p completes the refactoring started by δ_p, by propagating forward the changes from δ_p to use-sites that must be repaired to account for those changes,

where propagating changes forward means:

- propagating renaming of constants and variables,

- propagating changes in the number, order, and type of arguments to functions, obtained from their definition, to their use-site,

- propagating changes in the definition of inductive data types to use-sites of the type, its constructors, and its eliminators (those will be explained in more details in Section 3.7).

We give a top-down description of the repair algorithm, starting at the level of whole programs, descending all the way to terms and data type definitions.

3.6.1 Repairing programs

The repair algorithm for programs R_{Program} is described formally in Figure 3.17 using the following judgment:

$$\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}(\begin{bmatrix} p \\ \delta_p \end{bmatrix}) = \boxed{\delta'_p}$$

Notation

This judgment can be read as follows:

- **Top**: If the original program p type-checked in the original environment E,
- **Bottom left**: and the environment underwent modification δ_E,
- **Bottom center**: and the program underwent modification δ_p,
- **Right**: then R_{Program} proposes to repair the program with modification $\boxed{\delta'_p}$.

To be more precise, E is the environment in which the original program p was defined: a list of term definitions, with type Definition, and of inductive data type definitions, with type Inductive (as presented in Section 3.3.2). Each vernacular command is type-checked in such a global typing environment, and upon being executed, populates it with some additional definitions for the subsequent commands.

δ_E is a diff indicating how the global environment has been changed by the time p is reached by the repair algorithm: it accounts for whether some definitions have been added, removed, or modified, in the prefix of the already repaired program preceding p. For instance, in our motivating example, by the time we reach the inductive definition being modified (`list` , becoming `vec`), E would contain the inductive data type definition for the type `nat` , and δ_E would keep it intact. After this definition has been processed, E would contain, in order, `nat` , and `list` , and δ_E would still keep the former intact, but would register the diff turning `list` into `vec` .

The repair algorithm for programs takes as input the original program p, and the user-provided modification δ_p. From those, it outputs a repaired modification $\boxed{\delta'_p}$, that propagates the changes introduced by δ_E and δ_p to the rest of the program p. The algorithm essentially folds over the sequence of vernacular commands that make up the

$$\textsc{Repair-Program-Same-Nil}\quad \frac{}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} [\,] \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\mathbb{1}}}$$

$$\textsc{Repair-Program-Same-Cons}\quad \frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} v :: p \\ \mathbb{1}_{\text{Vernacular}} \overset{\text{Mod}}{::} \mathbb{1}_{\text{Program}} \end{bmatrix}\right) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} v :: p \\ \mathbb{1}_{\text{Program}} \end{bmatrix}\right) = \boxed{\delta}}$$

$$\textsc{Repair-Program-Replace}\quad \frac{}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} p \\ \mathbb{K}(q) \end{bmatrix}\right) = \boxed{\mathbb{K}(q)}}$$

$$\textsc{Repair-Program-Insert}\quad \frac{\begin{bmatrix} E \\ v \overset{\text{Ins}}{::} \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} p \\ \delta_p \end{bmatrix}\right) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} p \\ v \overset{\text{Ins}}{::} \delta_p \end{bmatrix}\right) = \boxed{\delta}}$$

$$\textsc{Repair-Program-Modify}\quad \frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Vernacular}}\left(\begin{bmatrix} v \\ \delta_v \end{bmatrix}\right) = \boxed{\delta_v'} \qquad \begin{bmatrix} v :: E \\ \delta_v' \overset{\text{Mod}}{::} \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} p \\ \delta_p \end{bmatrix}\right) = \boxed{\delta_p'}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} v :: p \\ \delta_v \overset{\text{Mod}}{::} \delta_p \end{bmatrix}\right) = \boxed{\delta_v' \overset{\text{Mod}}{::} \delta_p'}}$$

$$\textsc{Repair-Program-Drop}\quad \frac{\begin{bmatrix} v :: E \\ \overset{\text{Drop}}{::} \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} p \\ \delta_p \end{bmatrix}\right) = \boxed{\delta_p'}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} v :: p \\ \overset{\text{Drop}}{::} \delta_p \end{bmatrix}\right) = \boxed{\overset{\text{Drop}}{::} \delta_p'}}$$

$$\textsc{Repair-Program-Permute}\quad \frac{q \overset{\overset{\overset{p}{\rightleftarrows}}{::}\mathbb{1}}{\rightsquigarrow} \boxed{q'} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} q' \\ \delta \end{bmatrix}\right) = \boxed{\delta'}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Program}}\left(\begin{bmatrix} q \\ \overset{\overset{p}{\rightleftarrows}}{::} \delta \end{bmatrix}\right) = \boxed{\overset{\overset{p}{\rightleftarrows}}{::} \delta'}}$$

Figure 3.17. Rules for repairing programs (R_{Program})

program, propagating changes from previous commands to subsequent ones.

Rule Repair-Program-Modify does the bulk of the work, dispatching the repair of each vernacular command to the repair algorithm for vernacular commands $R_{\text{Vernacular}}$ (described in Section 3.6.2). First, the head vernacular command v is repaired (using $R_{\text{Vernacular}}$), returning a repaired diff $\boxed{\delta'_v}$. Then, we want to repair the rest of the program, but the changes made to v may affect the global environment for the subsequent commands. For instance, in our motivating example, the vernacular command defining the inductive type `list` gets modified to define the inductive type `vec` instead: when we repair the rest of the program, we must remember that the original program was defined in a global environment where `list` was defined, and that the updated program must replace it with `vec` . We therefore repair the rest of the program p in the appropriate updated environment and its diff.

Rule Repair-Program-Same-Cons applies when the list of vernacular commands has not changed. It defers to Rule Repair-Program-Modify: even though it the commands have not changed, they might need repairs to account for changes in their dependencies, as accounted for in the global environment diff. Repair-Program-Insert and Repair-Program-Drop update their environment as necessary. Finally, Rule RProgPermute recursively repairs the permuted sequence of commands.

Note

While our presentation makes it look like vernacular commands each add exactly one element to the global environment, it is not always as simple: inductive data type definitions add 1. the inductive type (e.g. `list`), 2. all its constructors (e.g. `nil` , `cons`), and 3. all its eliminators (e.g. `list_rec` , `list_ind` , as described later in Section 3.7). For simplifying diff operations, the whole Inductive effectively acts as a placeholder, in our formalism, for all of those at once. The lookup rules, given in Appendix 3.5, encapsulate the complexity of accounting for all the definitions arising from one inductive data type definition.

3.6.2 Repairing vernacular commands

We formally describe the repair algorithm for vernacular commands $R_{\text{Vernacular}}$ in Figure 3.18 using the following judgment:

$$\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Vernacular}}(\begin{bmatrix} v \\ \delta_v \end{bmatrix}) = \boxed{\delta'_v}$$

which can be read analogously to R_{Program} as described in Section 3.6.1.

Rule Repair-Vernacular-Def shows how we attempt to repair a vernacular term definition. It is quite involved because we account for the possibility of a non-recursive definition becoming recursive and vice-versa. We want to account for the user-provided modifications δ_k, δ_n, δ_τ, and δ_t, but we might need to repair some of them. We first repair the type τ, and obtain its repaired diff δ'_τ. Now, the user has modified the name of the definition according to δ_n, but there are two situations where we must intervene:

1. If δ_n is $\mathbb{1}$, the user intends to keep the name n for this definition. However, they could have introduced another name n in the global environment, which would be accounted for in δ_E. In this case, we need to come up with a fresh name that is free in the new environment.

2. If δ_n is $\mathbb{K}(m)$, the user intends to rename this definition from n into m. Again, even though this is unlikely in practice, it could be a problem if the new global

$$\textsc{Repair-Vernacular-Definition}\quad \dfrac{\begin{array}{c}
\tau \overset{\delta_\tau}{\rightsquigarrow} \boxed{\tau'} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} [] \\ \mathbb{1} \end{bmatrix} \vdash R_{\text{Term}_1}\left(\tau' : \begin{bmatrix} \text{Type} \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta'_\tau} \qquad \tau' \overset{\delta'_\tau}{\rightsquigarrow} \boxed{\tau''} \\[2ex]
\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash \text{Fresh}_1\left(\begin{bmatrix} n \\ \delta_n \end{bmatrix}\right) = \boxed{\delta'_n} \qquad n \overset{\delta'_n}{\rightsquigarrow} \boxed{n'} \qquad \Gamma = \begin{cases} [] & \text{if } k = \text{Definition} \\ (n : \tau) :: [] & \text{if } k = \text{Fixpoint} \end{cases} \\[2ex]
\delta_\Gamma = \begin{cases} (n' : \tau'') \overset{\text{Ins}}{::} \mathbb{1} & \text{when } k = \text{Definition} \\ & \qquad \delta_k = \mathbb{K}(\text{Fixpoint}) \\ \overset{\text{Drop}}{::} \mathbb{1} & \text{when } k = \text{Fixpoint} \\ & \qquad \delta_k = \mathbb{K}(\text{Definition}) \\ \mathbb{1} & \text{otherwise} \end{cases} \\[2ex]
t \overset{\delta_t[\delta_n \leftarrow \delta'_n]}{\rightsquigarrow} \boxed{t'} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t' : \begin{bmatrix} \tau \\ \delta'_\tau \end{bmatrix}\right) = \boxed{\delta'_t}
\end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Vernacular}}\left(\begin{bmatrix} \text{Definition}(\{k, n, \tau, t\}) \\ \delta_{\text{Definition}}(\{\delta_k, \delta_n, \delta_\tau, \delta_t\}) \end{bmatrix}\right) = \boxed{\delta_{\text{Definition}}(\{\delta_k, \delta'_n, \delta'_\tau, \delta'_t\})}}$$

$$\textsc{Repair-Vernacular-Inductive}\quad \dfrac{\begin{array}{c}
I = \text{Inductive}(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\}) \\[2ex]
\delta_I = \delta_{\text{Inductive}}(\{\delta n_{\text{ind}}, \delta\overline{p_{\text{ind}}}, \delta\overline{i_{\text{ind}}}, \delta u_{\text{ind}}, \delta\overline{c_{\text{ind}}}\}) \\[2ex]
\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash \text{Fresh}_1\left(\begin{bmatrix} n_{\text{ind}} \\ \delta n_{\text{ind}} \end{bmatrix}\right) = \boxed{\delta n_{\text{ind}}'} \\[2ex]
\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Inductive}}\left(\begin{bmatrix} n_{\text{ind}} \\ \delta n_{\text{ind}} \end{bmatrix}, \begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} \overline{i_{\text{ind}}} \\ \delta\overline{i_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} u_{\text{ind}} \\ \delta u_{\text{ind}} \end{bmatrix}, \begin{bmatrix} \overline{c_{\text{ind}}} \\ \delta\overline{c_{\text{ind}}} \end{bmatrix}\right) = \boxed{(\delta\overline{p_{\text{ind}}}', \delta\overline{i_{\text{ind}}}', \delta\overline{c_{\text{ind}}}')}
\end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Vernacular}}\left(\begin{bmatrix} I \\ \delta_I \end{bmatrix}\right) = \boxed{\delta_{\text{Inductive}}(\{\delta n_{\text{ind}}', \delta\overline{p_{\text{ind}}}', \delta\overline{i_{\text{ind}}}', \delta u_{\text{ind}}, \delta\overline{c_{\text{ind}}}'\})}}$$

Figure 3.18. Rules for repairing vernacular commands ($R_{\text{Vernacular}}$)

environment contains an m already. In this case, we also need to come up with a fresh name that is free in the new environment.

The helper function Fresh_1 takes care of these two situations: it takes as input a name n, and a desired modification for it δ_n, and returns a suitable modification δ'_n, that first tries to preserve δ_n when possible, otherwise tries to preserve n when possible, or finally, comes up with a fresh name when neither of these conditions can be met.

Finally, in order to repair the body of the definition, we need to build an initial local context (as described in Section 3.5). This context is empty for non-recursive definitions, but should contain a self-reference for recursive definitions. The side conditions defining Γ and δ_Γ make sure that the local context is appropriately set in all possible combinations of the recursive flag before and after the user-modification.

3.6.3 Repairing inductive data type definitions

Figure 3.19 gives the high-level overview of how inductive data type definitions are repaired. The repair operates in three sequential steps:

1. First, the inductive parameters are repaired using $R_{\text{Parameters}}$. Starting with an empty local context, the first parameter is repaired, and added to the local context, with its repaired diff, before repairing the next parameter. Thanks to the local context, further parameters may be repaired appropriately if they depend on previous parameters that have undergone repair already. We return the final local context, alongside the repaired parameters, because we need to update the indices in the same context.

2. Second, the inductive indices are repaired using R_{Indices}, starting in the local context obtained at the end of the previous step. Indices are repaired in a similar

Repair-Vernacular-Inductive

$$\frac{\begin{array}{c}
\begin{bmatrix} [] \\ \mathbb{1} \end{bmatrix}, \begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix}, \delta\overline{p_{\text{ind}}}'\right)} \\[2ex]
\begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix}, \begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Indices}}\left(\begin{bmatrix} \overline{i_{\text{ind}}} \\ \delta\overline{i_{\text{ind}}} \end{bmatrix}\right) = \boxed{\delta\overline{i_{\text{ind}}}'} \\[2ex]
\text{InductiveType}(n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}) = \boxed{\tau} \\[2ex]
\delta_{\text{InductiveType}}\left(\begin{bmatrix} n_{\text{ind}} \\ \delta n_{\text{ind}} \end{bmatrix}, \begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} \overline{i_{\text{ind}}} \\ \delta\overline{i_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} u_{\text{ind}} \\ \delta u_{\text{ind}} \end{bmatrix}\right) = \boxed{\delta_\tau} \\[2ex]
\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (n_{\text{ind}} : \tau) :: [] \\ (\delta n_{\text{ind}} : \delta_\tau) \overset{\text{Mod}}{::} \mathbb{1} \end{bmatrix} \vdash R_{\text{Constructors}}\left(\begin{bmatrix} \overline{c_{\text{ind}}} \\ \delta\overline{c_{\text{ind}}} \end{bmatrix}\right) = \boxed{\delta\overline{c_{\text{ind}}}'}
\end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix} \vdash R_{\text{Inductive}}\left(\begin{bmatrix} n_{\text{ind}} \\ \delta n_{\text{ind}} \end{bmatrix}, \begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} \overline{i_{\text{ind}}} \\ \delta\overline{i_{\text{ind}}} \end{bmatrix}, \begin{bmatrix} u_{\text{ind}} \\ \delta u_{\text{ind}} \end{bmatrix}, \begin{bmatrix} \overline{c_{\text{ind}}} \\ \delta\overline{c_{\text{ind}}} \end{bmatrix}\right) = \boxed{(\delta\overline{p_{\text{ind}}}', \delta\overline{i_{\text{ind}}}', \delta\overline{c_{\text{ind}}}')}}$$

Figure 3.19. Rules for repairing inductive data type definitions ($R_{\text{Inductive}}$)

fashion to parameters, populating the context with each repaired index before repairing the next one. We don't need the final context further, so it is not returned.

3. Once we have repaired parameters and indices, we are ready to compute how the type of the inductive data type is changing, using $\delta_{\text{InductiveType}}$. We need this information in order to repair the constructors, because constructors can contain recursive references to the inductive data type being defined in their parameters.

We omit most of the details of repairing an inductive data type definition behind the symbol $R_{\text{Inductive}}$. It essentially needs to go recursively in all parameters and indices of both the inductive type itself and all of its constructors and repair them. Each subsequent parameter must be repaired in a local context where its predecessor parameter has been accounted, so as to react to possible modifications: for instance, if the first parameter must be renamed, and the second parameter's type refers to the first param-

Repair-Parameters-Same

$$\frac{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Parameters}}(\begin{bmatrix}(v:\tau) :: \overline{p_{\text{ind}}}\\ (\mathbb{1}_{\text{Variable}} : \mathbb{1}_{\text{Term}}) \overset{\text{Mod}}{::} \mathbb{1}_{\text{List}}\end{bmatrix}) = \boxed{(\begin{bmatrix}\Gamma'\\ \delta'_\Gamma\end{bmatrix}, \delta)}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Parameters}}(\begin{bmatrix}(v:\tau) :: \overline{p_{\text{ind}}}\\ \mathbb{1}_{\text{List}}\end{bmatrix}) = \boxed{(\begin{bmatrix}\Gamma'\\ \delta'_\Gamma\end{bmatrix}, \delta)}}$$

Repair-Parameters-Replace

$$\frac{\delta'_\Gamma = p'_{\text{ind}_{|\overline{p'_{\text{ind}}}|}} \overset{\text{Ins}}{::} (\ldots \overset{\text{Ins}}{::} (p'_{\text{ind}_1} \overset{\text{Ins}}{::} \delta_\Gamma))}{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Parameters}}(\begin{bmatrix}\overline{p_{\text{ind}}}\\ \mathbb{K}(\overline{p'_{\text{ind}}})\end{bmatrix}) = \boxed{(\begin{bmatrix}\Gamma\\ \delta'_\Gamma\end{bmatrix}, \mathbb{K}(\overline{p'_{\text{ind}}}))}}$$

Repair-Parameters-Permute

$$\frac{\overline{p_{\text{ind}}} \overset{\overset{p}{\rightleftarrows} :: \mathbb{1}}{\rightsquigarrow} \boxed{\overline{p'_{\text{ind}}}} \qquad \begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Parameters}}(\begin{bmatrix}\overline{p'_{\text{ind}}}\\ \delta\end{bmatrix}) = \boxed{(\begin{bmatrix}\Gamma'\\ \delta'_\Gamma\end{bmatrix}, \delta')}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Parameters}}(\begin{bmatrix}\overline{p_{\text{ind}}}\\ \overset{p}{\rightleftarrows}\\ ::\ \delta\end{bmatrix}) = \boxed{(\begin{bmatrix}\Gamma'\\ \delta'_\Gamma\end{bmatrix}, \delta')}}$$

Figure 3.20. Rules for repairing inductive parameters ($R_{\text{Parameters}}$, part 1/2)

eter's name, the type will need to be repaired. Indices must also be repaired in the context of the repaired parameters. Constructors are repaired one by one, in isolation, but the repairing of their parameters. Constructor indices are simply instances of the inductive indices with terms: they are simply repaired in isolation.

Repairing inductive parameters

Figures 3.20 and 3.21 show the rules for $R_{\text{Parameters}}$, the algorithm for repairing inductive parameters. As previously mentioned, the algorithm must return its final local context (and diff), so that the inductive indices may be repaired in the proper context. While our formalization in this dissertation returns the contexts explicitly, our imple-

$$\textsc{Repair-Parameters-Ins}\quad \frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ (v:\tau) \overset{\text{Ins}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma' \\ \delta'_\Gamma \end{bmatrix}, \delta\overline{p'_{\text{ind}}}\right)}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} \overline{p_{\text{ind}}} \\ (v:\tau) \overset{\text{Ins}}{::} \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma' \\ \delta'_\Gamma \end{bmatrix}, \delta\overline{p'_{\text{ind}}}\right)}}$$

$$\textsc{Repair-Parameters-Mod}\quad \frac{\begin{gathered} \text{Fresh}_1\left(\begin{bmatrix} v \\ \delta_v \end{bmatrix}\right) = \boxed{\delta'_v} \qquad \tau \overset{\delta_\tau}{\rightsquigarrow} \boxed{\tau'} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(\tau : \begin{bmatrix} \mathtt{Type} \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta'_\tau} \\ \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (v:\tau) :: \Gamma \\ (\delta'_v : \delta'_\tau) \overset{\text{Mod}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma' \\ \delta'_\Gamma \end{bmatrix}, (\delta'_v : \delta'_\tau) \overset{\text{Mod}}{::} \delta\overline{p'_{\text{ind}}}\right)} \end{gathered}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} (v:\tau) :: \overline{p_{\text{ind}}} \\ (\delta_v : \delta_\tau) \overset{\text{Mod}}{::} \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma' \\ \delta'_\Gamma \end{bmatrix}, \delta\overline{p'_{\text{ind}}}\right)}}$$

$$\textsc{Repair-Parameters-Drop}\quad \frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (v:\tau) :: \Gamma \\ \overset{\text{Drop}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} \overline{p_{\text{ind}}} \\ \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma' \\ \delta'_\Gamma \end{bmatrix}, \delta\overline{p'_{\text{ind}}}\right)}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Parameters}}\left(\begin{bmatrix} (v:\tau) :: \overline{p_{\text{ind}}} \\ \overset{\text{Drop}}{::} \delta\overline{p_{\text{ind}}} \end{bmatrix}\right) = \boxed{\left(\begin{bmatrix} \Gamma' \\ \delta'_\Gamma \end{bmatrix}, \delta\overline{p'_{\text{ind}}}\right)}}$$

Figure 3.21. Rules for repairing inductive parameters ($R_{\text{Parameters}}$, part 2/2)

mentation uses a *state monad* to pass around local context and global environment.

The rules are fairly straightforward, recursively going over every parameter in the list, and accounting for their changes in the local context.

Repairing inductive indices

Inductive indices are also given as a telescope, like the inductive parameters, with the only difference being that their binder may be omitted, where inductive parameters must be named. They can be repaired in a similar fashion to inductive parameters, so we omit the rules for indices.

3.6.4 Repairing terms

The repair algorithm for terms is described formally in Figures 3.22, 3.23, 3.24, and 3.25 using the following judgments:

$$\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\mathrm{Term}_1}(t : \begin{bmatrix} \tau \\ \delta_\tau \end{bmatrix}) = \boxed{\delta_t}$$

$$\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\mathrm{Term}_2}(t) = \boxed{\delta_t}$$

The first judgment (described in Figures 3.22 and 3.23) describes how a term is repaired when we know both its old type τ, and how that type got modified δ_τ. It is syntax-directed by the structure of δ_τ. The second judgment (described in Figures 3.24 and 3.25) describes how a term is repaired when we no longer make decisions based on its old type and type diff: this happens for constructs whose type does not inform us about the type of its constituents (for instance, function application), and for simple terms (holes, universes, variables).

Notation

The first judgment can be read as follows:

- **Top**: Given that a term t had type τ in local context Γ and global environment E,
- **Bottom left**: and the global environment underwent modification δ_E,
- **Bottom left**: and the local context underwent modification δ_Γ,
- **Bottom right**: and the type underwent modification δ_τ,
- **Right**: then, we suggest to repair the term t with diff $\boxed{\delta_t}$.

The second judgment can be read analogously, but does not have access to typing information for the term being repaired. Importantly, the first judgment assumes the diff δ_τ is repaired already: it is the responsibility of the caller to ensure so.

Diff-directed term repair algorithm

R_{Term_1} (described in Figures 3.22 and 3.23) is a repair algorithm for terms, directed by the syntax of the diff for the type of the term being repaired. We will describe these rules one by one, as they are often quite complex.

Repair-Term-1-Same-Π

$$\textsc{Repair-Term-1-Same-}\Pi$$
$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ {}^{\text{Mod}}\Pi\ (\mathbb{1}_{\text{Binder}} : \mathbb{1}_{\text{Term}}) \to \mathbb{1}_{\text{Term}} \end{bmatrix}\right) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ \mathbb{1}_{\text{Term}} \end{bmatrix}\right) = \boxed{\delta}}$$

Repair-Term-1-Same-Π

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ \overset{\text{Mod}}{\Pi}\ (\mathbb{1}_{\text{Binder}} : \mathbb{1}_{\text{Term}}) \to \mathbb{1}_{\text{Term}} \end{bmatrix}\right) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ \mathbb{1}_{\text{Term}} \end{bmatrix}\right) = \boxed{\delta}}$$

Repair-Term-1-Same-Other

$$\frac{\text{Repair-Term-1-Same-}\Pi \text{ does not apply} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \tau \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta_t}}$$

Repair-Term-1-Mod-Π

$$\frac{\begin{array}{c} \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \text{Fresh}_1\left(\begin{bmatrix} x \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta_x} \qquad x \overset{\delta_x}{\rightsquigarrow} \boxed{x'} \\[1em] \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \text{Fresh}_2\left(\begin{bmatrix} x \\ \delta_x \end{bmatrix}, \begin{bmatrix} \chi \\ \delta_\chi \end{bmatrix}\right) = \boxed{z} \qquad \chi \overset{\delta_\chi}{\rightsquigarrow} \boxed{\chi'} \\[1em] \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (z : \tau_1) :: \Gamma \\ (\mathbb{K}(z) : \delta_{\tau_1}) \overset{\text{Mod}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t[x \leftarrow z] : \begin{bmatrix} \tau_2[\chi \leftarrow z] \\ \delta_{\tau_2}[\mathbb{K}(\chi') \leftarrow \mathbb{K}(z)] \end{bmatrix}\right) = \boxed{\delta_t} \end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(\lambda x \to t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ \overset{\text{Mod}}{\Pi}\ (\delta_\chi : \delta_{\tau_1}) \to \delta_{\tau_2} \end{bmatrix}\right) = \boxed{\overset{\text{Mod}}{\lambda}\ \delta_x \to \delta_t[\mathbb{K}(z) \leftarrow \mathbb{K}(x')]}}$$

Repair-Term-1-Ins-Π

$$\frac{\begin{array}{c} \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \text{Fresh}_1\left(\begin{bmatrix} \chi \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta_\chi} \qquad \chi \overset{\delta_\chi}{\rightsquigarrow} \boxed{x} \qquad \tau \overset{\delta_{\tau_1}}{\rightsquigarrow} \boxed{\tau_1} \\[1em] \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ (x : \tau_1) \overset{\text{Ins}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \tau \\ \delta_{\tau_2}[\mathbb{K}(\chi) \leftarrow \mathbb{K}(x)] \end{bmatrix}\right) = \boxed{\delta_t} \end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \tau \\ \overset{\text{Ins}}{\Pi}\ (\chi : \delta_{\tau_1}) \to \delta_{\tau_2} \end{bmatrix}\right) = \boxed{\overset{\text{Ins}}{\lambda}\ x \to \delta_t}}$$

Figure 3.22. Rules for repairing terms, diff-directed (R_{Term_1}, part 1/2)

Repair-Term-1-Drop-Π

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \mathsf{Fresh}_2(\begin{bmatrix} x \\ \mathbb{1} \end{bmatrix}, \begin{bmatrix} \chi \\ \mathbb{1} \end{bmatrix}) = \boxed{z} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (z : \tau_1) :: \Gamma \\ \overset{\text{Drop}}{::} \ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t[x \leftarrow z] : \begin{bmatrix} \tau[\chi \leftarrow z] \\ \delta_{\tau_2} \end{bmatrix}) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(\lambda x \to t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ \overset{\text{Drop}}{\Pi} \ \delta_{\tau_2} \end{bmatrix}) = \boxed{\overset{\text{Drop}}{\lambda} \ \delta_t}}$$

Repair-Term-1-Permute-Πs

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(\lambda \overline{x_{p(i)}}^{i \in \{1, \ldots, |p|\}} \to t : \begin{bmatrix} \Pi \overline{(\chi_{p(i)} : \tau_{p(i)})}^{i \in \{1, \ldots, |p|\}} \to \tau \\ \delta_\tau \end{bmatrix}) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(\lambda \overline{x_i}^{i \in \{1, \ldots, |p|\}} \to t : \begin{bmatrix} \Pi \overline{(\chi_i : \tau_i)}^{i \in \{1, \ldots, |p|\}} \to \tau \\ \overset{p}{\rightleftarrows} \\ \Pi \delta_\tau \end{bmatrix}) = \boxed{\begin{matrix} \overset{p}{\rightleftarrows} \\ \lambda \, \delta \end{matrix}}}$$

Repair-Term-1-Replace

$$\frac{}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t : \begin{bmatrix} \tau \\ \mathbb{K}(\tau') \end{bmatrix}) = \boxed{\mathbb{K}(\_ : \tau')}}$$

Repair-Term-1-Otherwise

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t : \begin{bmatrix} \tau \\ \delta_\tau \end{bmatrix}) = \boxed{\delta_t}}$$

Figure 3.23. Rules for repairing terms, diff-directed (R_{Term_1}, part 2/2)

This rule applies when the type of the term being repaired used to be a dependent function type, such as $\Pi(\chi : \tau_1) \rightarrow \tau_2$, and this type has not been modified. In order to not repeat ourselves, this rule simply expands $\mathbb{1}_{\text{Term}}$ into the equivalent $\overset{\text{Mod}}{\Pi}(\mathbb{1}_{\text{Binder}} : \mathbb{1}_{\text{Term}}) \rightarrow \mathbb{1}_{\text{Term}}$, so that rule Repair-Term-1-Mod-Π applies.

Repair-Term-1-Same-Other

Repair-Term-1-Same-Other

$$\frac{\textsc{Repair-Term-1-Same-}\Pi \text{ does not apply} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \tau \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta_t}}$$

This rule applies when the type of the term being repaired has not been modified, for all other cases than a dependent function type. Naively, one might expect the output to simply be $\boxed{\mathbb{1}}$, however, this would not account for changes to definitions in the global environment and local context, that may have repercussions within t. The rule therefore defers to the term-directed repair algorithm R_{Term_2}.

Repair-Term-1-Mod-Π

$$\textsc{Repair-Term-1-Mod-}\Pi\quad \dfrac{\begin{array}{c}\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \mathrm{Fresh}_1(\begin{bmatrix} x \\ \mathbb{1} \end{bmatrix}) = \boxed{\delta_x} \qquad x \overset{\delta_x}{\rightsquigarrow} \boxed{x'} \\[2ex] \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \mathrm{Fresh}_2(\begin{bmatrix} x \\ \delta_x \end{bmatrix}, \begin{bmatrix} \chi \\ \delta_\chi \end{bmatrix}) = \boxed{z} \qquad \chi \overset{\delta_\chi}{\rightsquigarrow} \boxed{\chi'} \\[2ex] \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (z : \tau_1) :: \Gamma \\ (\mathbb{K}(z) : \delta_{\tau_1}) \overset{\mathrm{Mod}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\mathrm{Term}_1}(t[x \leftarrow z] : \begin{bmatrix} \tau_2[\chi \leftarrow z] \\ \delta_{\tau_2}[\mathbb{K}(\chi') \leftarrow \mathbb{K}(z)] \end{bmatrix}) = \boxed{\delta_t}\end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\mathrm{Term}_1}(\lambda x \rightarrow t : \begin{bmatrix} \Pi(\chi : \tau_1) \rightarrow \tau_2 \\ \overset{\mathrm{Mod}}{\Pi}\ (\delta_\chi : \delta_{\tau_1}) \rightarrow \delta_{\tau_2} \end{bmatrix}) = \boxed{\overset{\mathrm{Mod}}{\lambda}\ \delta_x \rightarrow \delta_t[\mathbb{K}(z) \leftarrow \mathbb{K}(x')]}}$$

This rule requires a lot of care. It applies when the term is an explicit term abstraction, of the form $\lambda x \rightarrow t$, its type is an explicit type abstraction, of the form $\Pi(\chi : \tau_1) \rightarrow \tau_2$, and the diff to its type is an explicit modification of the type abstraction, of the form $\overset{\mathrm{Mod}}{\Pi}\ (\delta_\chi : \delta_{\tau_1}) \rightarrow \delta_{\tau_2}$.

Perhaps surprisingly, the first thing we'll do is possibly find a fresh name for x. Consider that the term abstraction under repair could be $(\lambda x \rightarrow f x)$, where f was a function in scope. Now consider what happens in the rare but possible case where the user renamed f into x. If we aim to repair the term abstraction, we will want to rename the occurrence of f into x, but it would be wrong to do so, because this x would be accidentally captured by the term abstraction! In order to repair such an f within the term abstraction, we had better alpha-rename it first, using a suitably fresh variable, say $(\lambda y \rightarrow f y)$. This lets us safely perform the renaming, obtaining $(\lambda y \rightarrow x y)$. We use a helper function, Fresh$_1$, in order to detect the conditions within which the variable x

would require freshening. The output of the helper has type Δ_{Binder}, and will be either $\boxed{\mathbb{1}}$ when the variable does not need to be refreshed, or $\boxed{\mathbb{K}(y)}$ when the variable needs to be refreshed and y is a suitable fresh variable.

A similar issue arises when we want to repair the body of the abstraction: we want to introduce the variable being abstracted over in our local context, but it already has two names, x in the term abstraction, and χ in the type abstraction. For the same reason as in the previous paragraph, neither x, nor χ, might always be suitable in the repaired program. We need a new helper function, Fresh_2, to pick a suitably fresh variable, $\boxed{z}$, that can be substituted in both the term abstraction *and* the type abstraction without accidental capture. In many cases, z will simple be x or χ, but in some cases, it might need to be a new variable.

Now armed with $\boxed{\delta_x}$ and $\boxed{z}$, we can introduce $(z : \tau_1)$ in the local context. As for the local context diff, we preserve the binding, using $(\mathbb{K}(z) : \delta_{\tau_1})$. Using $\mathbb{K}(z)$ for the binder diff ensures that all occurrences of z, that is, old occurrences of x, get properly rewritten into z. We must now perform appropriate substitutions so as to keep the body of the term abstraction, the body of the type abstraction, and the diff of the body the type abstraction, all in agreement. For the body of the term abstraction, we substitute z for x. For the body of the type abstraction, we substitute z for χ. But how should we alter the diff of this body, that is, δ_{τ_2}? Looking back at the provenance of δ_{τ_2}, we see it is a diff that has been created under the assumption that variable χ was to undergo modification δ_χ. However, we changed this resolution so that variable χ will instead become x. Therefore, within δ_{τ_2}, we should replace occurrences of χ', the variable that δ_{τ_2} was expecting, to z, the variable that will actually be in scope. Since the only diff construct that can mentions variables is the "replace" atomic diff operation, $\mathbb{K}(\ldots)$, this means we only need to substitute $\mathbb{K}(z)$ everywhere we see a free $\mathbb{K}(\chi')$.

Repairing the body of the term abstraction gives us back a diff, $\boxed{\delta_t}$. However, remember that this diff was built with z substituted for the bound variable. Repairing the term abstraction does *not* require renaming x to z though, this was only necessary to account for the dependent function type in repairing the body. In fact, remember we had decided that variable x should change according to δ_x. We can therefore build the appropriate output diff by replacing the binder following δ_x, and performing a diff substitution within δ_t, substituting $\mathbb{K}(x')$ everywhere we see a free $\mathbb{K}(z)$, where x' is the variable chosen by δ_x.

Repair-Term-1-Ins-Π

Repair-Term-1-Ins-Π

$$\dfrac{\begin{array}{c}\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \mathsf{Fresh}_1(\begin{bmatrix} \chi \\ \mathbb{1} \end{bmatrix}) = \boxed{\delta_\chi} \qquad \chi \overset{\delta_\chi}{\rightsquigarrow} \boxed{x} \qquad \tau \overset{\delta_{\tau_1}}{\rightsquigarrow} \boxed{\tau_1} \\[2ex] \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ (x : \tau_1) \overset{\text{Ins}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t : \begin{bmatrix} \tau \\ \delta_{\tau_2}[\mathbb{K}(\chi) \leftarrow \mathbb{K}(x)] \end{bmatrix}) = \boxed{\delta_t}\end{array}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t : \begin{bmatrix} \tau \\ \overset{\text{Ins}}{\Pi}(\chi : \delta_{\tau_1}) \to \delta_{\tau_2} \end{bmatrix}) = \boxed{\overset{\text{Ins}}{\lambda}\, x \to \delta_t}}$$

This rule applies when the type of a term t has changed to include a new type abstraction, as in $\overset{\text{Ins}}{\Pi}(\chi : \delta_{\tau_1}) \to \delta_{\tau_2}$. To allow the term to receive this new argument, we will want to turn it into a term abstraction. But what name should we pick for the variable to abstract over? The obvious candidate is χ, the same variable that appears in the type abstraction. However, once again, there is a chance that some unrelated χ appears free in t. Abstracting over the same variable would result in an accidental capture! We must pick a suitably fresh name, possibly χ, in the current context. This is once again achieved using Fresh$_1$, and we obtain a diff, $\boxed{\delta_\chi}$, which will either be $\mathbb{1}$ if it is possible to name the binder χ, or $\mathbb{K}(y)$ with a suitably fresh y otherwise. Let us name

$\boxed{x}$ the variable name that was picked.

We can compute the type that this x should have, τ_1, and add a binding $(x : \tau_1)$ to the local context as we repair t. Once again, we must perform a diff substitution, because δ_{τ_2} was built under the assumption that the value abstracted over would be named χ, but we instead chose to name it x. The resulting $\boxed{\delta_t}$ is how we want to repair the body of our soon-to-be term abstraction. The final output is simply $\boxed{\overset{\text{Ins}}{\lambda}\ x \to \delta_t}$. We do *not* need to undo the diff substitution, because δ_t indeed appears under a binder named x.

Repair-Term-1-Drop-Π

Repair-Term-1-Drop-Π

$$\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \text{Fresh}_2(\begin{bmatrix} x \\ \mathbb{1} \end{bmatrix}, \begin{bmatrix} \chi \\ \mathbb{1} \end{bmatrix}) = \boxed{z}$$

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} (z : \tau_1) :: \Gamma \\ \overset{\text{Drop}}{::}\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t[x \leftarrow z] : \begin{bmatrix} \tau[\chi \leftarrow z] \\ \delta_{\tau_2} \end{bmatrix}) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(\lambda x \to t : \begin{bmatrix} \Pi(\chi : \tau_1) \to \tau_2 \\ \overset{\text{Drop}}{\Pi}\ \delta_{\tau_2} \end{bmatrix}) = \boxed{\overset{\text{Drop}}{\lambda}\ \delta_t}}$$

This rules applies when some explicit term abstraction, $\lambda x \to t$, had an explicit type abstraction for its type, $\Pi(\chi : \tau_1) \to \tau_2$, but no longer receives this argument. In this case, we will want to drop the , but repair its body. Once again, we will use Fresh_2 to pick a suitably fresh variable that will neither be captured by x in t, nor by χ in τ_2. We simply add the binding $(z : \tau_1)$ to the local context, and $\overset{\text{Drop}}{::}$ for its diff. Thanks to our lookup rules (from Section 3.5.2), any reference to z will be replaced with a deprecated variable name. We simply need to repair the body at the appropriate type and type diff. The result, $\boxed{\delta_\tau}$, will contain both repairs to t, and remove all references to the now obsolete binder x, replacing those with some variable name, depending on the implementation of Deprecate.

Repair-Term-1-Permute-Πs

Repair-Term-1-Permute-Πs

$$\dfrac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(\lambda \overline{x_{p(i)}}^{i \in \{1,\ldots,|p|\}} \rightarrow t : \begin{bmatrix} \Pi\overline{(\chi_{p(i)} : \tau_{p(i)})}^{i \in \{1,\ldots,|p|\}} \rightarrow \tau \\ \delta_\tau \end{bmatrix}\right) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(\lambda \overline{x_i}^{i \in \{1,\ldots,|p|\}} \rightarrow t : \begin{bmatrix} \Pi\overline{(\chi_i : \tau_i)}^{i \in \{1,\ldots,|p|\}} \rightarrow \tau \\ \overset{p}{\rightleftarrows} \\ \Pi\delta_\tau \end{bmatrix}\right) = \boxed{\begin{matrix} \overset{p}{\rightleftarrows} \\ \lambda\,\delta \end{matrix}}}$$

This rule applies when the term being repaired is a sequence of term abstractions $\lambda \overline{x_i}^{i \in \{1,\ldots,|p|\}} \rightarrow t$, while its type used to be some telescope of type abstractions $\Pi\overline{(\chi_i : \tau_i)}^{i \in \{1,\ldots,|p|\}} \rightarrow \tau$, and its type has undergone a permutation $\overset{p}{\overset{\rightleftarrows}{\Pi}}\delta_\tau$. Note that the sequence of term abstractions, as well as the telescope of type abstractions, could be longer than the length of the permutation, and of different lengths. All that matters is that we have *at least* $|p|$ explicit λs in the term, and *at least* $|p|$ explicit Πs in the type abstraction, so that we can perform the permutation. When this is possible, the rule is fairly simple: it applies the permutation to both the term and the type, and repairs the resulting term against the resulting type. The output of this result is then permuted to obtain the result of the original problem.

Repair-Term-1-Replace

$$\dfrac{}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t : \begin{bmatrix} \tau \\ \mathbb{K}(\tau') \end{bmatrix}) = \boxed{\mathbb{K}(\_ : \tau')}}\ \textsc{Repair-Term-1-Replace}$$

When the type of a term has been replaced by an arbitrary type τ', there is nothing meaningful that can be done to repair the old term. We have to give up and return a typed hole, $(\_ : \tau')$.

Repair-Term-1-Otherwise

$$\dfrac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(t : \begin{bmatrix} \tau \\ \delta_\tau \end{bmatrix}) = \boxed{\delta_t}}\ \textsc{Repair-Term-1-Otherwise}$$

Finally, when none of the other rules apply, we have exhausted the ways in which we could use the type to guide the repair. There are still opportunities for repairing the term, but they are no longer directed by its type, only by the syntax of the term itself. Therefore, we simply forget about the type information, and call R_{Term_2}, which we will discuss now.

Term-directed term repair algorithm

R_{Term_2} (described in Figures 3.25 and 3.25) is an algorithm directed by the syntax of the term being repaired, and is used when no information is known about its type, or how that type might have changed. Let us again go through the rules one by one.

Repair-Term-2-Variable

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \begin{bmatrix} v \\ \boxed{\delta_v} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_v}} \end{bmatrix} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Arguments}}([], \tau_v, \delta_{\tau_v}, \delta_v) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(v) = \boxed{\delta}}$$

Repair-Term-2-Application-Variable

$$\frac{\begin{gathered} E, \Gamma \vdash f : \boxed{\tau_f} \\ \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \begin{bmatrix} f \\ \boxed{\delta_f} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_f}} \end{bmatrix} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Arguments}}(\overline{a_i}^{i \in \{1,\ldots,n\}}, \tau_f, \delta_{\tau_f}, \delta_f) = \boxed{\delta} \end{gathered}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(f\ \overline{a_i}^{i \in \{1,\ldots,n\}}) = \boxed{\delta}}$$

Repair-Term-2-Application-Term

$$\frac{\begin{gathered} t \text{ is not a variable} \\ \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t} \qquad \overline{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(a_i) = \boxed{\delta_{a_i}}}^{i \in \{1,\ldots,n\}} \end{gathered}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t\ \overline{a_i}^{i \in \{1,\ldots,n\}}) = \boxed{\delta_t\ \overline{\overset{\text{Mod}}{\$}\ \delta_{a_i}}^{i \in \{1,\ldots,n\}}}}$$

Repair-Term-2-Π

$$\frac{\begin{gathered} \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \text{Fresh}_1(\begin{bmatrix} x \\ \mathbb{1} \end{bmatrix}) = \boxed{\delta_x} \qquad \begin{bmatrix} e \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(\tau_1 : \begin{bmatrix} \text{Type} \\ \mathbb{1} \end{bmatrix}) = \boxed{\delta_{\tau_1}} \\ \begin{bmatrix} e \\ \delta_E \end{bmatrix}, \begin{bmatrix} (x : \tau_1) :: \Gamma \\ (\delta_x : \delta_{\tau_1}) \overset{\text{Mod}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}(\tau_2 : \begin{bmatrix} \text{Type} \\ \mathbb{1} \end{bmatrix}) = \boxed{\delta_{\tau_2}} \end{gathered}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(\Pi(x : \tau_1) \to \tau_2) = \boxed{\overset{\text{Mod}}{\Pi}\ (\delta_x : \delta_{\tau_1}) \to \delta_{\tau_2}}}$$

Repair-Term-2-λ

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \text{Fresh}_1(\begin{bmatrix} x \\ \mathbb{1} \end{bmatrix}) = \boxed{\delta_x} \qquad \begin{bmatrix} e \\ \delta_E \end{bmatrix}, \begin{bmatrix} x :: \Gamma \\ \delta_x \overset{\text{Mod}}{::} \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(\lambda x \to t) = \boxed{\overset{\text{Mod}}{\lambda}\ \delta_x \to \delta_t}}$$

Figure 3.24. Rules for repairing terms, term-directed (R_{Term_2})

Repair-Term-2-Match

$$\frac{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t} \qquad \begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Branches}}(t, [b_1 \dots b_n]) = \boxed{\delta_b}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_2}(\,\texttt{match}\ t\ \texttt{with}\ b_1 \dots b_n) = \boxed{\delta}}$$

Repair-Term-2-Annotation

$$\frac{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_1}\left(\tau : \begin{bmatrix}\texttt{Type}\\ \mathbb{1}\end{bmatrix}\right) = \boxed{\delta_\tau} \qquad \begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix}\tau\\ \delta_\tau\end{bmatrix}\right) = \boxed{\delta_t}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_2}(t : \tau) = \boxed{\delta_t : \delta_\tau}}$$

Repair-Term-2-Otherwise

$$\frac{}{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\mathbb{1}}}$$

Figure 3.25. Rules for repairing terms, term-directed (R_{Term_2}, part 2/2)

Repair-Term-2-Variable

Repair-Term-2-Variable

$$\frac{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \begin{bmatrix}v\\ \boxed{\delta_v}\end{bmatrix} : \begin{bmatrix}?\\ \boxed{\delta_{\tau_v}}\end{bmatrix} \qquad \begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Arguments}}([],\tau_v,\delta_{\tau_v},\delta_v) = \boxed{\delta}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix},\begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\text{Term}_2}(v) = \boxed{\delta}}$$

This rule applies when repairing a variable. By using our lookup rules (from Section 3.5), we can hope to obtain a diff for v, $\boxed{\delta_v}$, telling us whether it has been renamed or not. Naively, we might want to call this the answer, and move on. But, our lookup also lets us know how the type of v has changed, captured in the output $\boxed{\delta_{\tau_v}}$. And it could be that the type of v has changed, for instance, from a constant to a function of some number of arguments! Were this the case, repairing an occurrence of v properly would require adding values for these new arguments.

This is just a special case of repairing a function application, that we will see in the next rule (Rule Repair-Term-2-Application-Variable). We will solve the problem by using a helper function, $R_{\text{Arguments}}$, which repairs an arbitrary function applied to some number of arguments. A variable is simply a special case where the list of arguments is empty, i.e. [].

Repair-Term-2-Application-Variable

Repair-Term-2-Application-Variable

$$\frac{E, \Gamma \vdash f : \boxed{\tau_f} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash \begin{bmatrix} f \\ \boxed{\delta_f} \end{bmatrix} : \begin{bmatrix} ? \\ \boxed{\delta_{\tau_f}} \end{bmatrix} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Arguments}}(\overline{a_i}^{i \in \{1,\ldots,n\}}, \tau_f, \delta_{\tau_f}, \delta_f) = \boxed{\delta}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(f\ \overline{a_i}^{i \in \{1,\ldots,n\}}) = \boxed{\delta}}$$

This rule applies when the term is a function application. Repairing nested function applications is not as trivial as it could seem. Consider some function f, whose type used to be $(A \to C)$, and undergoes the transformation:

$$\delta_\tau = \overset{\text{Mod}}{\Pi}\ (\mathbb{1} : \mathbb{1}) \to (\overset{\text{Ins}}{\Pi}(\_ : B) \to \mathbb{1})$$

yielding the type $(A \to B \to C)$. Suppose the original code contains a call to f, for instance $(f\ a)$. The repaired function call ought to be $(f\ a\ (\_ : B))$. This means that the repair diff ought to be:

$$\delta_t = ((\mathbb{1} \overset{\text{Mod}}{\$} \mathbb{1}) \overset{\text{Ins}}{\$} (\_ : B))$$

Notice how the outermost term modification in δ_t, i.e. $\overset{\text{Ins}}{\$}$, corresponds to the innermost type modification in δ_τ, i.e. $\overset{\text{Ins}}{\Pi}$. This is not surprising: a function g whose type is $(A \rightarrow (B \rightarrow (C \rightarrow D)))$, and its application to arguments $(((g\ a)\ b)\ c)$, exhibit the same inversion. For this reason, $R_{\text{Arguments}}$ works by processing the Π-telescope type diff (a sequence of nested modification of Πs, such as δ_τ) from outside-in, and builds the term applications diff (such as δ_t) from inside-out.

In order to do so, we syntactically extract as many nested applications as possible, yielding a sequence of arguments of some arbitrary length $\overline{a_i}^{i\in\{1,\ldots,n\}}$, and pass this array of arguments to $R_{\text{Arguments}}$.

We omit the rules for $R_{\text{Arguments}}$ as they would be cumbersome. The algorithm is first directed by the diff of the function's type, performing the appropriate addition, modification, deletion, and permutation of arguments, while also repairing existing arguments. Care is taken in keeping track of the unprocessed list of arguments, so that when the function's type diff becomes $\mathbb{1}$, we can introduce the appropriate amount of $\overset{\text{Mod}}{::}$ in the result. For instance, consider the following change and repair, and their diffs:

In this figure, we annotate the instances of $\mathbb{1}$ with a subscript indicating the term that they are preserving. Notice how, in the type, we have $\mathbb{1}_{\texttt{C → D → E}}$, but in the diff for `x`, we have no choice but to expand it to a sequence of $\overset{\text{Mod}}{\$}$ operations, one for each argument, because of the reversed order. In practice, this can be achieved in one of two ways:

- by pre-processing instances of $\mathbb{1}$ in input types, and expanding them to the diffs that they stand for. For instance here, we would expand $\mathbb{1}_{\texttt{C → D → E}}$ into the equivalent, but more verbose $\overset{\text{Mod}}{\Pi}(\mathbb{1} : \mathbb{1}_{\texttt{C}}) \rightarrow \overset{\text{Mod}}{\Pi}(\mathbb{1} : \mathbb{1}_{\texttt{D}}) \rightarrow \mathbb{1}_{\texttt{E}}$,

```
Definition f : A → C → D → E → F :=
 ↪
…
.

Definition x := f a c d e.
```

```
Definition f : A → B → C → D → E →
 ↪  F :=
…
.

Definition x := f a (_ : B) c d e.
```

$$\delta_{\tau_f} = \overset{\text{Mod}}{\Pi}\,(\mathbb{1} : \mathbb{1}_{\texttt{A}}) \to \overset{\text{Ins}}{\Pi}(B : A) \to \mathbb{1}_{\texttt{C → D → E}}$$

$$\delta_x = \mathbb{1} \overset{\text{Ins}}{\$}\ \texttt{(\_ : B)}\ \overset{\text{Mod}}{\$}\ \mathbb{1}_{\texttt{c}} \overset{\text{Mod}}{\$}\ \mathbb{1}_{\texttt{d}} \overset{\text{Mod}}{\$}\ \mathbb{1}_{\texttt{e}}$$

Figure 3.26. Modification and repair of a function and function call, and their respective diff

- or by keeping track of the number of arguments in the $R_{\text{Arguments}}$ algorithm, and processing $\mathbb{1}$ as if it was a $\overset{\text{Mod}}{\Pi}\,(:) \to$ for as long as we have arguments remaining (essentially dynamically emulating the previous option).

We present the rules following the latter approach.

Repair-Term-2-Application-Term

$$\textsc{Repair-Term-2-Application-Term}$$

$$\dfrac{t \text{ is not a variable} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t} \qquad \overline{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(a_i) = \boxed{\delta_{a_i}}}^{i \in \{1,\ldots,n\}}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t\ \overline{a_i}^{i \in \{1,\ldots,n\}}) = \boxed{\overline{\delta_t \overset{\text{Mod}}{\$} \delta_{a_i}}^{i \in \{1,\ldots,n\}}}}$$

This rule applies when we are trying to repair a function application that does not fit the previous rule: its leftmost nested node that is not an application node is not a variable either, but some other term. An example of such term is an immediately applied function obtained by pattern matching:

```
(match b with true => f | false => g end) x
```

Because we do not know the type of the function being applied, we cannot repair the function applications properly. We simply repair each argument, and leave the sequence of applications unchanged. A more interesting repair would consist in first inferring the type of t, then repairing it to obtain a diff for its type, and then calling $R_{\text{Arguments}}$. We have not yet explored this option.

Repair-Term-2-Π

$$\textsc{Repair-Term-2-}\Pi\quad \dfrac{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash \mathrm{Fresh}_1\left(\begin{bmatrix}x\\ \mathbb{1}\end{bmatrix}\right) = \boxed{\delta_x} \qquad \begin{bmatrix}e\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\mathrm{Term}_1}\left(\tau_1 : \begin{bmatrix}\mathrm{Type}\\ \mathbb{1}\end{bmatrix}\right) = \boxed{\delta_{\tau_1}} \\ \begin{bmatrix}e\\ \delta_E\end{bmatrix}, \begin{bmatrix}(x : \tau_1) :: \Gamma\\ (\delta_x : \delta_{\tau_1}) \overset{\mathrm{Mod}}{::} \delta_\Gamma\end{bmatrix} \vdash R_{\mathrm{Term}_1}\left(\tau_2 : \begin{bmatrix}\mathrm{Type}\\ \mathbb{1}\end{bmatrix}\right) = \boxed{\delta_{\tau_2}}}{\begin{bmatrix}E\\ \delta_E\end{bmatrix}, \begin{bmatrix}\Gamma\\ \delta_\Gamma\end{bmatrix} \vdash R_{\mathrm{Term}_2}(\Pi(x : \tau_1) \to \tau_2) = \boxed{\overset{\mathrm{Mod}}{\Pi}\ (\delta_x : \delta_{\tau_1}) \to \delta_{\tau_2}}}$$

This rule applies to repair a dependent function space. It is a somewhat straightforward process. Once again, we need to possibly refresh the binder, using Fresh_1, for the same reasons as explained for Rule Repair-Term-1-Mod-Π.

Repair-Term-2-Match

Repair-Term-2-Match

$$\frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\delta_t} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Branches}}(t,\ [b_1 \ldots b_n]) = \boxed{\delta_b}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(\ \texttt{match}\ t\ \texttt{with}\ b_1 \ldots b_n) = \boxed{\delta}}$$

This rule applies to repair a `match` construct. The repair process is quite complex, so we abstract over it with the R_{Branches} helper function. The discriminee, t, can be readily repaired.

Subsequently, we need to repair the branches. There are two complications:

1. We need to find what type is being matched, so that we can look up its constructors and how they have changed. However, the type never appears explicitly in the program.

2. We would like to keep the existing branches in the same order as they appear in the user's program, which can be different from the order in which the constructors are declared in the data type.

To resolve the first issue, we use a sequence of heuristics: we can look up the type of t, which might be the inductive type. However, remember that in a dependently-typed language, the type of t could be an arbitrary computation, rather than simply a concrete type constructor applied to its arguments. Our second heuristic looks at the constructors in the branches, and looks for an inductive data type in scope that has these constructors. This is more involved, but should cover cases where the type of t is not helpful.

Once we know the inductive type of t, we are ready to attack the second problem. Let us consider a concrete case to make matters clear. Consider the code:

```
Inductive T : Type := A : T | B : T | C : T.

Definition code (t : T) : … :=
  match t with
  | C => …
  | A => …
  | B => …
  end.
```

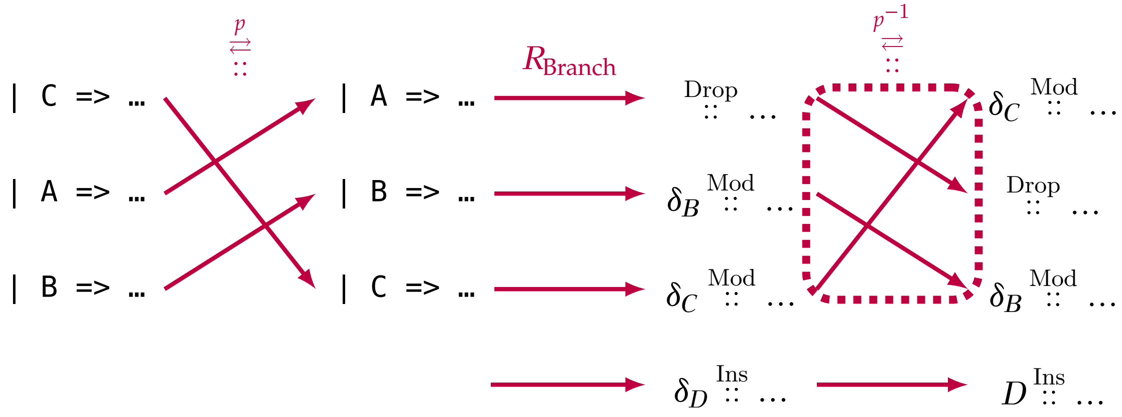
And let's consider the following modification to inductive data type `T`:

```
Inductive T : Type := B : T | C : T | D : T.
```

where constructor `A` has been removed, and constructor `D` has been added. We would like the repair to be:

```
Definition code (t : T) : … :=
  match t with
  | C => …
  (* | A => … *) (* this branch should be deleted *)
  | B => …
  | D => …       (* this branch should be added *)
  end.
```

A simple way to obtain this result is to find a permutation from the old list of branches to the old list of constructors, and a mapping from the old list of constructors to the new list of constructors, as depicted in Figure 3.27. We first compute the permutation p that will let us reorder the patterns as they appear in the user's program, to the order in which they appear in the data type declaration. Now that they are in the same order, and we know the inductive data type being modified, we can also obtain the diff to that inductive data type, and in particular, the diff to the list of constructors. From this, it is easy to compute a diff-continuation, represented in Figure 3.27 as a list diff operation with "…".



| C => …
| A => …
| B => …
p
$\rightleftarrows$
::
| A => …
| B => …
| C => …
R_{Branch}
$\overset{\text{Drop}}{::}$ …
$\delta_B \overset{\text{Mod}}{::}$ …
$\delta_C \overset{\text{Mod}}{::}$ …
$\delta_D \overset{\text{Ins}}{::}$ …
p^{-1}
$\rightleftarrows$
::
$\delta_C \overset{\text{Mod}}{::}$ …
$\overset{\text{Drop}}{::}$ …
$\delta_B \overset{\text{Mod}}{::}$ …
$D \overset{\text{Ins}}{::}$ …

Figure 3.27. Repairing branches of a `match` using permutations

For instance, since the `A` constructor was removed, we can compute that the `A` branch must be removed. Had the `B` constructor been modified, we could compute a relevant δ_B patch to repair the branch. In our simple example where it has not changed, we have $\delta_B = \mathbb{1}$. For new constructors, like `D`, we can just fabricate empty branches (with the proper constructor and arity for each pattern) and append them last.

We have not yet extended our language to dependent pattern matching of the form:

```
match … as … in … return … with
| …
end
```

as found in *Coq*, but we believe it should not pose a significant challenge: the `as` clause, `in` clause, and `return` clause, should be repaired in order, with proper care taken about what variables come in and out of scope. They can be repaired independently from the discriminee and the branches. When an `in` clause is present, it would in fact tell us the type of the discriminee! When a `return` clause is present, however, we might be able to use R_{Term_1}, rather than R_{Term_2}, to repair the body of each branch, since we know what type to expect from the branch!

Our treatment of match has a great advantage over manual refactoring in the presence of a wildcard pattern. A common source of problems when refactoring a data type definition happens when some code uses wildcard patterns. Consider the following code:

```
Inductive T : Type := A : T | B : T | C : T.

(* then, somewhere far from this definition *)

Definition handle (t : T) : bool :=
  match t with
  | A => true
  | _ => false (* everything else is false! *)
  end.
```

If the programmer later decides to extend the datatype to the following:

```
Inductive T : Type := A : T | B : T | C : T | D : T.
```

then the old version of the handle function will happily type-check, mapping input D to output false , which might not have been the intent of the programmer. Pattern matches that does not use wildcard patterns, on the other hand, will complain that the D case is not handled. For this reason, wildcard patterns are often considered bad practice. However, with our technique, the same program will naturally be repaired into:

```
Inductive T : Type := A : T | B : T | C : T | D : T.

Definition handle (t : T) : bool :=
  match t with
  | A => true
  | D => (_ : bool)
  | _ => false (* everything else is false! *)
  end.
```

which will force the user to consider whether they want this branch to return true or false . In some sense, it makes working with wildcard patterns safer, as long as all refactoring attempts are carried through our tool, of course.

We omit the rules for R_{Branches}, but the high-level algorithm consists in:

- heuristically figure out the type of the discriminee t,
- perform the transformation described in Figure 3.27.

Repair-Term-2-Annotation

$$\textsc{Repair-Term-2-Annotation}\quad \frac{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(\tau : \begin{bmatrix} \texttt{Type} \\ \mathbb{1} \end{bmatrix}\right) = \boxed{\delta_\tau} \qquad \begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_1}\left(t : \begin{bmatrix} \tau \\ \delta_\tau \end{bmatrix}\right) = \boxed{\delta_t}}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t : \tau) = \boxed{\delta_t : \delta_\tau}}$$

This rule applied when the term being repaired is a type-annotated term. Note that $(t : \tau)$ *is* the term being repaired here, the black colon operator being part of the syntax of *Chick* (described in Section 3.3.1), as opposed to the purple colon operator that we use in the previous judgment R_{Term_1}. This is quite straightforward: we first repair the type, then we repair the term. Note that, even though we repair the type with R_{Term_1}, there are no rules that apply to a type like `Type` , so it will defer back to R_{Term_2}. We only choose to present the rule with R_{Term_1} so that, if someone extended R_{Term_1} to be more clever in its treatment of `Type` , this could benefit from it.

Repair-Term-2-Otherwise

$$\textsc{Repair-Term-2-Otherwise}\quad \frac{}{\begin{bmatrix} E \\ \delta_E \end{bmatrix}, \begin{bmatrix} \Gamma \\ \delta_\Gamma \end{bmatrix} \vdash R_{\text{Term}_2}(t) = \boxed{\mathbb{1}}}$$

For all other cases, we do not yet perform any meaningful repair. The concrete terms that we do not attempt to repair are:

- immediately-applied term abstractions, which are harder than applied functions because we cannot simply look up the type of the term abstraction,
- holes, that can remain the same,
- and universes, that can remain the same.

3.7 Deriving repair functions

Accounting for changes in inductive definitions requires quite a bit of work:

- the inductive type itself must be repaired, including its parameters, and its indices,
- each constructor must be repaired, including their parameters, and their instantiation of indices,
- the automatically generated elimination principles for the type must also be repaired accordingly.

Let us focus on the latter. In a proof assistant like Coq, when an inductive data type `t` is defined, the system also defines a family of *elimination principles* `t_ind` , `t_rec` , and `t_rect` , that essentially encode an induction principle for the given type. For instance, for the types `list` and `vec` , the induction principles are given in Figure 3.28.

```
list_ind :
∀ (A : Type)           ①
  (P : list A →        ②
       Prop),
  P (nil A) →          ③
  (∀ (a : A)           ③
     (l : list A),
     P l →
     P (cons A a l)
  ) →
                       ④
  ∀ l : list A,        ⑤
    P l                ⑥
```

```
vec_ind :
∀ (A : Type)                     ①
  (P : ∀ n : nat, vec A n →      ②
         Prop),
  P 0 (vnil A) →                 ③
  (∀ (a : A) (n : nat)           ③
     (v : vec A n),
     P n v →
     P (S n) (vcons A a n v)
  ) →
  ∀ (n : nat)                    ④
    (v : vec A n),               ⑤
    P n v                        ⑥
```

Figure 3.28. Induction principles for `list` and `vec`

The property being returned `P` is usually referred to as the *motive* of the elimination principle (see McBride [25] for a thorough introduction to elimination with a motive). The motive is a property about a value of the inductive type being eliminated: we will refer to this value as the *target* of the motive. The value being eliminated (the last universally quantified value, respectively `l` and `v`) are usually referred to as the *discriminee*. The process that computes the elimination principle's type from the inductive data type definition Inductive$(\{n_{\text{ind}}, \overline{p_{\text{ind}}}, \overline{i_{\text{ind}}}, u_{\text{ind}}, \overline{c_{\text{ind}}}\})$ is involved, but straightforward:

① the inductive parameters $\overline{p_{\text{ind}}}$ are universally quantified over,

② the motive `P` is universally quantified over, its type being computed by:

- universally quantifying over the inductive indices $\overline{i_{\text{ind}}}$,

- universally quantifying over a target, i.e. an element of the inductive type undergoing elimination, fully instantiated with parameters from ①and the indices from the previous sub-step,

- and returning in the appropriate universe for the desired elimination principle,

③ for each constructor Constructor$(\{n_{\text{ctor}}, \overline{p_{\text{ctor}}}, \overline{\imath_{\text{ctor}}}\})$, a case is universally quantified over, whose type is obtained thus:

- each constructor parameter $\overline{p_{\text{ctor}}}$ is universally quantified over, and for each parameter that is a recursive occurrence of the data type being eliminated, a fully instantiated motive, with appropriate arguments so as to target this parameter, is also universally quantified over immediately after this parameter,

- the return type is also a fully instantiated motive, where inductive indices are instantiated with the constructor indices $\overline{\imath_{\text{ctor}}}$, and the motive's target is an instantiation of the constructor n_c, with inductive parameters from step ①, and constructor parameters from the previous sub-step,

④ inductive indices $\overline{\imath_{\text{ind}}}$ are universally quantified over,

⑤ the discriminee is universally quantified over, its type being the inductive type applied to inductive parameters from ①and inductive indices from ④,

⑥ finally, the elimination rule returns a value of the fully instantiated motive, with indices from ④, and the discriminee as target.

We will use Haskell syntax to describe meta-language implementation details (we reserve Coq syntax for programs in the object language). Our function computing the type of an eliminator has the following high-level skeleton:

```
eliminatorType … =
  quantifyVars  inductiveParameters                       {- 1 -}
. mkPi          motiveType           motive               {- 2 -}
. quantifyCases inductiveConstructors                     {- 3 -}
. quantifyVars  inductiveIndices                          {- 4 -}
. mkPi          discrimineeType      discriminee          {- 5 -}
$ mkApp         indices              discriminee          {- 6 -}
  where
   indices = applyVars inductiveIndices motive
```

where `quantifyVariables`, `applyVariables` are folding functions that create `Pi` s, and `App` s, respectively, and `motiveType` and `discrimineeType` are computed by additional `fold` operations.

Our insight is as follows: while we cannot derive a diff-propagating function for any `fold` in general, we can describe how a given folding operation `f` reacts to differences in the input list `l` in the call `(fold f l b)`. From this description, we can derive the diff-propagating function for the whole `fold` operation. To make things more concrete, let's focus on `quantifyVariables` and build its corresponding, diff-propagating version `δquantifyVariables`. The original function is a right-fold that universally quantifies over every element encountered. We can describe how it reacts to changes in the input list using the following record type:

```
data ΔListFold τ δτ δ = ΔListFold
  { onInsert  ::      τ →     [τ] → δ → δ
  , onModify  ::    δτ → τ → [τ] → δ → δ
  , onPermute :: [Int] →     [τ] → δ → δ
  , onRemove  ::          τ → [τ] → δ → δ
  , onReplace ::   [τ] →     [τ] → δ → δ
  , onSame    ::              [τ] → δ → δ
  }
```

For each diff operation on lists, we record a transformer on the output diff type `δ`. In our example, the fold is building a `Term`, so the output diff type is `ΔTerm`, and we describe how the output term is affected by changes to the input list: when an element is inserted in the input list, a Π is added to the output term, when an element is removed from the input list, a Π is removed from the output term, etc. When the entire list gets replaced, the handler receives the old list `l` and the new list `l'`: the output term will see `(length l')` Πs removed, and for each element of `l`, a corresponding Π will be added. We then have two functions:

```
δListFoldLeft, δListFoldRight ::
  ΔListFold τ δτ δ →
  [τ] → ΔList τ δτ → Maybe (δ → δ)
```

which take such a description, an original list, and a list diff, and compute the resulting diff transformer for the output type, as long as the diff makes sense for the list. Receiving the original list lets us check the sanity of the list diff at each step (e.g. a diff should not ask to drop the head from an empty list), and lets us pass the element being modified or removed to the handlers who need the information. With all this machinery ready, we can derive `δEliminatorType` as:

```
δEliminatorType ... =
    δquantifyVars  ips              δips
<$> CopyPi         δmotiveType      Same
<$> δquantifyCases cs               δcs
 $  δquantifyVars  iis              δiis
 $  CopyPi         δdiscrimineeType Same
 $  CopyApp        δindices         Same
  where
    δindices = δapplyVars iis δiis Same
```

which matches closely the original definition of `EliminatorType`. We use the same mechanism to derive a diff-propagating version of the function that computes the type of an inductive type family (based on the diff of its parameters and indices lists), and a diff-propagating version of the function that computes the type of a constructor (based on the diff of its parameters and indices lists). At a high-level, we have the following property:

$$\frac{\begin{array}{c}\text{EliminatorType}(\text{Inductive}(\{n, \vec{p}, \vec{i}, u, \vec{c}\})) = \boxed{e} \\ \text{Inductive}(\{n, \vec{p}, \vec{i}, u, \vec{c}\}) \overset{\delta_i}{\rightsquigarrow} \boxed{\text{Inductive}(\{n', \vec{p'}, \vec{i'}, u', \vec{c'}\})} \\ \Delta\text{EliminatorType}(\text{Inductive}(\{n, \vec{p}, \vec{i}, u, \vec{c}\}), \delta_i) = \boxed{\delta_e} \\ \text{EliminatorType}(\text{Inductive}(\{n', \vec{p'}, \vec{i'}, u', \vec{c'}\})) = \boxed{e'}\end{array}}{e \overset{\delta_e}{\rightsquigarrow} \boxed{e'}}$$

i.e., `δEliminatorType` produces the correct diff to transport the old eliminator type to the new eliminator type.

Remark

In fact, we even have the following corollary:

$$\frac{\begin{array}{c}\text{EliminatorType}(i) = \boxed{e} \\ \Delta\text{EliminatorType}(i, \delta_{\text{Inductive}}(\{\mathbb{K}(n), \mathbb{K}(\vec{p}), \mathbb{K}(\vec{i}), \mathbb{K}(u), \mathbb{K}(\vec{c})\})) = \boxed{\delta_e} \\ \text{EliminatorType}(\text{Inductive}(\{n, \vec{p}, \vec{i}, u, \vec{c}\})) = \boxed{e'}\end{array}}{e \overset{\delta_e}{\rightsquigarrow} \boxed{e'}}$$

which makes EliminatorType seem redundant, as it coincides exactly with the result of ΔEliminatorType when passed a diff replacing all pre-existing data from the inductive type.

3.8 Guessing diffs

We showed how our algorithm could take an initial program, and a diff describing how to obtain the partial refactoring, in order to generate a repaired diff with the refactoring carried out. However, we had postponed the discussion of how this original diff was obtained.

Automatically generating this diff turns out to be a hard problem. It is equivalent to the tree difference problem: given two abstract syntax trees, we want to find a mapping of pairs of nodes from the old tree and the new tree, such that we have high confidence that they are related nodes.

There is a bit of literature on the subject. We chose to use the *GumTree* algorithm, for two reasons. First, its has a worst-case time complexity of $O(n^2)$, where the worst time is unlikely to happen in real abstract syntax trees. Second, it works by ranking candidate pairs according to some heuristic similarity metric, which is useful in order to compute not just the most likely answer, but, possibly, a list of likely answers ranked by the heuristic likelihood.

Unfortunately, our early attempts of using the algorithm with our data types proved unsuccessful. Let us illustrate where the algorithm fails in our setting, by first giving a broad overview of how it works:

- First, all leaves are distinguished based on their identity. In our figures, we will denote nodes that are considered similar using colors. After this first pass, all leaves should be colored, such that they are similar to equal leaves.

- Then, all pairs of old internal nodes and new internal nodes are compared (in an efficient order) for similarity of their children. The similarity S between two

$$\mathsf{S}(n_1, n_2) = \frac{2\times \Big| \{c_1 \in C(n_1) \text{ such that } \exists c_2 \in C(n_2), c_1 \approx c_2\} \Big|}{\Big| \{c \in C(n_1)\} \Big| + \Big| \{c \in C(n_2)\} \Big|}$$

where $C(n)$ is the set of children of n
and $c_1 \approx c_2$ means that c_1 is considered similar to c_2

Figure 3.29. Similarity between two nodes

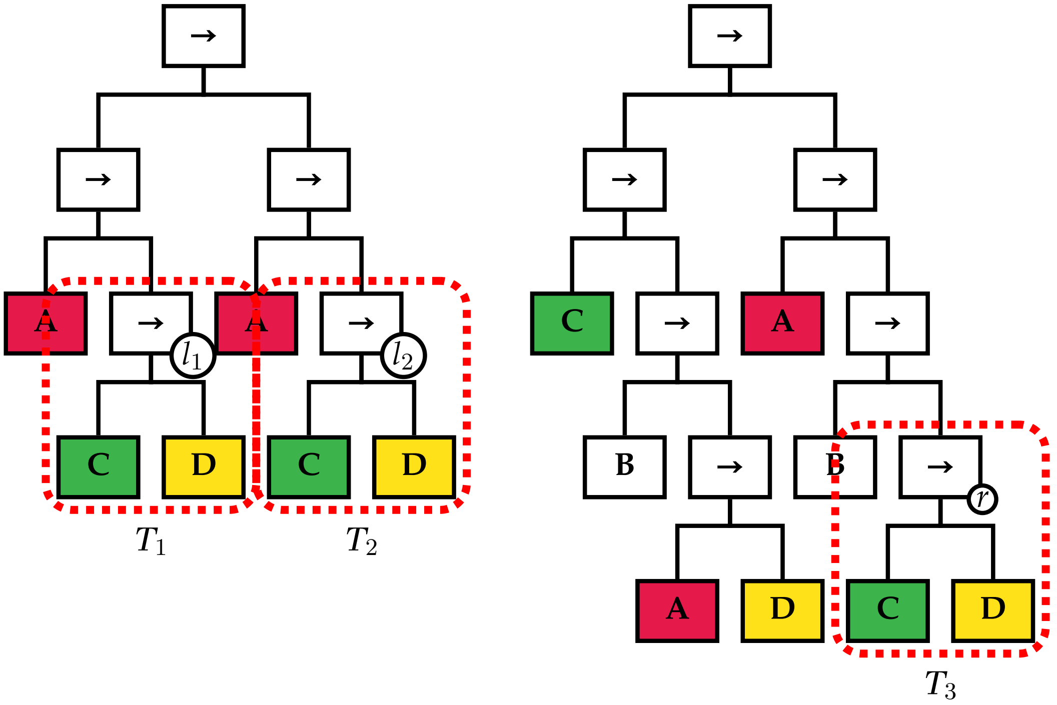
nodes is defined in Figure 3.29, and ranges between 0 (no shared children) to 1 (all children are shared). Note that it does not account for the order of the children.

- As items are found to be similar, they are added to a relation $n_1 \approx n_2$. A custom threshold allows us to tweak how similar two nodes need to be in order for our algorithm to consider them similar. Then, remaining pairs of nodes are considered by decreasing similarity, and added to the same relation.

Let us now see where this algorithm underperforms in our setting. Consider the abstract syntax trees in Figure 3.30. In it, we highlight three identical sub-trees, T_1, T_2, and T_3, in red dashed rounded rectangles. Due to being exactly the same sub-trees, their similarity score will be equal. Yet, to a human being, $T_2 \approx T_3$ seems more likely than $T_1 \approx T_3$: they are roughly in a similar location in the abstract syntax tree.

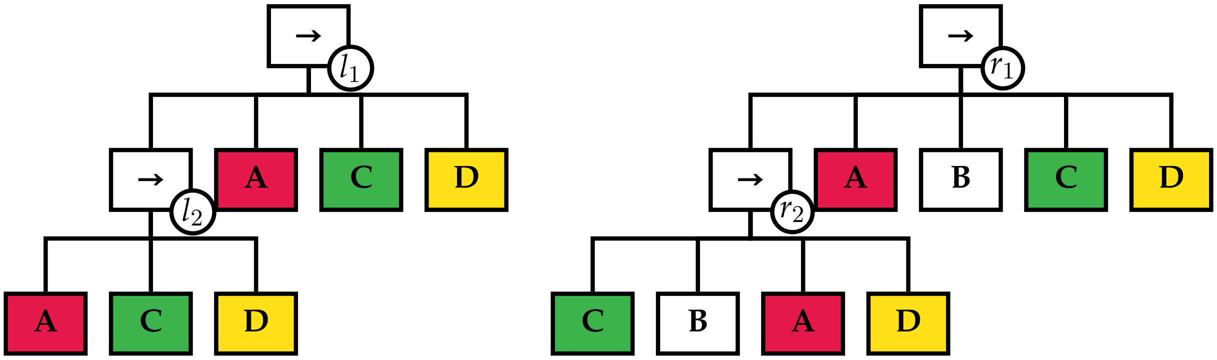
In order to improve the algorithm's detection of such cases, we modify our abstract syntax trees by "squashing" sequences of binary operators together into n-ary nodes. We show how this affects the same two trees in Figure 3.31. Thanks to the squashing, the four internal nodes l_1, l_2, r_1, and r_2 now display different similarities with each other. The algorithm will now consider that l_2 and r_2 are most likely to be similar, and mark them as such.

Once this is done, it will recompute similarities for the remaining, unmatched internal nodes. Here, l_1 and r_1, the only remaining nodes, will now have an increased



$S(l_1, r) = 1$ $\qquad$ $S(l_2, r) = 1$

Figure 3.30. Example of sub-trees with equal similarity



$$S(l_1, r_1) = \frac{6}{9} = 0.67 \qquad S(l_1, r_2) = \frac{6}{8} = 0.75$$

$$S(l_2, r_1) = \frac{6}{8} = 0.75 \qquad S(l_2, r_2) = \frac{6}{7} = 0.86$$

Figure 3.31. Example of squashed telescopes

similarity of $\frac{8}{9}$, or 0.89, due to l_2 and r_2 being considered similar. This should exceed our custom threshold and allow these nodes to be considered similar, resulting in the matching shown in Figure 3.32.

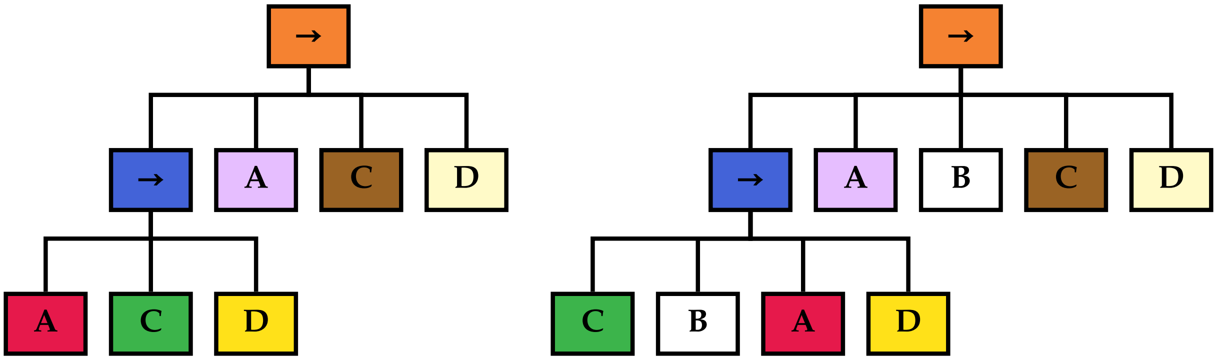


Figure 3.32. Guess for matching with squashed telescopes

However, we now need to unsquash the tree and figure out the proper relationship between the previously squashed nodes. This yields the syntax trees as shown in Figure 3.33, where several nodes share the same coloring. In order to produce a diff, we would like to have unique pairings between the two trees.

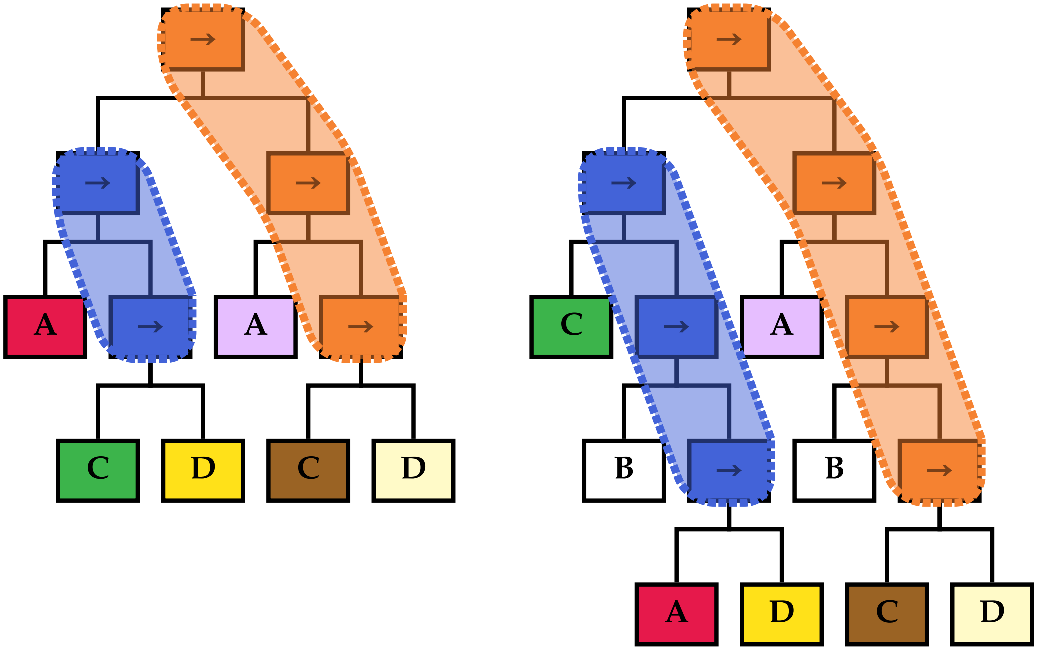


Figure 3.33. Guess for matching unsquashed, unresolved

Since → is a right-associative operation, we resolve the pairings by comparing the left children of the remaining nodes first, then by comparing similarity on the remaining nodes. This yields our final guess, as shown on Figure 3.34. The outlines highlight those nodes we considered similar due to their left child. The root nodes have different left children, but their similarity is $\frac{1}{2}$, or 0.5, due to having similar right children, so they are also considered similar.

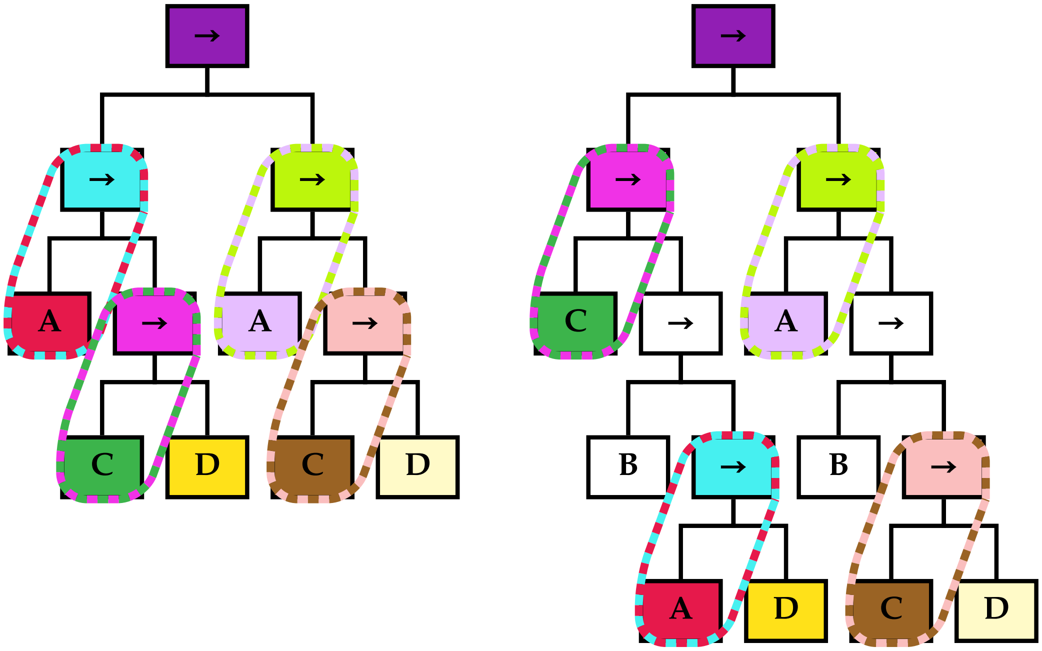


Figure 3.34. Guess for matching unsquashed, resolved

Now that we have identified similar nodes between the old tree and the new tree, the final step is to build an actual diff from this information. The algorithm is a fairly straightforward tree recursion. We highlight how it works on three relevant examples.

Example 1

When the nodes on either side are equal, as is the case with the root nodes of the two trees we've been considering so far, we can **keep** the node (here, a Π node, using $\overset{\text{Mod}}{\Pi}$), and recursively compute the diffs between its sub-trees. This is shown in Figure 3.35.

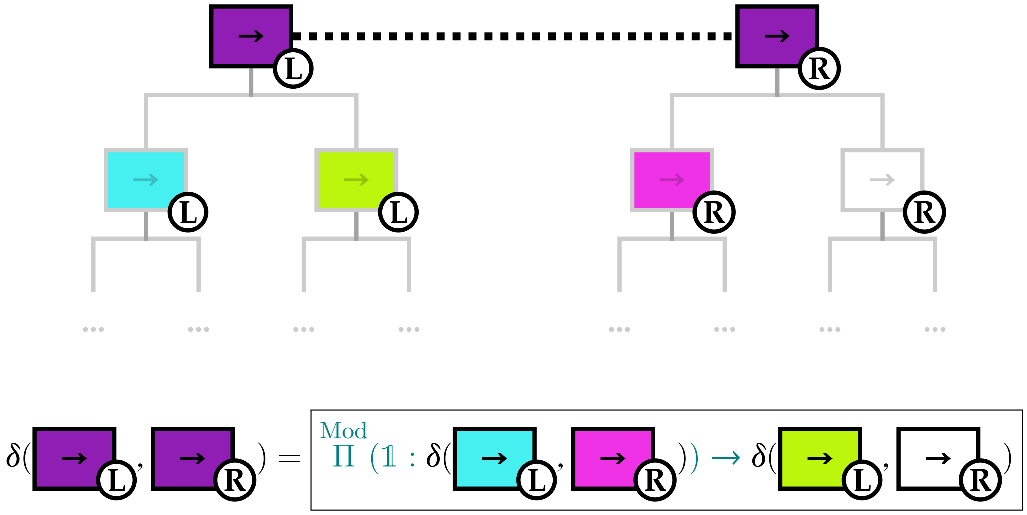


Figure 3.35. Run of the algorithm for turning a matching into a diff (example 1)

Example 2

When the nodes on either side are **not** equal, as is the case with the left sub-trees of the trees we previously considered, we must figure out whether the node on the left has been removed, or moved around, and if the node on the right has been added, or moved around. This is shown in Figure 3.36. In this case, the left node has a similar node in the right sub-tree, **and** the right node has a similar node in the left sub-tree. In order to resolve this crossing of nodes, we must introduce a **permutation**. Our algorithm figures out a minimal permutation that will re-order all the nodes in the left telescope, such that all such crossings are resolved at once. In this case, the permutation is simply switching the first and second elements of the telescope, indicated by the permutation $\overset{[1,0]}{\overset{\rightleftarrows}{\Pi}}$. We then proceed recursively with the permuted tree on the left, and the exact same tree on the right.

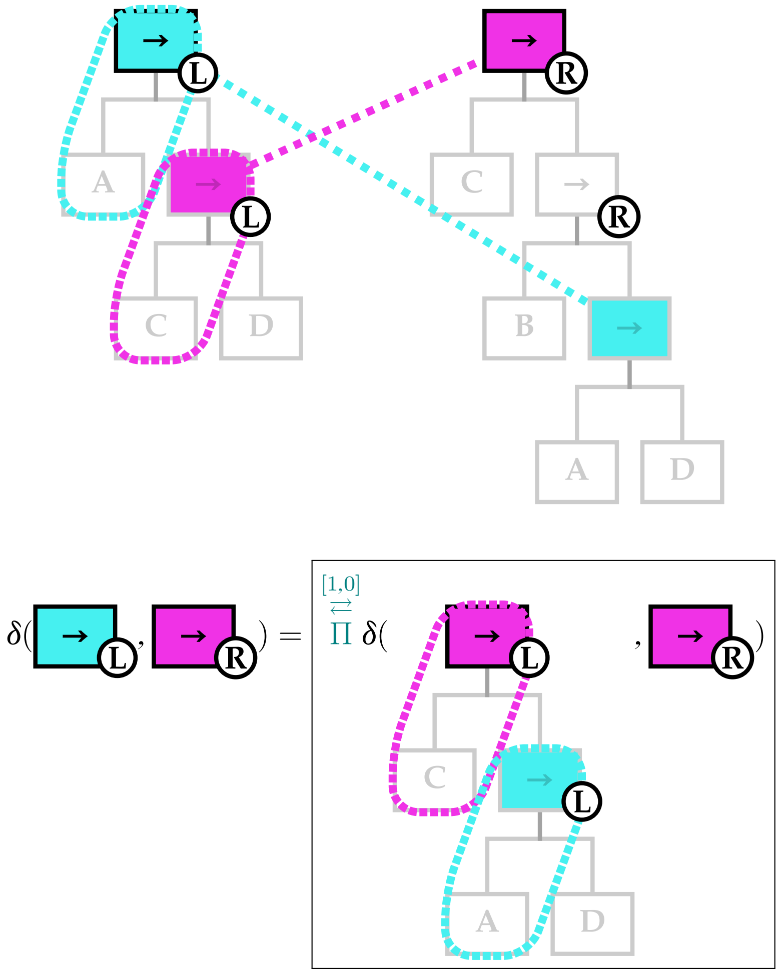


Figure 3.36. Run of the algorithm for turning a matching into a diff (example 2)

Example 3

Another case where the nodes considered do not match each other is shown in Figure 3.37. There, the right sub-tree contains a matching node for the left node, but the left sub-tree does **not** contain a matching node for the right node. This tells us that the right node must have been **inserted**. We output an insertion with $\overset{\text{Ins}}{\Pi}$, and recursively process the entire left tree against the right sub-tree of the right node.

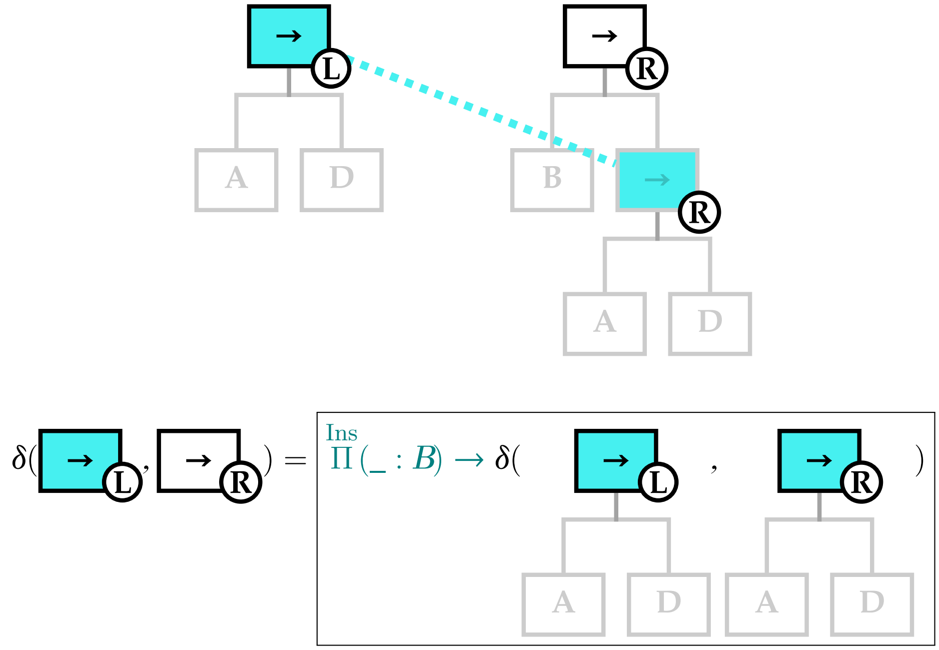


Figure 3.37. Run of the algorithm for turning a matching into a diff (example 3)

Figure 3.38 shows the final result of running our algorithm on our running example trees. The result is framed with colors indicating the provenance of each part of the large expression. The two-colored frame indicates the color of the two nodes that have given rise to a permutation. Notice how the result is neither computed via a top-down nor a bottom-up approach: the top node gives rise to the outermost $\overset{\text{Mod}}{\Pi}$, but its left child gives rise to one of the innermost $\mathbb{1}$.

This chapter is, in part, currently being prepared for submission for publication of the material. The dissertation author was the primary investigator and author of this

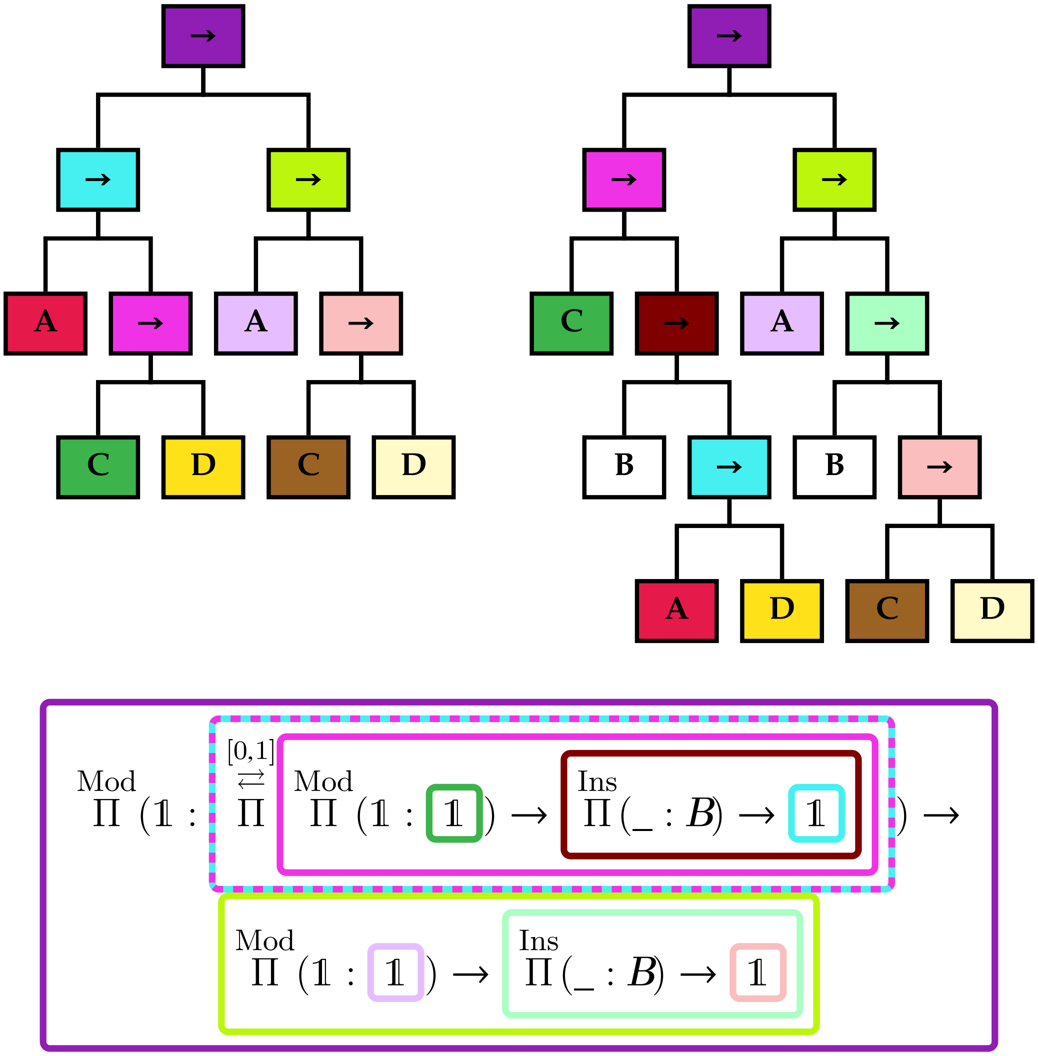


Figure 3.38. Final guess

material. The only co-author is their advisor, Sorin Lerner.

Chapter 4

Coop: extending *Chick* to repair other languages

In this chapter, we present ongoing work to extend *Chick* so that it can repair programs regardless of their surface level language. We call this extension *Coop*.

Section 4.1 presents a high-level view of the architecture of *Coop*.

Section 4.2 explains how we embed constructs from foreign languages within *Chick*, and how we extract those back after repair.

Section 4.3 describes our use of traversals to repair inner constructs that we handle within outer constructs that we do not handle.

Section 4.4 shows how we can turn the output abstract syntax tree of *Coop* into concrete syntax with minimal impact on the original layout of the code.

4.1 Design of *Coop*

Unfortunately, testing *Chick* on real-world *Coq* programs is extremely complicated. For one, the language described by *Chick* is a very small subset of Coq's *Gallina* and *Vernacular* languages, ignoring not only many of the core constructs of those languages, but also all of *Ltac*. Even ignoring this lack of support, parsing *Coq* code correctly is almost always not possible outside of *Coq* itself, because the language supports an extensible syntax, adding almost arbitrary notations to its core syntax, and does not (yet) expose this information in a convenient way to external tools.

Even assuming the problem of parsing solved, *Chick* as described in Section 3.3 has no way of representing all the features of *Coq* that we do not explicitly support. In order to build a robust tool, that can withstand future evolution of those languages, we need a way to account for language constructs that we do not know. What we would like is the ability to gloss over syntactic constructs that we do not know, and provide repairs for the rest of the program, to the best of our ability. This is unsafe by nature, since the syntactic constructs we gloss over can change the global environment in ways we do not account for, but any help attempt is better than none, since our tool only claims to cut some of the tedious work for its users.

We present the workflow that we envision for *Coop*, an extension of *Chick* to support repairing other programming languages, in Figure 4.1. Starting with an original program and a partially refactored program, in some source language, we can use a parser for the source language to obtain abstract syntax trees (*AST*) for those. Now, in order to embed these into *Chick*, we need some program we call an *embedder*, and discuss in Section 4.2. Now we can run the original *Chick* pipeline, described in Section 3.2, until we obtained a repaired abstract syntax tree, still in the *Chick* language. We can now run an *extracter*, which performs the inverse operation to our embedder. We obtain a repaired abstract syntax tree, now in the syntax of the source language. We can then obtained the repaired program, using the surface syntax of the source language, by calling a pretty-printer. We use a diff-pretty-printer, described in Section 4.4, so as to obtain a program syntactically as close as possible to the original.

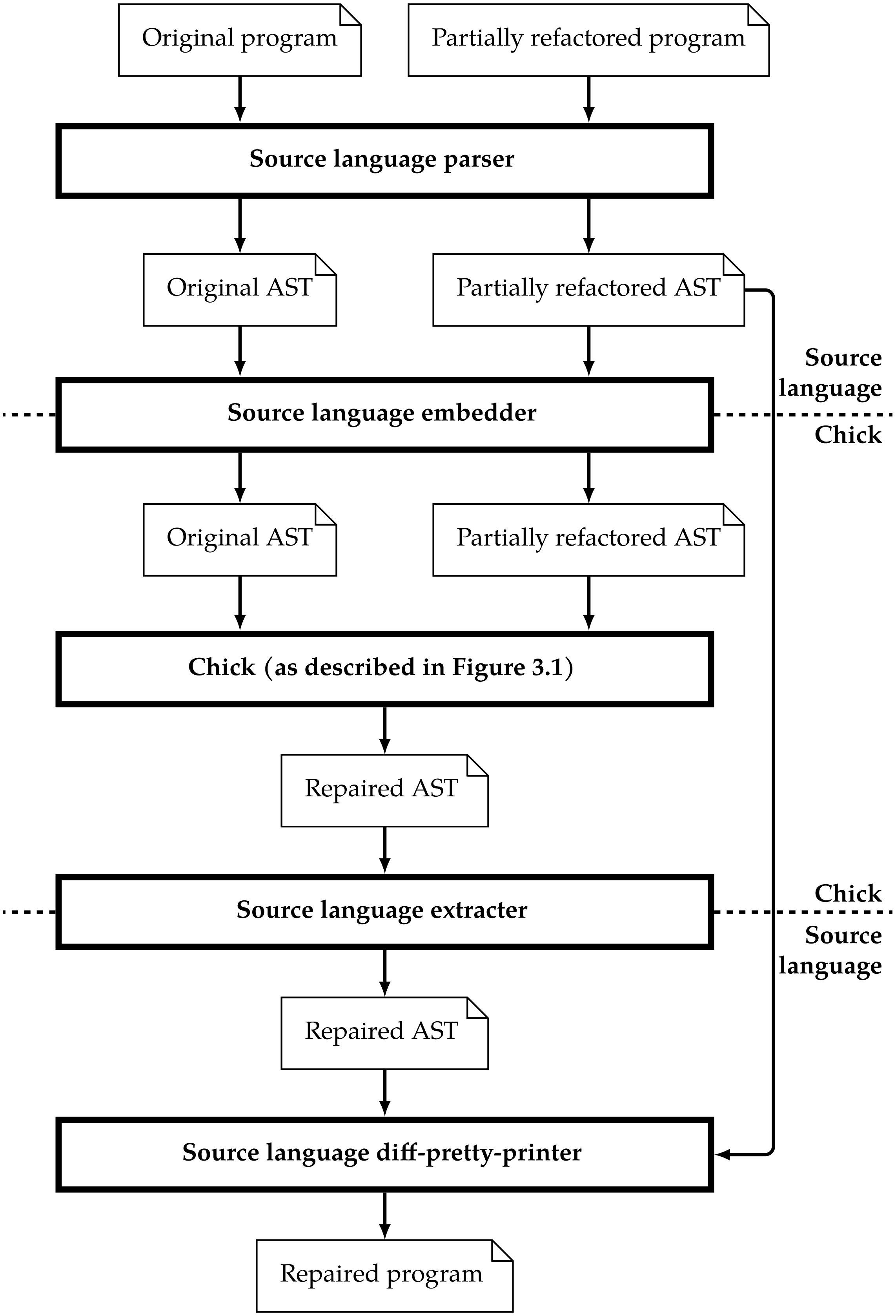


Figure 4.1. *Coop*'s workflow

4.2 Embedding and extracting programs

In order to embed a new language into *Chick*, we simply need to add a constructor to the constructor of our `Vernacular` and `Term` types. For instance, we added support for *OCaml* with the following extension:

⟨*vernacular*⟩ ::=
| … (same as in Section 3.3.2)
| `OCamlStructureItem` ⟨*ocaml-structure-item*⟩
⟨*term*⟩ ::=
| … (same as in Section 3.3.1)
| `OCamlExpression` ⟨*ocaml-expression*⟩

Now, our *embedder* can inspect the incoming abstract syntax tree, and decide to map constructs from the source language to *Chick* constructs, when it makes sense to do so, or store the constructs in the extra constructors when there is no matching concept in *Chick*.

For instance, in the *OCaml* embedder, we map constructs such as *OCaml*'s (non-polymorphic) variants to our notion of inductive data types, since the concepts match. We can also map most simple expressions to our term type, but complex features that we do not support (say, nested pattern matching) get put aside in the `OCamlExpression` constructor.

The *extracter* acts as an inverse of the embedder, mapping *Chick* constructs back to their source language counterpart. Constructs that have been stashed in one of our constructors for unsupported language features are simply restored as is, and *Chick* constructs are mapped back to the source language construct they came from.

Unfortunately, we sometimes need to map different source language abstract syntax trees to the same *Chick* abstract syntax tree. For instance, using *OCaml* as an example again, the following two functions:

```
let f x y = 42
let g x = fun y -> 42
let h = fun x -> fun y -> 42
```

map to similar *Chick* terms:

```
Definition f : _ := λ x, λ y, 42.
Definition g : _ := λ x, λ y, 42.
Definition h : _ := λ x, λ y, 42.
```

In order to try and preserve surface-level syntax as much as possible, we would like to attach metadata about such source language syntactic choices in *Chick* terms. This goals seems at odds with our intent of making *Coop* extensible, but we can actually achieve this somewhat easily by using the so-called "Trees that grow" technique employed by Najd and Jones [28] for the Glasgow *Haskell* Compiler (GHC). Using generalized algebraic data types (GADTs), and type families, this technique allows us to build abstract syntax trees for *Chick* that can be decorated with arbitrary metadata, and use type-level computations to determine what this metadata may be for different constructors in different contexts.

For instance, we can solve the previous problem by attaching metadata about the number of function arguments that are marked as parameters, as opposed to being abstracted over.

```
-- We declare the existence of a family of types for storing the
-- metadata attached to a Definition.
type family DefinitionMetadata ξ

-- We attach this metadata to the Definition contructor.
data Vernacular ξ
  = Definition (DefinitionMetadata ξ) (DefinitionData ξ)
  | …

-- We instantiate the family for the OCaml language with an
-- integer representing the number of explicit parameters.
type instance (DefinitionMetadata OCaml) = Int
```

With this information attached to our `Definition` nodes, the extracter can pick the appropriate form, so as to preserve the original syntactic choice.

```
extracterVernacularOCaml ∷ Vernacular 'OCaml → StructureItem
extracterVernacularOCaml (Definition metadata data) =
  -- Here, metadata ∷ Int, tells us how many parameters should
  -- be before the equal sign, vs. bound by a lambda
```

Sometimes, *Chick* might need to come up with a new datum, for which there is no sensible metadata it could attach to it. For such cases, using an optional type for the metadata (say, `Maybe Int` instead of `Int` here) allows us to avoid this issue.

Overall, this technique gives us great flexibility, as different languages will need different metadata for the same concepts. The language of choice is carried throughout the *Chick* pipeline as a type-level tag, using *Haskell*'s `DataKinds` extension. This also means that, anywhere in the pipeline, we can readily choose to inspect the tag and do something specific based on it. For now, we only use this technique to pick the appropriate parser and pretty-printer, but it could also be used to change parts of the repair algorithm if some language needs specific, distinct support.

4.3 Optics to repair unknown language constructs

There is a major problem with the approach described in Section 4.2. Consider the following *OCaml* code:

```
type a = … (* some type definition *)

module SomeModule =
  struct
    type b = … (* some other type definition, depends on a *)
    let  f = … (* some function, depends on a *)
  end
```

Our current implementation of *Chick* does not have a notion of modules. Therefore, *Coop* has to stash away the whole module declaration in its `OCamlStructureItem`, and will provide no repair for it. This is unfortunate, because the module contains data type declarations, and function declarations, all of which we know how to repair, and could attempt to, were they visible.

We can find a solution to this problem using functional lenses. *Lenses* are an abstraction mechanism encompassing a pair of a getter and a setter for some value within some datum. They are often described in the simplified version:

```
data Lens s a = Lens
  { get ∷ s → a
  , set ∷ s → a → s
  }
```

where `s` stands for the type of the store, or the outer structure, and `a` stands for the type of the value under focus.[1]

Lenses are useful because they are first-class, composable values that allow modifying parts of a datum while ignoring the rest of it. This matches quite well with our goal: we wanted to zoom in on the inner parts of the module, getting and setting the components of the module we understand, ignoring the rest. However, lenses only have one focus, whereas our data could contain an arbitrary amount of components we might want to focus upon.

Traversals are a generalization of lenses with multiple foci. They are also an ab-

[1] Actual implementations of lenses are usually more complex, with two additional parameters allowing the setter to output at a different store type. Some instances also add a `Functor` constraint to allow getting and setting in a functorial context.

straction of a getter and a setter, except that they may operate on multiple instances of the focus type within a datum. In order to allow maximum flexibility, traversals process the foci within an applicative functor. Uniform traversals over parameterized data structures can be automatically derived using extensions like *Haskell*'s TemplateHaskell, but in our case, we will need to write our traversals manually. This allows us to specifically choose the order of the traversal, as well as selectively choose whether some parts of data types should be traversed or not. Let us see how we can define such a traversal over *OCaml* abstract syntax trees. For this, we will use type classes to capture data types that contain structures we are interested in. For the *OCaml* language, we are interested in values of type Structure (which correspond to our Vernacular data type), and values of type CoreType and Expression (which correspond to our Term data type).

```
class HasStructure t where
  structure ∷ Traversal' t Structure

class HasCoreType t where
  coreType ∷ Traversal' t CoreType

class HasExpression t where
  expression ∷ Traversal' t Expression
```

We can then instantiate those classes for all of *OCaml*'s abstract syntax tree constructs, for instance:

```
instance HasStructure ModuleExprDesc where
  structure f = case f of
    -- if something does not contain a Structure,
    -- we can skip it entirely:
    PmodIdent     i → PmodIdent <$> pure i
    -- if something contains a Structure right here,
    -- we can apply `f`
    PmodStructure s → PmodStructure <$> f s
    -- if something may contain Structures recursively,
    -- we keep traversing it
    PmodApply m1 m2 → PmodApply
                      <$> traverseOf structure f m1
```

```
                           <*> traverseOf structure f m2
      …
```

Writing those instances is entirely systematic, so we believe it can be automated using *Haskell*'s TemplateHaskell facilities, though we have not yet written such automation.

4.3.1 Traversals over concrete syntax and functorial syntax

Depending on how the abstract syntax tree of the source language has been defined, there are actually two distinct ways of building those traversals. They correspond to the following two ways of defining a data type of "commands" over a datatype of "expressions":

```
-- Variant 1: concrete representation
data Language
  = DoSomething Language.Term
  | …

-- Variant 2: functorial representation
data LanguageF e
  = DoSomething e
  | …

type Language = LanguageF Language.Term
```

With the *concrete representation*, the constructor DoSomething may only ever contain values of type Language.Term. It is more rigid than the *functorial representation*, wherein the type of expressions is abstracted over. In the functorial setting, one can instantiate the LanguageF functor with different types, yielding a family of concrete types.

With the functorial approach, *Coop* can replace the source language's expression type with its own type, essentially storing *Chick* abstract syntax trees within the source language’s abstract syntax tree:

```
data Vernacular f
  = …
  | ForeignLanguage (LanguageF f)
```

In this case, *Chick* can, given a traversal for `LanguageF f`, transform all occurrences of `f` into `Chick.Term`, and repair those using the applicative functor. While it is convenient to store our expressions within the original abstract syntax tree, we are forced to replace all occurrences so that the types match up. Once all of this is done, we have the following endpoints:

```
embed ∷ Vernacular Language.Term → Vernacular Chick.Term
embed = …

chick ∷ Vernacular Chick.Term → Vernacular Chick.Term
chick = …

extract ∷ Vernacular Chick.Term → Vernacular Language.Term
extract = …
```

and we can simply pipeline them together to obtain the resulting abstract syntax tree, that must finally be processed by our pretty-printer (as described in Section 4.4).

In the case of the concrete representation, because we cannot store *Chick* terms within the foreign language’s abstract syntax tree, we must proceed with a level of indirection: we can use an *indexed* traversal to extract a list of all the values being traversed, turn those values into *Chick* terms, repair those *Chick* terms, turn the repaired terms back into source language terms, and finally reinsert those repaired terms in the proper locations using the indexed traversal. The process looks like the following partial code:

```
itraverseLanguage ∷
  IndexedTraversal' Int Language.AST Language.Term
itraverseLanguage = …

```

```
getTerms ∷ Language.AST → [Language.Term]
getTerms = toListOf itraverseLanguage

-- From there, we do the following:
-- 1. map `embed`  to get a  [Chick.Term]    (unrepaired)
-- 2. map `repair` to get a  [Chick.Term]    (repaired)
-- 3. map `extract` to get a [Language.Term] (repaired)

insertRepairedTerms ∷
  [Language.Term] → Language.AST → Language.AST
insertRepairedTerms l =
  imapOf itraverseLanguage (λ index _ → l ‼ index)
```

The indexed traversal allows us to know where to insert the repaired terms in the original abstract syntax tree.

4.3.2 Scope-aware traversals

There remains yet another trouble with our traversals. Consider the following *OCaml* code:

```
module SomeModule (A : ModuleTypeForA) =
  struct
    (* ... *)
  end
```

Thanks to our traversals, we might be able to repair terms that appear within this *OCaml* functor[2]. Unfortunately, while glossing over the `module` construct, we also gloss over a binding construct, introducing a binder `A`. This might create problems when this module appears in an ambient context where some other variable `A` is bound. In particular, if our repair algorithm is trying to repair instances of the variable `A`, it will, erroneously, attempt to repair the local references to `A`, mistaking them for references the outer `A`.

[2] Note that *OCaml* uses the term *functor* to describe about parameterized *modules*, while, so far, this dissertation has been using the term functor to describe functorial parameterized *data types*.

In order to avoid this issue, we can change our traversals so that they accumulate bound variables as they traverse the source language abstract syntax tree, and turn our focal points into a pair of the construct under focus, and the list of binders it lives under. With this information, we can easily alter the term so that *Chick* knows that those variables have been rebound. This can be achieved in two ways:

- either by over-populating, before running the repair algorithm, its local context with those binders (bound to an unknown type, represented as a hole), and populating the local context diff with as many $\mathbb{1} \overset{\text{Mod}}{::} \ldots$,

- or, by adding extraneous λ-abstractions to the term, and adding as many extraneous $\overset{\text{Mod}}{\lambda}\ \mathbb{1} \to \ldots$ to its diff, and running the repair algorithm in the current global environment and local context. This requires a post-processing pass to remove those extraneous lambdas from the repaired term, before injecting it back into its abstract syntax tree.

4.4 Diff-aware pretty-printing

So far, we have explained how we can embed the constructs and binding structure from the abstract syntax tree of some arbitrary source language, repair those constructs as part of our core language, and then extract the repaired constructs back into an abstract syntax tree for the original language. However, we need to repair the concrete syntax of the original program, not just the abstract syntax tree.

While we could just call a naive pretty-printer with the final abstract syntax tree, there is a high chance that doing so would interfere with the layout of the original code, unless we somehow stored enough information to reproduce the exact original layout. In order to minimize our impact on the concrete syntax of the program, we can instead try to only modify those parts of the concrete syntax that need being altered.

To do so, we need the abstract syntax tree of the source language to contain location information about the provenance of its nodes. This requirement is not hard to satisfy: we expect most languages to have parsers that produce such information, if only to be able to indicate error locations. Now, when we obtain the repaired source abstract syntax tree, we can simply follow a simultaneous recursive descent through the original and repaired trees. At any point where they differ, we can:

- obtain the source span for the text to be replaced from the original tree,
- obtain the replacement text by pretty-printing the node from the repaired tree.

Then, we can graft the pretty-printed repaired term in place of the original text span. If the language is indentation-sensitive, we might need to be more clever, figuring out the level of indentation, and hanging the replacement text at that depth.

We can also come up with extra quality of life improvements, such as retaining and lifting the comments from the deleted span, so that no comment is lost in the repair. Repairs can also be processed sequentially, with a human in the loop, so that repair suggestions are reviewed by the user before being applied, for instance, by using a merging tool such as the ones provided for version control systems.

This chapter is, in part, currently being prepared for submission for publication of the material. The dissertation author was the primary investigator and author of this material. The only co-author is their advisor, Sorin Lerner.

Chapter 5

Related work

Refactoring functional programs

A main line of work in refactoring functional programs is the work done on the HaRe refactoring tool for *Haskell* [23] and subsequently on refactoring *Erlang* programs [24]. The scope of the former is much larger than this paper: they implement a large library of refactoring algorithms, including some that this paper covers, for the *Haskell* programming language. In particular, their tool handles *Haskell* 98, and provides an API to perform refactorings. They also refactor while preserving the user's syntactic choices, which proves challenging in layout-based languages like *Haskell*. However, their publications do not distill the details of how to implement those refactorings, and thus, does not allow for easy experimentation, porting to other languages, or verification of their techniques.

A very promising ongoing line of work is that of *ornaments* [35]. Their formalism allows to describe data type transformations from within the language itself, and perform similar transformations to ours. The ornamentation language is more expressive than our language of diffs in many ways, allowing arbitrary mappings between constructors before and after modification: for instance, one of their motivating example cuts a constructor into two different constructors depending on some Boolean condition. As a result, their technique requires the programmer to describe the transformation they wish to carry as an ornament, and the programmer's help is required alongside the transformation to indicate to the repair algorithm what to do in corner cases. Our restriction to a simple language of modifications allows us to guess the transformation based on the syntactic modification of the program itself, and grants us more leverage on automation, at the price of a much lower expressiveness.

Refactoring proofs

One of the few tools that attempts refactoring in the context of the proof assistants is the Levity tool by Ruegenberg [32]. The refactoring it performs is referred to as *levitation*, and corresponds to the move of a lemma up its dependency chain, to a location that is deemed more relevant. This transformation is hard to perform in many proof assistants, because figuring out where the lemma can exist can be challenging and time-consuming, and because the action of levitating the lemma might derail other parts of the proof development, even when they seem unrelated, because the declaration of a lemma might have unintended side-effects through automation mechanisms.

Another main line of work in this domain is the line of work of Whiteside et al. [34], followed up by Dietrich et al. [9]. They define a framework for specifying refactoring rules over an idealized proof language called Hiproof, and its idealized tactic language called Hitac. The latter is designed to be close to the declarative Isar tactic language of Isabelle. This framework allows them to define many interesting refactorings, and to prove their correctness, informally, with low effort. They acknowledge that it would be more difficult to develop the same refactorings for a less declarative tactic language like the *Ltac* language of the *Coq* proof assistant.

Boite [3] also addresses the problem of modifying an existing inductive type definition while reusing existing proofs. Their technique does not alter existing code, rather, they internalize the notion of *extension* of an inductive data type (that is, the adding of a constructor, or parameters), and provide commands that perform such extensions, deriving extended data types and extended functions from the original ones. It would be interesting to have proof assistants internalizing and exposing this kind of algebraic manipulations of data types.

Finally, recent work by Ringer et al. [31] builds semantic patches to help programmer refactor their proofs. Their approach does not modify the old program into a new program. Rather, the old program is kept with its proofs, alongside the new program, and patches are automatically derived to transport proof obligations between the old program and the new program. Their approach seems complementary to ours, depending on the use case for the programmer.

Automatic code generation

Another exciting line of work consists in removing the friction of performing those manual refactorings altogether.

General purpose automation like Isabelle's sledgehammer Blanchette et al. [2] or the recent for on a similar hammer for Coq Czajka and Kaliszyk [8] can help reduce the work of programmers in those systems by avoiding writing terms entirely. Hopefully, doing so removes a lot of brittle tactic calls, but they do not help in refactoring data type definitions when the objects of discourse evolve.

The body of work on program synthesis could also prove very beneficial. For instance, [30] can perform program synthesis from refinement types, which could potentially spare the programmer from fixing code that is never manually-written, but rather synthesized. However, the specifications themselves could still require refactoring tools, since they are themselves terms from a refinement type system.

IDE-based detection of refactoring attempts

The work of Foster et al. [12] focuses on detecting when a programmer is trying to perform a refactoring in Java programs. Many times, the programmer will not know that refactoring tools exist in their IDE, or will not realize that they are performing a task that corresponds to an existing refactoring algorithm. In particular, they use a text-based diff in order to be able to recognize refactoring attempts even when the program is in a transient, non-parseable state.

Incremental computation

The literature on incremental computation focuses on the problem of reliably and efficiently re-computing data that is the output of some computation after the input undergoes a modification, which is similar to our deriving of repairing functions from Section 3.7. For instance, Acar et al. [1], Carlsson [5], as well as Firsov and Jeltsch [10], all use an abstract data type for values that can be efficiently recomputed, and propose a monadic interface to build incremental computations while providing the programmer with an interface that hides those details behind the scenes. Similarly, Chen et al. [6] automatically transform a non-incremental program to inject incremental behavior into it. While this is similar to what we do in spirit, we care about more than just obtaining the result of the incremental computation, since we sometimes reuse the diff of one data type to compute a diff for another, related data type. Incremental computation techniques are typically not interested in this, and do not usually expose the internals of their functioning to the user.

Cai et al. [4] define a general way of using *derivatives* of functions to compute their reactions to some abstract modifications, and compute those derivatives for first-order functions. They demonstrate it on examples using unordered collections. Our diffs are an instance of their general notion, but we depart from it by having more specialized diffs to fit our needs, and by supporting a small family of higher-order functions (namely, folds over lists), using our deriving technique to help compute their derivative.

Chapter 6

Future work

In this chapter, we will go over several potential ways to extend *Chick* and *Coop* in the future.

6.1 Avoiding problematic binders using scope sets

Our treatment of binders in Section 3.6.4 is extremely cumbersome. Not only do we have to be careful about capture-avoiding substitutions, both in the old program and the new program, but we also have to be careful to repair every term only once, because the repair operation is *not* idempotent.

A novel approach from Flatt [11] developed for the new Racket macro expander might be useful here. In their setting, every scope created is assigned a label, and every reference to a bound variable is initially marked with the set of scopes within which it appears in the original program text. Subsequently, the macro expander discards the renaming approach entirely. Instead, every binding form, as well as every macro expansion, creates a scope, adds that scope to all identifiers in binding position, and keeps track of a mapping from identifiers with scope sets to the representation of a binding.

Subsequently, the macro expander is free to perform expansions without need for capture-avoiding freshness conditions, or, as it is called in the macro expansion literature, *hygienic macro expansion* Kohlbecker et al. [20]. Instead, a reference's binding is found by looking in the mapping for the binder with same name whose scope of sets is the largest subset of the reference.

We believe this approach could potentially be adapted to solve the problems we encounter. Our first issue (demonstrated in Rule Repair-Term-1-Mod-II) was that of an outer binding becoming shadowed by a binding position, when said outer binding is renamed by the programmer to have the same name as the binding position. For instance, in the following code:

`Definition f := …`	`Definition g := …`
`… (λ g → f g) …`	`… (λ g → f g) …`

we could not simply rename the variable `f` on the right side to `g`, because it would accidentally become captured by the term abstraction. We needed to be very careful about this problem in our rules. With a "set of scopes" approach, we would instead have:

Definition $\texttt{f}^{\{a\}}$:= …	Definition $\texttt{g}^{\{a,m\}}$:= …
… (λ $\texttt{g}^{\{a,b\}}$ → $\texttt{f}^{\{a,b\}}$ $\texttt{g}^{\{a,b\}}$) …	… (λ $\texttt{g}^{\{a,b\}}$ → $\texttt{f}^{\{a,b\}}$ $\texttt{g}^{\{a,b\}}$) …

where scope label a corresponds to the scope created by the definition of `f`, scope label b corresponds to the scope created by the term abstraction introducing `g`, and scope label m corresponds to a virtual scope that we introduce to mark where the user has modified their program. As a result, in this approach, we can repair the program on the right with simple substitution, yielding:

Definition $\mathtt{f}^{\{a\}}$:= … … (λ $\mathtt{g}^{\{a,b\}}$ → $\mathtt{f}^{\{a,b\}}$ $\mathtt{g}^{\{a,b\}}$) …	Definition $\mathtt{g}^{\{a,m\}}$:= … … (λ $\mathtt{g}^{\{a,b\}}$ → $\mathtt{g}^{\{a,m\}}$ $\mathtt{g}^{\{a,b\}}$) …

where the inner mention of `g` with scope set $\{a, m\}$ is *not* captured by the term abstraction, because its set of scopes mentions scope label m, preventing it from being a subset of the enclosing $\{a, b\}$ set. We will still need to resolve the name collision when producing the concrete repaired program, but as far as the repair algorithm is concerned, there is no need to pick a fresh name for the binder, as it can *not* capture the repaired variable.

The second problem we had with binders, still in Rule RModPi, was to pick a name that was both fresh for the term abstraction, and fresh for the type abstraction, so that we could add this name to the typing context, as we proceeded to repair the term abstraction's body, assuming its type was the type abstraction's body. Here again, we could circumvent the renaming altogether by introducing a new scope label.

If we decide to attach scope labels to any term of the abstract syntax tree, rather than only variables, we could even consider using scope labels as markers, separating terms that have been repaired from terms that have not been repaired yet.

6.2 Repairing tactic scripts

Using the same methodology we used to repair programs and terms, we have considered repairing proof scripts written in a tactic language like *Ltac*. The problem with *Coq* tactics is that their effect depends on what is in context when the tactic is called, making them hard to reason about statically. For instance, the `intros` tactic inspects the current sub-goal, and introduces as many universally quantified variables from the sub-goal as it can. To do so, it has to come up with names for those variables, yet the user never explicitly passes those names. Instead, the algorithm chooses suitable, fresh names, using the names of the variables as they appear in the sub-goal (when their quantification is explicitly named), or picking some fresh name.

For instance, if your current goal is:

```
nat → nat → nat → nat
```

and you run `intros`, the following names will be added to your context:

```
H, H0, H1 : nat
```

This makes tactic scripts complicated to repair, as there is no lexical binding position for those automatically introduced names. In order to repair tactic scripts, we *must* be able to replay them. This can only be done outside the tool if the tactic language has well-defined semantics. Unfortunately, *Ltac* is a very complicated tactic script, and formalizing its semantics is still an open-ended problem.

Had we done so, we could hope to repair tactics by replaying them, and computing diffs for the proof context, just like we did for our global environment and local context when repairing terms. The problem is very interesting, as a change in the number of constructors of a data type can have drastic changes in the structure of a proof. We have started working on a prototype, toying with repairing a tactic language that only provides tactics like `intro` and `destruct`, but we don't have a proper prototype for those yet.

6.3 Integrating code diffs in version control systems

An equally ambitious goal could be to treat our diffs, that is, first-class values that represent the intent of changes made to a program, the same way we treat the code of the program. For instance, as a program evolves, we keep a published index on the changes made to it using a version control system. We can envision a diff-aware version system wherein diffs are also published.

Consider a use case where Ada is publishing a software library that Bertrand depends upon for his project. When Ada decides to make a breaking change to her library, she could perform the change by using a diff-based repair algorithm like the one we present. When the repair is finished, she has a complete, structural diff, that captures exactly all of the changes that have happened to the library. Ada can publish this diff, alongside the new code, to her public version control system. Bertrand can update his dependency to Ada's new version, but he needs to update his client code to adapt to the changes made by Ada. Thanks to the published diffs, Bertrand could also run a repair algorithm over his code, automating much of the necessary code changes needed to stay compatible with Ada's library.

Of course, this workflow would need to be extremely polished to convince both users to opt in. In particular, we would definitely want a strong diff guessing algorithm, like the one presented in Section 3.8, and would need a nice way of presenting different guesses to the programmer so that they can safely pick the choice that best matches their intent.

Appendix A

PeaCoq

A.1 *PeaCoq* A-B study material

In this section, we report the entire listing of exercises provided to the participants of the A-B study described in Section 2.3.2.

```
(* The following material is derived from Software Foundations by Benjamin
Pierce et al. Their work is under the following MIT license: *)

(*
Copyright (c) 2012

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

```
 *)

(* ###################################################################### *)
(** ** Days of the Week *)

Inductive day : Type :=
| monday : day
| tuesday : day
| wednesday : day
| thursday : day
| friday : day
| saturday : day
| sunday : day
.

Definition tomorrow (d : day) : day :=
  match d with
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => saturday
  | saturday  => sunday
  | sunday    => monday
  end.

Theorem test_tomorrow:
  tomorrow saturday = sunday.
Proof.
  simpl. reflexivity.
Qed.

(* ###################################################### *)
(** ** Lists of Numbers *)

Inductive natlist : Type :=
| nil  : natlist
| cons : nat -> natlist -> natlist
.

Definition empty_list := nil.

Definition singleton_list := cons 42 nil.

Definition one_two_three := cons 1 (cons 2 (cons 3 nil)).

Fixpoint concat (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil      => l2
  | cons h t => cons h (concat t l2)
  end.

Theorem test_concat1 :
  concat (cons 1 (cons 2 nil))
         (cons 3 (cons 4 nil))
  = (cons 1 (cons 2 (cons 3 (cons 4 nil)))).
Proof.
  simpl. reflexivity.
Qed.

(* ###################################################### *)
(** * Reasoning About Lists *)

Theorem concat_nil_left : forall (l : natlist),
  concat nil l = l.
Proof.
```

```
  (* FILL IN HERE *)
Qed.

Theorem concat_nil_right : forall (l : natlist),
  concat l nil = l.
Proof.
  (* FILL IN HERE *)
Qed.

(* In-class exercise! *)
Theorem concat_associativity : forall (l2 l1 l3 : natlist),
  concat (concat l1 l2) l3 = concat l1 (concat l2 l3).
Proof.
  (* FILL IN HERE *)
Qed.

(*
  [snoc] adds an element [v] at the end of the list [l]:
    snoc (cons 1 (cons 2 nil)) 3 = cons 1 (cons 2 (cons 3 nil))
*)
Fixpoint snoc (l : natlist) (v : nat) : natlist :=
  match l with
  | nil      => cons v nil
  | cons h t => cons h (snoc t v)
  end.

(*
  [rev] reverses a list:
    rev (cons 1 (cons 2 nil)) = cons 2 (cons 1 nil)
*)
Fixpoint rev (l : natlist) : natlist :=
  match l with
  | nil      => nil
  | cons h t => snoc (rev t) h
  end.

(* ###################################################### *)
(**
  For each theorem:
  - Discuss the statement of the theorem with your partner.
  - Once you understand it, prove the theorem.

  Every time you solve a theorem, switch who uses the keyboard/mouse.
 *)

Theorem rev_snoc : forall x l,
  rev (snoc l x) = cons x (rev l).
Proof.
  (* FILL IN HERE *)
Qed.

Theorem rev_involutive : forall (l : natlist),
  rev (rev l) = l.
Proof.
  (* FILL IN HERE *)
Qed.

Theorem concat_cons_snoc : forall l1 x l2,
  concat l1 (cons x l2) = concat (snoc l1 x) l2.
Proof.
  (* FILL IN HERE *)
Qed.

(* ###################################################### *)

Module LogicExercises.
```

```

(* We now use notations from logic:
  /\   stands for the logical conjunction (AND) of two propositions
  \/   stands for the logical disjunction (OR)  of two propositions

  New tactics: left, right

  When your goal looks like [A \/ B]
  You get to pick which of [A] or [B] you will prove.
  If you believe you can prove [A], use the [left.] tactic.
  If you believe you can prove [B], use the [right.] tactic.

  Here is an example:
*)

Theorem right_example : 0 = 1 \/ 1 = 1.
Proof. right. reflexivity.
Qed.

Theorem go_somewhere : 0 = 1 \/ (2 = 2 \/ 2 = 3).
Proof.
  (* FILL IN HERE *)
Qed.

(*
  New tactic: apply

  If you ever have a goal [G]
  And a hypothesis [H : G] or [H : X -> ... -> G]
  You can use the tactic [apply H.] to solve your goal in the former case,
  or turn your goal into subgoal(s) [X], [...] in the latter case.
*)

Theorem B_is_enough : forall A B : Prop,
  B ->
  A \/ B.
Proof.
  (* FILL IN HERE *)
Qed.

(*
  New tactic: split

  When your goal looks like [A /\ B]
  You need to prove both [A] and [B].
  The tactic [split.] lets you split your goal into these two goals.

  Here is an example:
*)

Theorem two_facts : nil = nil /\ 42 = 42.
Proof. split. reflexivity. reflexivity.
Qed.

Theorem more_facts : 1 = 2 \/ (1 = 1 /\ nil = nil).
Proof.
  (* FILL IN HERE *)
Qed.

Theorem A_and_B : forall (A B : Prop),
  A ->
  B ->
  A /\ B.
Proof.
  (* FILL IN HERE *)
Qed.
```

```

End LogicExercises.

(* ###################################################### *)

Theorem snoc_concat_end : forall (l : natlist) (n : nat),
  snoc l n = concat l (cons n nil).
Proof.
  (* FILL IN HERE *)
Qed.

Theorem rev_distributes_over_concat : forall (l1 l2 : natlist),
  rev (concat l1 l2) = concat (rev l2) (rev l1).
Proof.
  (* FILL IN HERE *)
Qed.

(* ###################################################### *)
(** We now introduce [map], which applies a function [f] to
    every element of a list [l].
*)

Fixpoint map (f : nat -> nat) (l : natlist) :=
  match l with
  | nil => nil
  | cons x xs => cons (f x) (map f xs)
  end.

Theorem map_commutes : forall f g l,
  (forall x, f (g x) = g (f x)) ->
  map f (map g l) = map g (map f l).
Proof.
  (* FILL IN HERE *)
Qed.

(* In this theorem, "fun x =>" introduces an anonymous function which receives
   a parameter [x] and returns the result on the right of the arrow. *)
Theorem map_fusion : forall f g l,
  map f (map g l) = map (fun x => f (g x)) l.
Proof.
  (* FILL IN HERE *)
Qed.

(* ###################################################### *)
(** We now introduce [fold], which processes a list with an
    accumulating function [f], starting from an initial value [b].
*)
Fixpoint fold (f : nat -> natlist -> natlist) (l : natlist) (b : natlist) :=
  match l with
  | nil => b
  | cons x xs => f x (fold f xs b)
  end.

Theorem fold_snoc : forall f l x b,
  fold f (snoc l x) b = fold f l (f x b).
Proof.
  (* FILL IN HERE *)
Qed.

Definition map' f l := fold (fun x fxs => cons (f x) fxs) l nil.

(* We use [Lemma] instead of [Theorem] here to indicate that this theorem may
   help you in proving the next theorem. *)
Axiom map'_unroll : forall f x xs,
  map' f (cons x xs) = cons (f x) (map' f xs).

```

```
Axiom map'_nil : forall f, map' f nil = nil.

Theorem map_map' : forall f l, map f l = map' f l.
Proof.
  (* FILL IN HERE *)
Qed.

Ltac cases H := match type of H with _ \/ _ => destruct H end.

(*
  New tactics: cases, contradiction

  When a hypothesis looks like [H : A \/ B]
  You have to prove the goal for each case, to do so, use the tactic [cases H.]
  You will get two goals as a result, one with a [A] hypothesis, one with a [B] hypothesis.

  Finally, if you ever get a hypothesis like [H : False]
  You have derived a contradiction, and you can indicate this to the system by calling
  the tactic [contradiction.], which will solve your goal.
*)

Fixpoint In n l :=
  match l with
  | nil      => False
  | cons h t => h = n \/ In n t
  end.

Theorem In_cons : forall x h l,
  In x l ->
  In x (cons h l).
Proof.
  (* FILL IN HERE *)
Qed.

(*
  New tactic: simpl in *

  Sometimes, you might want to simplify things in your hypotheses, the same way things
  can be simplified in your conclusion.
*)

Theorem In_concat_left : forall x l1 l2,
  In x l1 ->
  In x (concat l1 l2).
Proof.
  (* FILL IN HERE *)
Qed.
```

A.2 *PeaCoq* A-B study post-study survey

We report the qualitative feedback obtained via a questionnaire given after the A-B study described in Section 2.3.2. We indicate participants anonymously as Xp_i where X stands for the study group (A being the *control group*, B being the group testing our tool), p stands for a given pair of participants, and i distinguishes between the two participants in a pair.

We report participants' answers as they were written, only fixing typos, but we do *not* alter their phrasing or grammatical structures whatsoever.

Participant	**In your own words, describe what you did today.**
$A1_1$	Proved several theorems in a pair using an interactive proof assistant.
$A1_2$	1. Proved some set and logic and operation lemmas and theorems. 2. Used a new tool to write the proofs.
$A2_1$	Tried to understand *Coq* by using an entire IDE and write some proofs.
$A2_2$	We used a proof assistant to prove theorems about lists and their properties.
$A3_1$	We used the theorem prover to prove that the functions of code did what they meant to in all cases.
$A3_2$	Today I learned about the *Coq* theorem proving language. After an introduction, I worked in a pair to solve various theorems for lists of natural numbers, and maps.
$A4_1$	Use an automated theorem prover to prove basic facts in data structures and logic.
$A4_2$	We used a tool that could define predicates, types and prove various properties about them.
$A5_1$	We learned a little about basic proofs in *Coq* then applied what we learned in the exercises.
$A5_2$	I used a proof assistant to prove several simple mathematical theorems.

Participant	**In your own words, describe what you did today.**
$B1_1$	I used a graphical tool to prove several invariants about functions/data structures in a given language (*Coq*).
$B1_2$	Proved some special cases of inferences from new definitions using induction and other primitive proof techniques. Use already proved theorems in future proofs.
$B2_1$	I used a programming language tool to prove various constructs.
$B2_2$	Proved a bunch of simple theorems about functions working on lists.
$B3_1$	Use *Coq* to prove a theorem. Reasoning the steps and select what makes sense from the options given by the tool.
$B3_2$	We were trying to use *PeaCoq* to prove a bunch of theorems by exploring all the possible strategies, such as induction, transformation, simplification, etc.
$B4_1$	We used a graphical version of an interactive proof assistant to prove some theorems involving list operations.
$B4_2$	Proved some functions using *PeaCoq*.
$B5_1$	We used the *PeaCoq* web editor interface to prove theorems on pieces of code via *Coq*. First we were given an introduction to *Coq* combined with a tutorial on using the tool, and then we worked in pairs on a series of example of proofs/theorems.
$B5_2$	I used a modified version of the *Coq* proof assistant to prove things about lists and functions over them.

Participant	**What did the experience today remind you of?**
$A1_1$	Learning to program for the first time. Lots of brute-forcing and trying things that worked in the past to see if they would work in a new context.
$A1_2$	Using induction in high school/undergrad and trying to draw insights from proving the base case to apply to the general case.
$A2_1$	Today was quite unique but I felt like writing functional programs since I tried to prove my implementation there too.
$A2_2$	It reminded me of working on large code bases where certain things appear to work by magic. Fixes/proofs were performed with little fundamental understanding.
$A3_1$	Taking a programming languages class, since the constructs are all the same.
$A3_2$	Today's experience reminded me of an intro to algorithms course.
$A4_1$	Proving mathematical theorems for research.
$A4_2$	Liquid types in Haskell. Mathematical induction.
$A5_1$	Learning programming in school and proofs in math classes.
$A5_2$	It reminded me of pen-and-paper proofs.

Participant	**What did the experience today remind you of?**
B1$_1$	Playing a logical/exploration game.
B1$_2$	1. Writing unit tests for corner cases. 2. Giving a systematic proof covering all cases.
B2$_1$	It reminded me of safe programming techniques in the way that if we can prove that our constructs work, there shouldn't be any problems at runtime.
B2$_2$	Functional programming and reasoning about it.
B3$_1$	First time to learn formal proof stuff.
B3$_2$	Decision tree.
B4$_1$	It reminded me of a puzzle game in which you have to reach a target and can use as keys some of the facts you encounter on your way there.
B4$_2$	Proving technique classes.
B5$_1$	The $\dfrac{\begin{array}{c}\text{rules}\\ \text{rules}\\ \text{rules}\end{array}}{\textit{statement}}$ format reminded me of constructions I have seen in computer science papers that have not formally learned about. The theorems we were proving, e.g. `concat xs nil = xs`, reminded in a different/more powerful direction (but requiring manual work to prove).
B5$_2$	Analysis class, proving simple-seeming things about math was similarly harder than it looked.

Participant	**Did you understand all the proofs you completed?**
$A1_1$	No.
$A1_2$	No. We figured out that there were some hammers which we should keep using (`simpl`, `reflexivity`, `rewrite`). They mostly made sense but we didn't spend time to understand the exact changes they made.
$A2_1$	Most of them. Especially the last ones were quite confusing.
$A2_2$	Not entirely. While I had conceptual knowledge of what was happening, I didn't understand it holistically.
$A3_1$	Yes.
$A3_2$	Yes.
$A4_1$	Yes.
$A4_2$	Yes.
$A5_1$	Yes.
$A5_2$	Yes.

Participant	**Did you understand all the proofs you completed?**
$B1_1$	Yes.
$B1_2$	Yes.
$B2_1$	Yes, the only one that I had trouble with was `destruct` .
$B2_2$	No.
$B3_1$	No. Sometimes it is magically done.
$B3_2$	No, but most of them.
$B4_1$	Yes (at least on a second closer inspection).
$B4_2$	No.
$B5_1$	For the most part. The last exercise (`In_concat_left`) had me a bit confused on how `destruct` worked/what it did. Once that clicked for me, what we were doing there made sense.
$B5_2$	I think so?

Participant	**Did you find the tool useful for achieving the tasks?**
$A1_1$	Error messages were mostly helpful sometimes. Sort of. Clear when something didn't work.
$A1_2$	Yes. Because it would stop me from trying to use wrong rules. Also because I could follow from one lemma to another in most cases.
$A2_1$	1. Has a simple, clean UI, 2. Doesn't require anything other than a browser.
$A2_2$	A little. It provided confidence that the task was completed correctly.
$A3_1$	Yes. I like how the right side always showed the next goal, in that it was easy to focus on one statement at a time.
$A3_2$	Yes, very.
$A4_1$	Yes, it was pretty self-explanatory. Inline code and comments very helpful.
$A4_2$	Yes.
$A5_1$	Yes, the step by step approach was helpful to visualize how the proof was coming along. It also helped with trial and error.
$A5_2$	Highlighting, previous/next were extremely useful. The information snippets would be useful if I weren't the proof author.

Participant	**Did you find the tool useful for achieving the tasks?**
B1$_1$	The tool was very useful, specifically: 1. Highlighting the diffs was helpful to explain what each tactic does, 2. Having all the options available to apply was helpful for a novice who doesn't know what the available tactics are.
B1$_2$	Yes.
B2$_1$	It definitely helped in showing the paths you could take in a proof, which is helpful if you can't think of them on the spot.
B2$_2$	I proved things without having to think too hard about it.
B3$_1$	1. As an inexperienced learner, I even don't know how to start a proof, but the tool gives me the guidance to do this. 2. After some time, when you get familiar with it, there exists some patterns to find the correct solution.
B3$_2$	I think it's useful. But it is hard to say "how" useful it is for two reasons: 1. I'm not aware of any order tools, 2. the problems are actually simple and intuitive to prove manually.

B4_1

1. Provided a broad overview of the available options (some perhaps “counter-intuitive” options that I might have thought of were presented right away),
2. fewer keystrokes
3. visual diffs are very useful for calculating how bigger / smaller our target is.

B4_2 I wouldn’t be able to achieve the tasks on my own, but that’s probably because I have no idea how to prove those things.

B5_1 Yes, absolutely. The tree interface made exploring possible tactics and seeing the results very quick and easy. I’ve never actually used *Coq* or anything like it before but with this interface I was able to get up to speed pretty quickly.

B5_2 I didn’t have to already know/remember the set of options, and it was useful to get a preview of what each one would do.

Participant	**Do you have suggestions on how to improve the tool you used today?**
A1_1	Backing up through a `Qed.` backs up the entire proof. Would prefer to step back one tactic at a time like during the proof. Right-hand pane sometimes not wide enough to see everything. Primitive sub-string matching to suggest rewrites would be useful.
A1_2	1. It should stop me from going down useless proof steps even though the rules apply. 2. It should stop me from going in circles by warning (we are not sure we did that but came close). 3. Difficult to translate intuition from base case to general case. Had to use pen-and-paper at times to write small examples.
A2_1	1. Missing some keyboard shortcuts. 2. Having a pane which shows previous proofs would be quite helpful.
A2_2	I had little confidence that I understood what was happening. Completing the same proofs later would likely result in similar trial and error. The only thing I was learning was the sort of patterns that proofs took on. Variable names were confusing and hard to work with.
A3_1	`split` should be syntax-highlighted.
A3_2	Adding auto-complete would be nice and a command list.

$A4_1$ Previously proved functions were difficult to find/see when scrolling up. I could see this being a major problem for anything complex. Could be solved with a better interface e.g. in an IDE.Not related to the software, but there's a tendency to avoid planning and just react to results of `simpl`, `rewrite`, etc.

$A4_2$

1. Bugs when we were using Firefox instead of Chrome. Tool wouldn't revert steps correctly.
2. Auto-completion of previously defined rewrite theorems

$A5_1$ Flip Next and Previous in the GUI.

$A5_2$ Some sort of search ("I want to use this") would be handy. The coloring breaks on exceptions.

Participant	**Do you have suggestions on how to improve the tool you used today?**
$\mathrm{B}1_1$	1. Highlight the border of the sub-window (code/proof) that has keyboard focus, 2. When deep in a proof tree, I kept forgetting what induction hypothesis I had accumulated so far (and I think I didn't always see them in the current goal). It would be useful to somehow get reminded of what invariants/induction hypothesis I had accumulated.
$\mathrm{B}1_2$	A first time user might find it difficult to switch between cases using "[" and "]". Some syntax in languages were not straightforward, for instance `fold`. Syntax of `map` functions were clear.
$\mathrm{B}2_1$	Maybe look ahead and see if some paths don't lead to solutions.
$\mathrm{B}2_2$	*No answer provided.*
$\mathrm{B}3_1$	*No answer provided.*
$\mathrm{B}3_2$	I guess it's definitely helpful if the tool can automatically probing some paths without clicking.
$\mathrm{B}4_1$	1. Trivialize options that lead to the target in one to two steps. 2. On the fly explanation of what the operators do.
$\mathrm{B}4_2$	It crashes when you hit the left button three or four times quickly.

B5_1 There was that one bug that we ran into a few times: going back causing the editor to crash, that made us lose progress a few times.

1. An auto-save feature would be appreciated as a guard against this (could use e.g. Local Storage API),
2. A way to "bookmark" and jump back to points in the tree would be appreciated, perhaps also some highlights of previously explored paths?

B5_2 A minor one: `intros.` should be option one (instead of `intro x.`) since it always seems like the right start.

Participant	**Explain in your own word what the `split` tactic does.**
$A1_1$	Splits up an `AND` so each side can be proven separately.
$A1_2$	Split/divides a proof of an `AND` expression into two different proof exercises.
$A2_1$	Used to prove both paths of a theorem (conjunction) with two parts.
$A2_2$	Splits a logical `and` into subgoals for each expression.
$A3_1$	Follow both branches of an `AND`, since both must be proven.
$A3_2$	Selects the terms from both sides of an `AND`.
$A4_1$	Enumerates all cases in a conjunction.
$A4_2$	Divides the `AND` predicate into two sub-problems so we can prove each separately.
$A5_1$	Splits an `and` into each side so they can be proved separately.
$A5_2$	Split conjunction.

Participant	**Explain in your own word what the `split` tactic does.**
$B1_1$	For a goal `A /\ B` introduces 2 subgoals `A` and `B`.
$B1_2$	Prove each child tree is correct.
$B2_1$	Applies some function to all elements in a set.
$B2_2$	Prove all sub-trees.
$B3_1$	Every element in a collection.
$B3_2$	Try to prove each case.
$B4_1$	Apply action to all elements of a collection.
$B4_2$	?
$B5_1$	When you have `A /\ B`, you need to prove both `A` and `B`, so split, doing `A` and then `B`.
$B5_2$	Splits an `A /\ B` goal into separate `A` and `B` goals.

Participant	**Explain in your own word what the `intro` tactic does.**
$A1_1$	Fixes some `forall` value.
$A1_2$	Introduces the different variables in the expression.
$A2_1$	Instantiates the `forall` variables.
$A2_2$	Do it first thing?
$A3_1$	Apply a `foreach`.
$A3_2$	Provides the types for the theorem we are proving.
$A4_1$	Instantiates a placeholder variable under a universal quantifier.
$A4_2$	Infers already known information from the setup to give us known or assumed values.
$A5_1$	List names of variables in `forall` and hypotheses.
$A5_2$	Instantiate `forall`.

Participant	**Explain in your own word what the `intro` tactic does.**
$B1_1$	Given a `forall x, p(x)` goal, make up a new variable name `x` and introduce it and make `p(x)` (with that specific `x`) the current sub-goal.
$B1_2$	Introduce (or define) the variables used in proof.
$B2_1$	Introduces all the components of your proof in order to use them.
$B2_2$	Introduce quantified variables (i.e. give them a name).
$B3_1$	Extend the original theorem to an easy-to-understand format.
$B3_2$	Give a start point.
$B4_1$	Add facts to your assumptions.
$B4_2$	?
$B5_1$	Take the names from the `forall` and put them in the usable rule environment. It was sort of unclear to me why we needed to do this explicitly every time.
$B5_2$	Equivalent of the mathematician's "fix", gets rid of `forall`.

Participant	**Explain in your own word what the `induction` tactic does.**
$A1_1$	Allows case analysis over inductive types.
$A1_2$	Applies induction the variable mentioned as parameter and breaks the proof into base case and general case.
$A2_1$	To use induction on a given variable. Proof by cases analysis.
$A2_2$	Perform induction with two steps/goals: base case + inductive case.
$A3_1$	Break a step into its base case(s) and recursive case(s).
$A3_2$	Performs induction on an element.
$A4_1$	Breaks a logical statement universally quantifying over a variable `x`, into cases based on `match`.
$A4_2$	Performs mathematical induction. Divides into base case, induction case. Assumes hypothesis.
$A5_1$	Breaks a list into the base case and induction step.
$A5_2$	Induct over recursive variable.

Participant	**Explain in your own word what the `induction` tactic does.**
B1$_1$	If `x` is of a given type (like ML algebraic types), do a case analysis for each of the constructors of the type. If one of them is recursive, introduce a hypothesis for its sub-part, and use it to prove the statement for the larger goal.
B1$_2$	Split into two branches: base case and induction case. Introduce a hypothesis argument called `IH` which can be invoked later in the proof.
B2$_1$	Proves that some function works for arbitrary element.
B2$_2$	Start an inductive proof (base case + step).
B3$_1$	1. Base case, 2. Induction...
B3$_2$	It's just "induction" we do... counting base case, inductive hypothesis, inductive steps.
B4$_1$	Do structural induction of a...
B4$_2$	Proof by induction.
B5$_1$	Case analysis: break e.g. a list into `nil` and `cons`, and branch along each path with an additional inductive hypothesis, proving each path (one for each possible value/case) proves the overall theorem.
B5$_2$	Splits a goal involving a recursively-defined data type into a base case and an inductive step goal (in the latter, you get the inductive hypothesis in the environment).

Participant	**Explain in your own word what the `reflexivity` tactic does.**
$A1_1$	Used when x = x.
$A1_2$	Comparing an expression with itself is true.
$A2_1$	Used to prove equality when both sides of the equality is the same (or very similar after simplifying).
$A2_2$	Attempt to reconcile an equality.
$A3_1$	When you have `x = x`, call that step finished and move on to the next.
$A3_2$	Simplifies like expressions on both sides of the equality.
$A4_1$	Uses the tautological identity to prove trivial equalities. Also does `simpl`.
$A4_2$	`A == A`.
$A5_1$	Simplifies and accepts if both sides are the same.
$A5_2$	Apply functions, find structural equality.

Participant	**Explain in your own word what the `reflexivity` tactic does.**
$B1_1$	`x = x` is trivially true.
$B1_2$	`a = a`. Also simplifies based on definition of types.
$B2_1$	Two things are equivalent.
$B2_2$	Check if two things are equal.
$B3_1$	`x = x`.
$B3_2$	Trivial equivalence.
$B4_1$	Use the reflexive property (along with some simplification).
$B4_2$	Checks if two things are the same.
$B5_1$	Prove `a = a` for some `a`.
$B5_2$	Proves `x = x` (also performs simple simplifications).

Participant	**Explain in your own word what the `rewrite` tactic does.**
$A1_1$	Used to pull in some fact proven earlier.
$A1_2$	Uses the rule provided to find a component in LHS/RHS (depending on arrow) and modify it (as per the rule).
$A2_1$	Uses a rule to rewrite a part of the proof.
$A2_2$	Pattern-match and replace.
$A3_1$	Rewrite the current step with a specified function.
$A3_2$	Applies an equality to your current goal or sub-goal.
$A4_1$	Pattern-match part of an expression using a (inductive) hypothesis in scope.
$A4_2$	Substitution.
$A5_1$	Replaces one side of an expression with the other.
$A5_2$	Replace equal terms.

Participant	**Explain in your own word what the `rewrite` tactic does.**
B1$_1$	Use a previous lemma/induction hypothesis to transform some terms.
B1$_2$	Replace parts of LHS, RHS, using the theorems proved earlier.
B2$_1$	Use previous lemma or theorems to modify the construct.
B2$_2$	Use previously proven things to rewrite the expression.
B3$_1$	Rewrite some formulas to ones that are closer to the proof's target. Better to utilize the existing hypothesis.
B3$_2$	Transformation.
B4$_1$	Use some rewrite property applied to parts of your expressions.
B4$_2$	Replaces a pattern in the text.
B5$_1$	Use a theorem/inductive hypothesis/other rewrite rule to transform part of the current node and continue with the proof.
B5$_2$	"Substitution" or "plugging int".

Participant	**Explain in your own word what the `right` tactic does.**
$A1_1$	Tells proof assistant I'm about to prove the right-hand side of an `OR` .
$A1_2$	Choose the right-side expression in a combination of Boolean expressions.
$A2_1$	To prove the right part of an `OR` expression.
$A2_2$	Take a goal with a logical `or` and solve only the right sub-expression.
$A3_1$	Follow the right branch of an `OR` , ignoring the left.
$A3_2$	Selects the right side of a logical comparison.
$A4_1$	Try to prove the second clause in a disjunction.
$A4_2$	Discard left side of the `or` predicate.
$A5_1$	Selects the right-hand side of an `or` gate to try to prove.
$A5_2$	Follow right arm of disjunction.

Participant	**Explain in your own word what the `right` tactic does.**
B1_1	Given a goal `A \/ B` try to prove `B`.
B1_2	While proving `\/`, choose the right clause to prove.
B2_1	Choose the right hand goal of an `OR` construct to continue a proof.
B2_2	Prove the right side of the `\/` expression.
B3_1	Prove the right side.
B3_2	For `A \/ B` or `A /\ B`, try to prove the right proposition.
B4_1	Prove the right part of a disjunction.
B4_2	Visits the right-hand side of an `OR` statement.
B5_1	When you have `A \/ B`, you need to prove `A` or `B`, here, we choose to prove `B` to prove the overall statement.
B5_2	Replaces `A \/ B` goal with `B`.

Participant	**I found the tool easy to understand.**				
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
$A1_1$					✓
$A1_2$				✓	
$A2_1$				✓	
$A2_2$			✓		
$A3_1$				✓	
$A3_2$					✓
$A4_1$					✓
$A4_2$			✓		
$A5_1$					✓
$A5_2$					✓
$B1_1$					✓
$B1_2$				✓	
$B2_1$					✓
$B2_2$				✓	
$B3_1$			✓		
$B3_2$					✓
$B4_1$					✓
$B4_2$		✓			
$B5_1$				✓	
$B5_2$					✓

Participant	I found the session educational.				
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
$A1_1$					✓
$A1_2$					✓
$A2_1$					✓
$A2_2$					✓
$A3_1$					✓
$A3_2$			✓		
$A4_1$				✓	
$A4_2$				✓	
$A5_1$					✓
$A5_2$					✓
$B1_1$					✓
$B1_2$				✓	
$B2_1$					✓
$B2_2$					✓
$B3_1$					✓
$B3_2$			✓		
$B4_1$					✓
$B4_2$		✓			
$B5_1$					✓
$B5_2$				✓	

Participant	**I found the session fun.**				
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
$A1_1$				✓	
$A1_2$				✓	
$A2_1$					✓
$A2_2$			✓		
$A3_1$				✓	
$A3_2$				✓	
$A4_1$				✓	
$A4_2$				✓	
$A5_1$					✓
$A5_2$				✓	
$B1_1$					✓
$B1_2$					✓
$B2_1$					✓
$B2_2$				✓	
$B3_1$					✓
$B3_2$				✓	
$B4_1$					✓
$B4_2$	✓				
$B5_1$				✓	
$B5_2$				✓	

Participant	**I would like to continue using this tool.**				
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
$A1_1$		✓			
$A1_2$			✓		
$A2_1$				✓	
$A2_2$	✓				
$A3_1$		✓			
$A3_2$				✓	
$A4_1$				✓	
$A4_2$				✓	
$A5_1$					✓
$A5_2$				✓	
$B1_1$					✓
$B1_2$					✓
$B2_1$				✓	
$B2_2$			✓		
$B3_1$			✓		
$B3_2$			✓		
$B4_1$					✓
$B4_2$			✓		
$B5_1$				✓	
$B5_2$					✓

Participant	**I would recommend this session to a friend.**				
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
$A1_1$					✓
$A1_2$				✓	
$A2_1$					✓
$A2_2$				✓	
$A3_1$				✓	
$A3_2$				✓	
$A4_1$				✓	
$A4_2$				✓	
$A5_1$					✓
$A5_2$				✓	
$B1_1$				✓	
$B1_2$				✓	
$B2_1$					✓
$B2_2$				✓	
$B3_1$				✓	
$B3_2$				✓	
$B4_1$					✓
$B4_2$	✓				
$B5_1$				✓	
$B5_2$					✓

Bibliography

[1] Umut A Acar, Guy E Blelloch, and Robert Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6): 990–1034, 2006.

[2] Jasmin Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. *Frontiers of Combining Systems*, pages 12–27, 2011.

[3] Olivier Boite. Proof reuse with extended inductive types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2004.

[4] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594304. URL http://doi.acm.org/10.1145/2594291.2594304.

[5] Magnus Carlsson. Monads for incremental computing. In *ACM SIGPLAN Notices*, volume 37, pages 26–35. ACM, 2002.

[6] Yan Chen, Joshua Dunfield, Matthew A Hammer, and Umut A Acar. Implicit self-adjusting computation for purely functional programs. In *ACM SIGPLAN Notices*, volume 46, pages 129–141. ACM, 2011.

[7] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, pages 50–66. Springer, 1990.

[8] Ł Czajka and C Kaliszyk. Hammer for Coq: Automation for dependent type theory. submitted. *URL http://cl-informatik. uibk. ac. at/cek/coqhammer*, 2017.

[9] Dominik Dietrich, Iain Whiteside, and David Aspinall. A framework for proof refactoring. In *International Conference on Logic for Programming Artificial Intelligence*

and Reasoning, pages 776–791. Springer, 2013.

[10] Denis Firsov and Wolfgang Jeltsch. Purely functional incremental computing. In *Brazilian Symposium on Programming Languages*, pages 62–77. Springer, 2016.

[11] Matthew Flatt. Binding as sets of scopes. *ACM SIGPLAN Notices*, 51(1):705–717, 2016.

[12] Stephen R Foster, William G Griswold, and Sorin Lerner. Witchdoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pages 222–232. IEEE Press, 2012.

[13] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55 (11):1382–1393, 2008.

[14] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In *ACM SIGPLAN Notices*, volume 48, pages 1–2. ACM, 2013.

[15] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[16] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[17] Wojciech Jedynak. Operational semantics of Ltac. Master's thesis, University of Wrocław, Institute of Computer Science, 2013.

[18] Andrew Kennedy. Dimension types. In *European Symposium on Programming*, pages 348–362. Springer, 1994.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[20] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[21] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Notices*, volume 49, pages 216–226. ACM, 2014.

[22] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

[23] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell refactorer, HaRe, and its API. *Electronic Notes in Theoretical Computer Science*, 141(4):29–34, 2005.

[24] Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Vıg, and Tamás Nagy. Refactoring Erlang programs. In *The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden*, volume 10, 2006.

[25] Conor McBride. Elimination with a motive. In *International Workshop on Types for Proofs and Programs*, pages 197–216. Springer, 2000.

[26] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, pages 2–15. ACM, 2017.

[27] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *ACM SIGPLAN Notices*, volume 47, pages 557–570. ACM, 2012.

[28] Shayan Najd and Simon Peyton Jones. Trees that grow. *J. UCS*, 23(1):42–62, 2017.

[29] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[30] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[31] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129. ACM, 2018.

[32] Marcel Ruegenberg. Semi-automatic proof refactoring for Isabelle. *Bacherlorarbeit in Informatik, Technische Universität München*, 2011.

[33] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Colloquium on Trees in Algebra and Programming*, pages 607–621. Springer, 1997.

[34] Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov. Towards formal proof script refactoring. *Intelligent Computer Mathematics*, pages 260–275, 2011.

[35] Thomas Williams and Didier Rémy. A principled approach to ornamentation in

ml. *Proceedings of the ACM on Programming Languages*, 2(POPL):21, 2017.

[36] Rebecca J Wirfs-Brock and Ralph E Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[37] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.